Efficient Index for Weighted Sequences

Carl Barton[1], Tomasz Kociumaka[*2], Solon P. Pissis[3], and Jakub Radoszewski[†4]

1 **The Blizard Institute, Barts and The London School of Medicine and Dentistry, Queen Mary University of London, UK**
c.barton@qmul.ac.uk
2 **Institute of Informatics, University of Warsaw, Warsaw, Poland**
kociumaka@mimuw.edu.pl
3 **Department of Informatics, King's College London, London, UK**
solon.pissis@kcl.ac.uk
4 **Institute of Informatics, University of Warsaw, Warsaw, Poland; and Department of Informatics, King's College London, London, UK**
jrad@mimuw.edu.pl

Abstract

The problem of finding factors of a text string which are identical or similar to a given pattern string is a central problem in computer science. A generalised version of this problem consists in implementing an index over the text to support efficient on-line pattern queries. We study this problem in the case where the text is *weighted*: for every position of the text and every letter of the alphabet a probability of occurrence of this letter at this position is given. Sequences of this type, also called position weight matrices, are commonly used to represent imprecise or uncertain data. A weighted sequence may represent many different strings, each with probability of occurrence equal to the product of probabilities of its letters at subsequent positions. Given a probability threshold $\frac{1}{z}$, we say that a pattern string P *matches* a weighted text at starting position i if the product of probabilities of the letters of P at positions $i, \ldots, i + |P| - 1$ in the text is at least $\frac{1}{z}$. In this article, we present an $O(nz)$-time construction of an $O(nz)$-sized index that can answer pattern matching queries in a weighted text over a constant-sized alphabet in optimal time. This improves upon the state of the art by a factor of $z \log z$ in construction time and space. Other applications of this data structure include an $O(nz)$-time construction of the weighted prefix table and an $O(nz)$-time computation of all covers of a weighted sequence, which improve upon the time complexity of the state of the art by the same factor.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases weighted sequence, position weight matrix, indexing, weighted suffix tree

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.4

1 Introduction

Finding factors of a *text* resembling a *pattern* constitutes a classical problem in computer science. Apart from its theoretical interest, it is the core computation of many applications [14] such as search engines, bioinformatics, natural language processing and database search.

* Supported by the Polish Ministry of Science and Higher Education under the "Iuventus Plus" program in 2015-2016 grant no 0392/IP3/2015/73.
† The author is a Newton International Fellow. Supported by the Polish Ministry of Science and Higher Education under the "Iuventus Plus" program in 2015-2016 grant no 0392/IP3/2015/73.

© Carl Barton, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski;
licensed under Creative Commons License CC-BY

In many situations the text can be considered as fixed and the patterns may arrive later. The algorithmic challenge is then to provide fast and direct access to all the factors of the text via the implementation of an *index*. The most widely used data structures for this purpose are the *suffix tree* and the *suffix array* [7]. These data structures can be constructed in $\mathcal{O}(n)$ time for a text of length n. Then all locations of a pattern of length m can be found in the optimal time $\mathcal{O}(m + Occ)$, where Occ is the number of occurrences.

The pattern matching problem for *uncertain* sequences has been less explored [12]. In this work we consider a type of uncertain sequences called *weighted sequences* (also known as position weight matrices, PWM). In a weighted sequence every position contains a subset of the alphabet and every letter is assigned a probability of occurrence such that at each position the probabilities sum up to 1. Such sequences are common in various applications: (i) data measurements such as imprecise sensor measurements; (ii) flexible modelling of DNA sequences such as DNA binding profiles; (iii) when observations are private and thus sequences of observations may have artificial uncertainty introduced deliberately.

In the *weighted pattern matching* (WPM) problem we are given a string of length m called a pattern, a weighted sequence of length n called a text, both over an alphabet Σ of size σ, and a *threshold probability* $\frac{1}{z}$. The task is to find all positions in the text where the fragment of length m represents the pattern with probability 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*. An $\mathcal{O}(\sigma n \log m)$-time solution for the WPM problem based on Fast Fourier Transform was proposed in [6]. This problem was also considered in [1] where a reduction to property matching in a text of size $\mathcal{O}(nz^2 \log z)$ was proposed.

In this article, we are interested in the indexing version of the WPM problem, that is, constructing an index to provide efficient procedures for answering queries related to the content of a fixed weighted sequence. In [11], the authors presented the *weighted suffix tree* allowing $\mathcal{O}(m + Occ)$-time WPM queries; the construction time and size of that data structure is $\mathcal{O}(n\sigma^{z \log z})$. A direct application of the results in [1] reduces the construction time and the size of that index to $\mathcal{O}(nz^2 \log z)$. The index structure built in [11] consists of a compacted trie of all of the factors with probability greater than or equal to $\frac{1}{z}$. A similar – though more general – indexing data structure, which assumes $z = \mathcal{O}(1)$, was presented in [4] with query time $\mathcal{O}(m + m \times Occ)$. Here we propose a tree-like data structure that is similar to the aforementioned ones which is, however, constructed and stored much more efficiently. Note that the proposed index is constructed and works for a predetermined parameter z, as opposed to the one of [4] which can additionally answer queries for $z' < z$.

Our model of computations. We assume 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.

A common assumption in practice is that $\sigma = \mathcal{O}(1)$ since the most commonly studied alphabet is $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. In this case a weighted sequence of length n has a representation of $\mathcal{O}(n)$ size. We describe the indexing data structure under this assumption. In the Conclusions Section we briefly discuss the construction of the index for larger alphabets.

Our contribution. We present an $\mathcal{O}(nz)$-time construction of an $\mathcal{O}(nz)$-sized index that answers weighted pattern matching queries in optimal $\mathcal{O}(m + Occ)$ time improving upon [1] by a factor of $z \log z$. Applications of our data structure include an $\mathcal{O}(nz)$-time construction of the *weighted prefix table* and an $\mathcal{O}(nz)$-time computation of all *covers* of a weighted sequence, which improve upon [2] and [11], respectively, by the same factor in the complexity.

Structure of the article. In Section 2 basic notation related to weighted sequences, tries and compacted tries is presented. In particular, we introduce an important notion of *extensions of solid prefixes*, which is then used to construct an intermediate data structure that is crucial to our index, called *solid factor trie*, in Section 3. The weighted index is described in Section 4. First, in Section 4.1, we show how the main component of the index, *compacted trie of maximal solid factors*, is obtained from the solid factor trie, and then, in Section 4.2, a black-box description of the weighted index together with all the auxiliary data structures is given. Section 5 contains two examples of applications of the weighted index. We end with a Conclusions Section where we sketch changes to be made to the index in the case of a superconstant-sized integer alphabet.

2 Preliminaries

Let $\Sigma = \{s_1, s_2, \ldots, s_\sigma\}$ be an alphabet. A *string* S over Σ is a finite sequence of letters from Σ. By $S[i]$, for $1 \leq i \leq |S|$, we denote the i-th letter of S. The *empty string* is denoted by ε. By $S[i..j]$ we denote the string $S[i] \ldots S[j]$ 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 $j = |S|$. A factor U of a string S is called *proper* if $U \neq S$. By S^R we denote the reversal (the mirror image) of S.

▶ **Definition 1** (Weighted sequence). A weighted sequence $X = x_1 x_2 \ldots x_n$ of length $|X| = n$ over an alphabet $\Sigma = \{s_1, s_2, \ldots, s_\sigma\}$ is a sequence of sets of pairs of the form:

$$x_i = \{(s_j,\ \pi_i^{(X)}(s_j))\ :\ j \in \{1, 2, \ldots, \sigma\}\}.$$

If the considered weighted sequence is unambiguous, we write π_i instead of $\pi_i^{(X)}$. Here, $\pi_i(s_j)$ is the occurrence probability of the letter s_j at the position $i \in \{1, \ldots, n\}$. These values are non-negative and sum up to 1 for a given i.

The *probability of matching* of a string P with a weighted sequence X, both having the same length, equals

$$\mathcal{P}(P, X) = \prod_{i=1}^{|P|} \pi_i^{(X)}(P[i]).$$

We say that a string P *matches* a weighted sequence X with probability at least $\frac{1}{z}$, denoted by $P \approx_{\frac{1}{z}} X$, if $\mathcal{P}(P, X) \geq \frac{1}{z}$. By $X[i..j]$ we denote a weighted sequence called a *factor* of X and equal to $x_i \ldots x_j$ (if $i > j$, then the factor is an empty weighted sequence). We then say that a string P *occurs* in X at position i if P matches the factor $X[i..i + |P| - 1]$. 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 considered in the article can be formulated as follows.

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

We say that P is a *(right-)maximal solid factor* of X at position i if P is a solid factor of X at position i and no string $P' = Ps$, for $s \in \Sigma$, is a solid factor of X at this position.

▶ **Fact 2** (Amir et al. [1]). *A weighted sequence has at most z different maximal solid factors starting at a given position.*

For each position of a weighted sequence X we define the *heaviest letter* as the letter with the maximum probability (breaking ties arbitrarily). By $\mathbf{X}$ we denote a string obtained from X by choosing at each position the heaviest letter. We call $\mathbf{X}$ the *heavy string* of X.

2.1 Extensions of solid factors

Let us fix a weighted sequence X of length n. If F is a solid factor of X starting at position i and ending at position j, $j \geq i-1$, then the string $F\mathbf{X}[j+1..n]$ is called *the extension of the solid factor* F. By $\mathcal{E}$ we denote the set of extensions of all solid factors of X.

▶ **Observation 3.** *$\mathcal{E}$ is exactly the set of extensions of all* maximal *solid factors of X.*

Proof. Let $F\mathbf{X}[j+1..n] \in \mathcal{E}$ be an extension of a solid factor F starting at position i and let $k \in \{j, \ldots, n\}$ be the maximum index such that $F\mathbf{X}[j+1..k]$ is a solid factor of X starting at position i. Then $M = F\mathbf{X}[j+1..k]$ is a maximal solid factor, as it cannot be extended by the most probable letter $\mathbf{X}[k+1]$, and $F\mathbf{X}[j+1..n] = M\mathbf{X}[k+1..n]$ is its extension. ◀

The following observation shows that $\mathcal{E}$ is closed under suffixes.

▶ **Observation 4.** *If $S \in \mathcal{E}$, $S \neq \varepsilon$, then the longest proper suffix S' of S also belongs to $\mathcal{E}$.*

Proof. Assume that S is an extension of a solid factor F. If $|F| \geq 1$, then S' is an extension of the longest proper suffix of F. Otherwise, S' is an extension of an empty factor. ◀

2.2 Tries

We consider rooted labeled trees with labels on edges, called *tries*. The labels are letters from Σ; edges going down from a single node have distinct labels. The root is denoted by *root*.

If T is a trie and u, v are its two nodes such that v is an ancestor of u, then by $str(u, v)$ we denote the string spelled by the edge labels on the path from u to v. We say that $\{str(u, root) : u \in T\}$ are *the suffixes of the trie* T. As usual by $lca(x, y)$ we denote the lowest common ancestor of the nodes x and y. By L_i for $i \geq 0$ we denote the i-th *level* of T that consists of nodes at depth i in the trie.

A *compacted trie* is a trie in which maximal paths whose inner nodes have degree 2 are represented as single edges with string labels. Usually such labels are not stored explicitly, but as pointers to a base string (or base strings); only the first letters are stored. The remaining nodes are called *explicit*, whereas the nodes that are removed due to compactification are called *implicit*. A well-known example of a compacted trie is a suffix tree of a string [7].

A *suffix tree of a trie* T, denoted by $\mathcal{S}(T)$, is a compacted trie of the strings $str(u, root)$ for $u \in T$; see [5, 15, 16]. The explicit nodes of $\mathcal{S}(T)$ that correspond to $str(u, root)$ for $u \in T$ are called *terminal* nodes. The string labels of the edges of $\mathcal{S}(T)$ are not stored explicitly, but correspond to upward paths in the trie T. For a node v of $\mathcal{S}(T)$, by $str(v)$ we denote the concatenation of labels of the edges from the root of $\mathcal{S}(T)$ to v.

▶ **Fact 5** (Shibuya [16]). *The suffix tree of a trie with N nodes has size $\mathcal{O}(N)$ and can be constructed in $\mathcal{O}(N)$ time.*

3 Solid factor trie

For a weighted sequence X of length n, a *solid factor trie* of X, denoted by $\mathcal{T}$, is a trie having as suffixes the *reversals of the strings from* $\mathcal{E}$. By this definition:

▶ **Observation 6.** *If S is a solid factor of X, then there exist nodes u, v in $\mathcal{T}$ such that $str(u, v) = S$.*

It turns out that the solid factor trie represents all maximal solid factors of X much more efficiently than if each of them was stored separately.

▶ **Lemma 7.** *The solid factor trie $\mathcal{T}$ has at most z nodes at each level.*

Proof. By Observation 4, each node at the level i in $\mathcal{T}$ comes from a string of length i in $\mathcal{E}$. By Observation 3 and Fact 2, there are at most z strings of length i in $\mathcal{E}$. ◀

We proceed with a construction of the solid factor trie in time linear in the size of the trie. For this, we need to equip the data structure with additional values; these enhancements will also turn out useful in the construction of the weighted index.

For each edge of the trie we store, in addition to its letter label, its probability defined as the probability of this letter at the respective position in X. If v is an ancestor of u, then by $\pi(u, v)$ we denote the product of probabilities of edges on the path from u to v. Let H be the heavy path in $\mathcal{T}$ that corresponds to $\mathbf{X}$ and let h be the leaf on this path. For each node v of $\mathcal{T}$ we retain the node $\mathit{back}(v)$ defined as $\mathit{lca}(v, h)$ and the probability $\pi\text{-}\mathit{back}(v) = \pi(v, \mathit{back}(v))$. We also denote $\mathit{str}\text{-}\mathit{back}(v) = \mathit{str}(v, \mathit{back}(v))$ (those values are not stored).

▶ **Theorem 8.** *The solid factor trie $\mathcal{T}$ of a weighted sequence X of length n can be constructed in $\mathcal{O}(nz)$ time.*

Proof. The trie is constructed by the algorithm Construct-$\mathcal{T}(X, n)$. We add new nodes to $\mathcal{T}$ level by level. First we extend the heavy path. A node v at level $i-1$ receives a child with an edge labeled by a letter s if and only if $s\,\mathit{str}\text{-}\mathit{back}(v)$ is a solid factor at position $n-i+1$; this condition is checked using the $\pi\text{-}\mathit{back}(v)$ values. Then we assign the child its values of back and $\pi\text{-}\mathit{back}$. The correctness of the algorithm follows from the claim below.

▶ **Claim.** After the i-th step of the outmost loop of the algorithm Construct-$\mathcal{T}(X, n)$, the trie's suffixes are the reversals of the strings from $\mathcal{E}$ of length at most i.

Proof. The proof goes by induction on i. The case of $i = 0$ is trivial. Let us assume that the claim holds for $i-1$ and prove that it then also holds for i. We need to show that if a node u is created by the algorithm at the i-th level, then $\mathit{str}(u, \mathit{root}) \in \mathcal{E}$ and, conversely, if $S \in \mathcal{E}$ is a string of length i, then a node u such that $\mathit{str}(u, \mathit{root}) = S$ is created by the algorithm at the i-th level. We prove the two implications separately.

(⇒) If the node u is created for some letter s, then, by the inductive hypothesis and the condition checked in the algorithm, $s\,\mathit{str}\text{-}\mathit{back}(v)$ is a solid factor of X starting at position $n-i+1$. Let j be the level of the node $\mathit{back}(v)$. Then:

$$\mathit{str}(u, \mathit{root}) = s\,\mathit{str}\text{-}\mathit{back}(v)\,\mathbf{X}[n-j+1..n] \in \mathcal{E}.$$

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

$h_0 := root$; $L_0 := \{h_0\}$;
for $i := 1$ **to** n **do**
  Create a new node h_i being a child of h_{i-1} with the letter $\mathbf{X}[n-i+1]$;
  $back(h_i) := h_i$;
  $\pi\text{-}back(h_i) := 1$;
  $L_i := \{h_i\}$;
  foreach $v \in L_{i-1}$ **do**
    foreach $s \in \Sigma$ *in order of non-increasing* $\pi^{(X)}_{n-i+1}(s)$ **do**
      if $v = h_{i-1}$ **and** $s = \mathbf{X}[n-i+1]$ **then continue**;
      if $\pi^{(X)}_{n-i+1}(s) \cdot \pi\text{-}back(v) \geq \frac{1}{z}$ **then**
        Create a new node u being a child of v with the letter s;
        $back(u) := back(v)$;
        $\pi\text{-}back(u) := \pi^{(X)}_{n-i+1}(s) \cdot \pi\text{-}back(v)$;
        $L_i := L_i \cup \{u\}$;
      else break;

(⇐) Let S' be the longest proper suffix of S. Then $S' \in \mathcal{E}$ due to Observation 4. By the inductive hypothesis, there exists a node v in L_{i-1} such that $str(v, root) = S'$. Then S is an extension of the solid factor $s\, str\text{-}back(v)$, so indeed $\pi^{(X)}_{n-i+1}(s) \cdot \pi\text{-}back(v) \geq \frac{1}{z}$ and the node u corresponding to S will be created. ◀

Let us proceed with the complexity analysis. In each step of the innermost foreach-loop (apart from the step involving a node of the heavy path), either a new node is created or the execution of the loop is interrupted. For a given i, the former takes place $|L_i|$ times in total and the latter takes place at most $|L_{i-1}|$ times in total. The whole algorithm works in $\mathcal{O}(\sum_{i=0}^{n} |L_i|) = \mathcal{O}(nz)$ time due to Lemma 7. ◀

Let us introduce additional values to $\mathcal{T}$ that enable recovering the maximal solid factors of X. For a node $u \in L_i$, by $end(u)$ we denote its ancestor v such that $str(u, v)$ is a maximal solid factor at position $n-i+1$ in X. Moreover, by $len(u)$ we denote $|str(u, v)|$.

▶ **Lemma 9.** *The values $end(u)$ and $len(u)$ for all nodes u of $\mathcal{T}$ can be computed in $\mathcal{O}(nz)$ time.*

Proof. Clearly, it suffices to focus on the *end*-pointers, as the *len*-values can be computed from these pointers in linear time if only we store for each node its level in the trie.

For each node u, $end(u)$ is an ancestor of $back(u)$ (possibly equal to $back(u)$), therefore it is located on the heavy path H. For each node $v \in H$ from the leaf h up to the root we will set the *end*-pointers for all nodes u such that $end(u) = v$. In the computation we use the following property of the pointers:

▶ **Observation 10.** *If x is an ancestor of y, then $end(x)$ is an ancestor of $end(y)$.*

A node will be called *active* if it is a descendant of v such that its *end*-pointer has not been computed yet but its children's *end*-pointers have all been computed. After a node $v \in H$ has been considered, a set A containing all the active nodes u together with the values $\pi(u, v)$ is stored. Initially the set is empty.

For the next node $v \in H$ we first update the set A. If $v = h$, then we simply insert v to A with the probability 1. Otherwise, we iterate through all the nodes u in the set A and multiply their probabilities by the probability of the edge $\pi(v', v)$ where v' is the child of v on the heavy path. Then we insert to A all the leaves in the subtrees of $\mathcal{T}$ corresponding to children of v other than v'; their probabilities in A are the values of π-*back*.

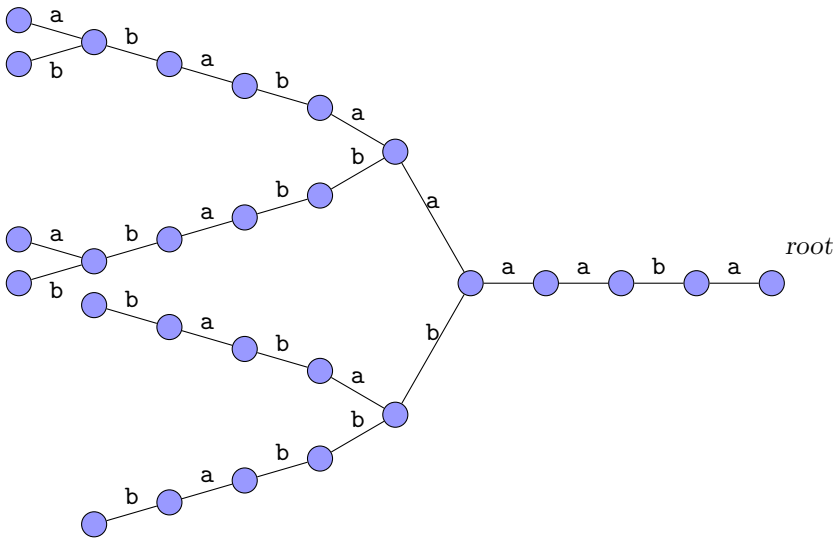
Next, we try to set the *end*-pointers for the elements of A and their ancestors. If v is the root, we simply set the pointers to the root to all the elements of A and their ancestors. Otherwise, let $w \in H$ be the parent of v. We iterate through all the elements $u \in A$ and for each of them check if $\pi(u, w) = \pi(u, v)\pi(v, w)$ is at least $\frac{1}{z}$. If so, we simply leave u in A for the next iterations. Otherwise, we set $\mathit{end}(u) = v$. If u was the last child of its parent for which we computed the *end*-pointer, we add the parent of u to A. In order to efficiently check this condition, each node counts its children whose *end*-pointer is yet to be determined.

The correctness of the algorithm follows from Observation 10. The running time is proportional to the total number of times a node from A is visited. When a node $v \in H$ is considered, for each node $u \in A$ either its *end*-pointer is set, which obviously happens at most $|\mathcal{T}| = \mathcal{O}(nz)$ times in total, or $\mathit{str}(u, v)$ corresponds to a left-maximal solid factor ending at position corresponding to the level of v in $\mathcal{T}$, which can happen at most z times by Fact 2. This implies $\mathcal{O}(nz)$ time complexity of the whole algorithm. ◀

▶ **Example 11.** The figure below shows an example of $\mathcal{T}$ for $z = 4$ and

$$X = [(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)]\mathtt{bab}[(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)][(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)]\mathtt{aaba}.$$

Among a few heavy strings of X, we can select $\mathbf{X} = \mathtt{ababaaaaba}$.



4 Construction of the weighted index

Our index for a weighted sequence X is based on a compacted trie of all maximal solid factors of X. We first show how this compacted trie can be constructed from the suffix tree $\mathcal{S}(\mathcal{T})$ of the solid factor trie $\mathcal{T}$. Next, we describe in detail all the components of the resulting weighted index.

4.1 Compacted trie of maximal solid factors

First of all, from Fact 5 and Lemma 7 we obtain an efficient construction of $\mathcal{S}(\mathcal{T})$:

▶ **Lemma 12.** *The suffix tree of the solid factor trie can be constructed in $\mathcal{O}(nz)$ time.*

The trie $\mathcal{T}$ represents more than the (maximal) solid factors of X, and so does $\mathcal{S}(\mathcal{T})$. However, the *len*-values that we computed in $\mathcal{T}$ let us delimit the maximal solid factors. Using them we can transform $\mathcal{S}(\mathcal{T})$ into a compacted trie $\mathcal{T}'$ of all maximal solid factors of X. Assume that in $\mathcal{S}(\mathcal{T})$ each terminal node stores, as its label, the starting position in X of the string from $\mathcal{E}$ that it represents (i.e., its depth). Then in $\mathcal{T}'$ a terminal's label is a list of starting positions in X of occurrences of the corresponding maximal solid factor.

▶ **Theorem 13.** *A compacted trie $\mathcal{T}'$ of all maximal solid factors of a weighted sequence X of length n can be constructed in $\mathcal{O}(nz)$ time.*

Proof. We start by constructing the solid factor trie $\mathcal{T}$ of X, together with the *len*-values, and its suffix tree $\mathcal{S}(\mathcal{T})$. By Theorem 8 and Lemmas 9 and 12, these steps take $\mathcal{O}(nz)$ time. Now it suffices to properly trim $\mathcal{S}(\mathcal{T})$. For a terminal node v in $\mathcal{S}(\mathcal{T})$ corresponding to $str(u, root)$ in $\mathcal{T}$, as $len(v)$ we store $len(u)$. Then we need to "lift" such a terminal node to depth $len(v)$ in $\mathcal{S}(\mathcal{T})$. In practice we proceed as follows.

For an (explicit or implicit) node u of $\mathcal{S}(\mathcal{T})$, by $maxlen(u)$ we denote the maximum value of $len(v)$ for a descendant terminal node v. As a result of trimming we leave only those (explicit or implicit) nodes u for which $maxlen(u)$ is at least as big as their depth in the trie; we call such nodes *relevant* nodes and the remaining nodes *irrelevant* nodes.

This procedure can be implemented in linear time. Indeed, the *maxlen*-values for all explicit nodes can be computed with a single bottom-up traversal. In another bottom-up traversal, we consider all irrelevant explicit nodes. Let w be such a node and let v be its explicit parent. Assume that v is located at depth d. If $maxlen(w) \leq d$, w is removed from $\mathcal{S}(\mathcal{T})$ and its label is appended to its parent's label. Otherwise, we cut the edge connecting v and w at depth $maxlen(w)$ and move the irrelevant node w there, making it relevant. ◀

4.2 The weighted index

As already mentioned, our weighted index is based on the compacted trie $\mathcal{T}'$ of all maximal solid factors of X. We also need to store the solid factor trie $\mathcal{T}$ which lets us access the string labels of the edges of the compacted trie. For convenience we extend each maximal solid factor in $\mathcal{T}'$ by a symbol $\$ \notin \Sigma$. As a result, each maximal solid factor corresponds to a leaf in $\mathcal{T}'$ which is labeled with a list of starting positions of its occurrences in X.

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 leaves of the trie $\mathcal{T}'$ in pre-order. Each node v stores, as $OL(v)$, the occurrence list of leaves 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 colored range listing problem.

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

▶ **Fact 14** (Muthukrishnan [13])**.** *A data structure for the colored range listing problem of $\mathcal{O}(N)$ size can be constructed in $\mathcal{O}(N + S)$ time and answers queries in $\mathcal{O}(k + 1)$ time where k is the number of distinct elements reported.*

For all nodes of $\mathcal{T}'$ 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).

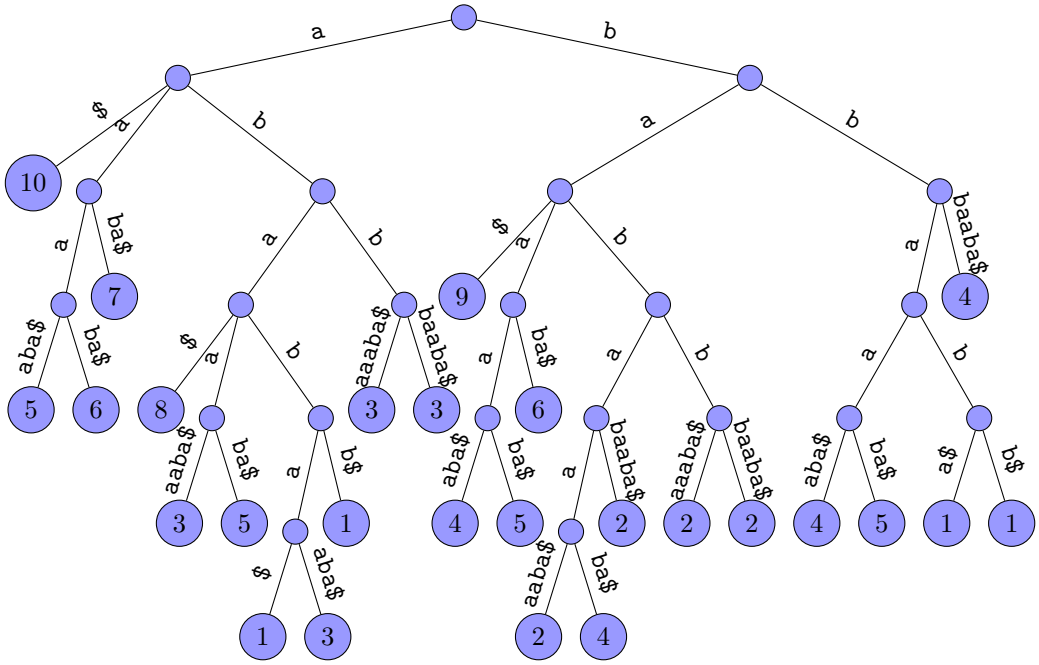
▶ **Fact 15** (Color set size, Hui [10]). *Given a rooted tree of size N with L leaves colored from $[1..S]$, in $\mathcal{O}(N+S)$ time one can find for each node u the number of distinct leaf colors in the subtree of u.*

We denote the resulting data structure as $\mathcal{I}$.

▶ **Theorem 16.** *The index $\mathcal{I}$ for a weighted sequence X can be constructed in $\mathcal{O}(nz)$ time. It answers decision and counting variants of weighted pattern matching queries in $\mathcal{O}(m)$ time, and, if required, reports all occurrences of the pattern in $\mathcal{O}(m+|Occ_{\frac{1}{z}}(P,X)|)$ time.*

Proof. The compacted trie $\mathcal{T}'$ can answer queries if $Occ_{\frac{1}{z}}(P,X) \neq \emptyset$ in $\mathcal{O}(m)$ time. We can use Fact 15 to equip each explicit node with the number of 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 colored range listing, we can also report $Occ_{\frac{1}{z}}(P,X)$ in time proportional to the number of reported elements. ◀

▶ **Example 17.** The figure below shows the trie $\mathcal{T}'$ constituting the weighted index for the solid factor trie $\mathcal{T}$ shown in Example 11.



5 Applications of the weighted index

In this section we present two non-trivial applications of the weighted index. In both cases we improve the time complexity of the previously known results by a factor of $z \log z$.

5.1 Weighted longest common prefixes and weighted prefix table

For a weighted sequence X of length n and a pair of indices i, j, $1 \leq i, j \leq n$, 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 18.** *Given a weighted sequence X of length n, after $\mathcal{O}(nz)$-time preprocessing we can answer $wlcp(i,j)$ queries for any $1 \leq i,j \leq n$ in $\mathcal{O}(z)$ time.*

Proof. For each position i in X we precompute the list of leaves $L(i)$ of the weighted index $\mathcal{I}$ that contain i in their occurrence lists. Prior to that, all leaves are numbered in pre-order, and the elements of $L(i)$ are stored in this order. By Fact 2, $|L(i)| \leq z$ for each i.

Observe that $wlcp(i,j)$ is the maximum depth of a lowest common ancestor (lca) of a leaf in $L(i)$ and a leaf 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 [7]) implies that, computing $wlcp(i,j)$, it suffices to consider pairs of leaves that are adjacent in the resulting list.

▶ **Claim.** If l_1, l_2 and l_3 are three leaves 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 [3, 9]. ◀

The weighted prefix table $WPT[1..n]$ of X is defined as $WPT[i] = wlcp(1,i)$; see [2]. As a consequence of Theorem 18 we obtain an $\mathcal{O}(nz)$-time algorithm for computing this table. It outperforms the algorithm of [2], which works in $\mathcal{O}(nz^2 \log z)$ time.

▶ **Theorem 19.** *The weighted prefix table WPT of a given weighted sequence of length n can be computed in $\mathcal{O}(nz)$ time.*

5.2 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 [11]. More formally, if we define *maxgap* of an ordered set $A = \{a_1, \ldots, a_k\}$ (with $a_1 < \ldots < a_k$) as

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

then P is a cover of X if and only if

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

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 [11]. Here we show an algorithm that works in $\mathcal{O}(nz)$ time.

The algorithm of [11] uses a data structure (which we denote here by $\mathcal{D}$) to store a multiset of elements A from the set $\{2, \ldots, n\}$ allowing three operations:

1. initialisation with a given multiset of elements A;
2. computing $maxgap(\mathcal{D}) = maxgap(A \cup \{1, n+1\})$ 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 $C[1..n+1]$ that counts the multiplicity of each element; (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 [11], 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 a node v to its child w on the path, from $\mathcal{D}(v)$ we remove all elements from $OL(w')$ for w' being children of v other than w. Afterwards for the node w we perform the following check, which we call *cover-check*(w): if $maxgap(\mathcal{D}(w)) \leq depth(w)$, report the covers being prefixes of $str(w)$ of length $[\max(maxgap(\mathcal{D}(w)), depth(v)+1)..depth(w)]$. 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 (Fact 2).

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 trie from a starting prefix node and passing only through non-starting prefix nodes will be called a *covering* path. 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 20.** *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. The proofs of the following two lemmas are deferred to the full version of the article.

▶ **Lemma 21.** *$\mathcal{D}(v)$ for all starting prefix nodes v can be computed in $\mathcal{O}(nz)$ time.*

▶ **Lemma 22.** *The values $maxgap(\mathcal{D}(v))$ for all prefix nodes can be computed in $\mathcal{O}(nz)$ time.*

▶ **Theorem 23.** *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 22 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. ◀

6 Conclusions

We have presented an index for weighted pattern matching queries which for a constant-sized alphabet has $\mathcal{O}(nz)$ size and admits $\mathcal{O}(nz)$ construction time. It answers queries in optimal $\mathcal{O}(m + Occ)$ time. We have also mentioned two applications of the weighted index. Our index outperforms the previously existing solutions by a factor of $z \log z$ in the complexity.

Generalization to integer alphabets. Let us briefly discuss how to adapt our index to a general integer alphabet. The size of the input is then the total length R of the lists in the representation of the weighted sequence. In the construction of the solid factor trie we need the list at each position to be ordered according to the probabilities of letters. As the size of each list to be sorted is $\min(z, \sigma)$ (at most z letters can have probability at least $\frac{1}{z}$), the

sorting requires $\mathcal{O}(R \log \min(\sigma, z))$ time. The construction of a suffix tree of a tree of [16] works for any integer alphabet. Finally, our weighted index is a compacted trie with children of a node being indexed by the letter of the alphabet. Hence, to avoid an increase of the complexity of a query for a particular child of a node, for a general alphabet one requires to store a hash table of children. With perfect hashing [8] the complexity does not increase but becomes randomized (Las Vegas, running time w.h.p.).

An open question is whether our weighted index, constructed for a predetermined z, can be adapted to answer weighted pattern matching queries for $z' < z$, as it is in the case of [4].

References

1 Amihood Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. *Theor. Comput. Sci.*, 395(2-3):298–310, April 2008. doi:10.1016/j.tcs.2008.01.006.

2 Carl Barton and Solon P. Pissis. Linear-time computation of prefix table for weighted strings. In Florin Manea and Dirk Nowotka, editors, *Combinatorics on Words, WORDS 2015*, volume 9304 of *LNCS*, pages 73–84. Springer, 2015. doi:10.1007/978-3-319-23660-5.

3 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *Latin American Symposium on Theoretical Informatics, LATIN 2000*, volume 1776 of *LNCS*, pages 88–94. Springer Berlin Heidelberg, 2000. doi:10.1007/10719839_9.

4 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. doi:10.5441/002/edbt.2016.37.

5 Dany Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theor. Comput. Sci.*, 191(1-2):131–144, 1998. doi:10.1016/S0304-3975(96)00319-2.

6 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.

7 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007.

8 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

9 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

10 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. doi:10.1007/3-540-56024-6_19.

11 Costas S. Iliopoulos, Christos Makris, Yannis Panagis, Katerina Perdikuri, Evangelos Theodoridis, and Athanasios K. Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundam. Inform.*, 71(2-3):259–277, 2006. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi71-2-3-07.

12 Yuxuan Li, James Bailey, Lars Kulik, and Jian Pei. Efficient matching of substrings in uncertain sequences. In Mohammed Javeed Zaki, Zoran Obradovic, Pang-Ning Tan,

Arindam Banerjee, Chandrika Kamath, and Srinivasan Parthasarathy, editors, *SIAM International Conference on Data Mining, SDM 2014*, pages 767–775. SIAM, 2014. doi:10.1137/1.9781611973440.88.

13 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. URL: http://dl.acm.org/citation.cfm?id=545381.

14 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001. doi:10.1145/375360.375365.

15 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. doi:10.1007/3-540-46632-0_24.

16 Tetsuo 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):1061–1066, 2003.