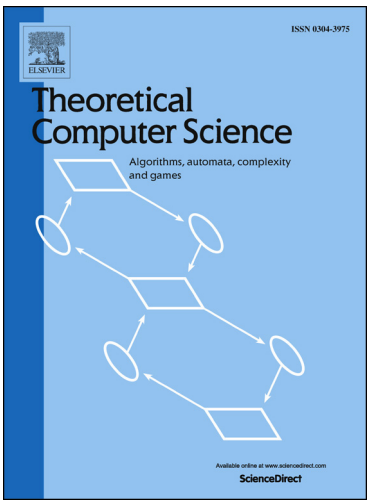
Accepted Manuscript



Linear-time computation of prefix table for weighted strings & applications

Carl Barton, Chang Liu, Solon P. Pissis

PII: S0304-3975(16)30068-8
DOI: http://dx.doi.org/10.1016/j.tcs.2016.04.029
Reference: TCS 10738

To appear in: *Theoretical Computer Science*

Received date: 29 July 2015
Revised date: 29 March 2016
Accepted date: 25 April 2016

Please cite this article in press as: C. Barton et al., Linear-time computation of prefix table for weighted strings & applications, *Theoret. Comput. Sci.* (2016), http://dx.doi.org/10.1016/j.tcs.2016.04.029

Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

00 (2016) 1–15

Linear-Time Computation of Prefix Table for Weighted Strings & Applications[☆]

Carl Barton[a], Chang Liu[b], Solon P. Pissis[b]

[a]*The Blizard Institute, Barts and The London School of Medicine and Dentistry, Queen Mary University of London, London, UK*
[b]*Department of Informatics, King's College London, London, UK*

Abstract

The *prefix table* of a string is one of the most fundamental data structures of algorithms on strings: it determines the longest factor at each position of the string that matches a prefix of the string. It can be computed in time linear with respect to the size of the string, and hence it can be used efficiently for locating patterns or for regularity searching in strings. A *weighted string* is a string in which a set of letters may occur at each position with respective occurrence probabilities. Weighted strings, also known as *position weight matrices* or *uncertain strings*, naturally arise in many biological contexts; for example, they provide a method to realise approximation among occurrences of the same DNA segment. In this article, given a weighted string x of length n and a constant *cumulative weight threshold* $1/z$, defined as the minimal probability of occurrence of factors in x, we present an $O(n)$-time algorithm for computing the prefix table of x. Furthermore, we outline a number of applications of this result for solving various problems on non-standard strings, and present some preliminary experimental results.

Keywords: algorithms on strings, weighted strings, uncertain sequences

1. Introduction

An *alphabet* Σ is a finite non-empty set of size σ, whose elements are called *letters*. A *string* on an alphabet Σ is a finite, possibly empty, sequence of elements of Σ. The zero-letter sequence is called the *empty string*, and is denoted by ε. The *length* of a string x is defined as the length of the sequence associated with the string x, and is denoted by $|x|$. We denote by $x[i]$, for all $0 \leq i < |x|$, the letter at index i of x. Each index i, for all $0 \leq i < |x|$, is a position in x when $x \neq \varepsilon$. It follows that the i-th letter of x is the letter at position $i-1$ in x.

The *concatenation* of two strings x and y is the string of the letters of x followed by the letters of y; it is denoted by xy. A string x is a *factor* of a string y if there exist two strings u and v, such that $y = uxv$. Consider the strings x, y, u, and v, such that $y = uxv$, if $u = \varepsilon$ then x is a *prefix* of y, if $v = \varepsilon$ then x is a *suffix* of y. Let x be a non-empty string and y be a string, we say that there exists an *occurrence* of x in y, or more simply, that x *occurs in* y, when x is a factor of y. By $y[i\,..\,j]$ we denote the factor $y[i]\dots y[j]$ of string y. Every occurrence of x can be characterised by a position in y; thus we say that x occurs at the *starting position* i in y when $y[i\,..\,i+|x|-1] = x$.

Single nucleotide polymorphisms, as well as errors introduced by wet-lab sequencing platforms during the process of DNA sequencing, can occur in some positions of a DNA sequence. In some cases, these uncertainties can be

[☆]A preliminary version of this article appeared in the Proceedings of Combinatorics on Words - 10th International Conference, WORDS 2015, LNCS, Springer.

Email addresses: c.barton@qmul.ac.uk (Carl Barton), chang.2.liu@kcl.ac.uk (Chang Liu), solon.pissis@kcl.ac.uk (Solon P. Pissis)

accurately modelled as a *don't care* letter [1]. However, in other cases they can be more subtly expressed, and, at each position of the sequence, a probability of occurrence can be assigned to each letter of the nucleotide alphabet; this process gives rise to a *weighted string* or a *position weight matrix*. For instance, consider a IUPAC-encoded [2] DNA sequence, where the ambiguity letter `M` occurs at some position of the sequence, representing either base `A` or base `C`. This gives rise to a weighted DNA sequence, where at the corresponding position of the sequence, we can assign to each of `A` and `C` an occurrence probability of 0.5.

A weighted string x of length n on an alphabet Σ is a finite sequence of n sets. Every $x[i]$, for all $0 \leq i < n$, is a set of ordered pairs $(s_j, \pi_i(s_j))$, where $s_j \in \Sigma$ and $\pi_i(s_j)$ is the probability of having letter s_j at position i. Formally, $x[i] = \{(s_j, \pi_i(s_j)) | s_j \neq s_\ell \text{ for } j \neq \ell, \text{ and } \sum_j \pi_i(s_j) = 1\}$. A letter s_j *occurs* at position i of a weighted string x if and only if the *occurrence probability* of letter s_j at position i, $\pi_i(s_j)$, is greater than 0. A string u of length m is a *factor* of a weighted string if and only if it occurs at starting position i with *cumulative occurrence probability* $\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) > 0$. Given a *cumulative weight threshold* $1/z \in (0, 1]$, we say that factor u is *z-valid*, or equivalently that factor u has a z-valid occurrence, if it occurs at starting position i and $\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) \geq 1/z$. A position $i \in \{0, \ldots, n-1\}$ of x is called *solid* if x_i contains exactly one pair $(s_j, \pi_i(s_j))$, with $\pi_i(s_j) = 1$. For succinctness of presentation, for a non-solid position i, the set of pairs is denoted by $[(s_{j_1}, \pi_i(s_{j_1})), \ldots, (s_{j_k}, \pi_i(s_{j_k}))]$; for a solid position i it is simply denoted by the letter s_j with $\pi_i(s_j) = 1$.

A great deal of research has been conducted on weighted strings for pattern matching [3, 4], for computing various types of regularities [5, 6, 7, 8], for indexing [3, 9], and for alignments [10, 11]. The efficiency of most of the proposed algorithms relies on the assumption of a given *constant* cumulative weight threshold defining the minimal probability of occurrence of factors in the weighted string.

Similar to the standard setting [12, 13], we can define the prefix table for a weighted string. Given a weighted string x of length n and a constant cumulative weight threshold $1/z$, we define the *prefix table* WP of x as follows:

$$\mathsf{WP}[i] = \begin{cases} |u| & \text{if } i = 0 \text{ and } u \text{ is the longest } z\text{-valid prefix of } x. \\ |v| & \text{if } 0 < i < n \text{ and } v \text{ is the longest } z\text{-valid prefix of } x \text{ with a } z\text{-valid} \\ & \text{occurrence at } i. \end{cases}$$

For large alphabets it makes sense to perform a simple filtering on x to filter out letters with occurrence probability less than $1/z$. This is required as if the alphabet is not of fixed size, we may have many letters with low occurrence probability that are not of interest. We simply read the entire string and keep only those letters with probability greater than or equal to $1/z$; these are at most z for each position, so still constant. We are thus left with a data structure of size $O(zn)$: a weighted string consisting of n sets of ordered pairs of size $O(z)$ each; the entire stage takes time $O(\sigma n)$. For clarity of presentation, in the rest of this article, we assume that the string resulting from this stage is the input weighted string x, and consider the following problem.

WeightedPrefixTable
Input: a weighted string x of length n and a constant integer $z > 0$
Output: prefix table WP of x

Example 1. *Let* $x = \mathtt{aab}[(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)][(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)]\mathtt{bab}$ *and* $z = 4$. *Then we have*

i :	0	1	2	3	4	5	6	7
$WP[i]$:	8	1	0	5	1	0	1	0.

$WP[3] = 5$ *since* $\mathtt{aabbab}$ *has a z-valid occurrence at position* 0 *with probability* $1/4 \geq 1/4$ *and a z-valid occurrence at position* 3 *with probability* $1/4 \geq 1/4$. □

The main contribution of this article is the following.

Theorem 1. *Problem* WeightedPrefixTable *can be solved in time* $O(n)$.

A preliminary version of this article appeared first in [14].

2. Auxiliary Data Structures and Properties

We first start by developing basic tools and auxiliary data structures for processing weighted strings. These tools are interesting in their own right. We next present some properties of the prefix table of a weighted string.

2.1. Basic Tools and Auxiliary Data Structures

Fact 1. *Any factor of a z-valid factor of x is also z-valid.*

We perform a colouring stage on x, similar to the one before the construction of the weighted suffix tree [9], which assigns a colour to every position in x according to the following scheme:

- mark position i *black* (b), if *none* of the occurring letters at position i has probability of occurrence greater than $1 - 1/z$.
- mark position i *grey* (g), if *one* of the occurring letters at position i has probability of occurrence greater than $1 - 1/z$ and less than 1.
- mark position i *white* (w), if *one* of the occurring letters at position i has probability of occurrence 1.

Notice that if $z \geq 2$, then at every white and grey position there is only one relevant letter since only one letter can have probability of occurrence greater than $1 - 1/z \geq 1/2$, whereas in a black position there may be several letters to consider. However, if $z < 2$ there are no valid letters with probability of occurrence at most $1 - 1/z$ since $1 - 1/z < 1/z$ and therefore no black positions. Therefore for the rest of this article we assume $z \geq 2$. This stage can be trivially performed in time $O(zn)$.

Lemma 1 ([9]). *Any z-valid factor of x contains at most $\lfloor \log z / \log(\frac{z}{z-1}) \rfloor$ black positions.*

From Lemma 1 we know that any z-valid factor contains a *constant* number of black positions; for the rest of the article, we denote this constant by $\ell = \lfloor \log z / \log(\frac{z}{z-1}) \rfloor$. We can also see that any z-valid factor of a weighted string is *uniquely* determined by the letters it has at black positions: *any* white or grey position is common to all z-valid factors that contain the same positions. Hence an occurrence of a z-valid factor of x can be memorised as a tuple $< i, j, s >$, where i is the starting position in x, $j \geq i$ is the ending position in x, and s is a set of ordered pairs (c, p) denoting that letter $c \in \Sigma$ occurs at black position p, $i \leq p \leq j$. This representation requires space $O(\ell)$ per occurrence by Lemma 1; it can also be used to efficiently compute the occurrence probability of any factor of x in time proportional to the number of black positions it contains.

We start by computing an array BP of integers, such that $\mathsf{BP}[i]$ stores the smallest position greater than i that is marked black; otherwise (if no such position exists) we set $\mathsf{BP}[i] = n$. We next view x as a sequence of the form $u_0 b_0 u_1 b_1 \dots u_{k-1} b_{k-1} u_k$, such that u_0, u_k are (possibly empty) strings on Σ, $u_1, \dots, u_{k-1}$ are non-empty strings on Σ, and $b_0, \dots, b_{k-1}$ are maximal sequences consisting only of black positions. We compute an array FP of factor probabilities, such that $\mathsf{FP}[i + q]$ stores the occurrence probability of $u_j[0\,..\,q]$, for all $0 \leq q < |u_j|$, $0 \leq j \leq k$, starting at position i of x; otherwise (if no such factor starts at position i) we set $\mathsf{FP}[i] = 0$. For an example, see Table 1. Both arrays BP and FP can be trivially computed in time $O(n)$.

Given a range of positions in x, the black positions indices within this range, and the letters at black positions, that is, given a tuple $< i, j, s >$, we can easily compute the occurrence probability of the respective factor: we take the probability of any factor between black positions, the probability of the letters at black positions, the probability of any leading or trailing segments, and, finally, we multiply them all together. Notice that we can deal with any leading segments by computing the differences between prefixes via division. This takes time proportional to the number of black positions within the factor, and is thus $O(\ell)$ for any z-valid factor.

Example 2. *Consider the weighted string x in Table 1 and $z = 2$. We wish to determine the probability of the factor starting at position 2 and ending at position 7, with the two black positions as letters* a *and* a*; that is factor* $< 2, 7, \{(4, \mathtt{a}), (7, \mathtt{a})\} >$. *We determine the probability of factor* tt *starting at position* 2 *by taking* $\mathsf{FP}[3]/\mathsf{FP}[1] = 1$. *We take the probability of factor* tc *starting at position* 5*, which is given by* $\mathsf{FP}[6] = 1$. *The probability of the two black positions are 0.5 and 0.5. By multiplying all of these together we get the occurrence probability of factor* $< 2, 7, \{(4, \mathtt{a}), (7, \mathtt{a})\} >$ *which is* 0.25. □

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	a	c	t	t	(a, 0.5)	t	c	(a, 0.5)	t	t	(a, 0.6)
					(c, 0.5)			(t, 0.5)			(t, 0.4)
Colour	w	w	w	w	b	w	w	b	w	w	g
FP	1	1	1	1	0	1	1	0	1	1	0.6
BP	4	4	4	4	7	7	7	11	11	11	11

Table 1: Weighted string x and $z = 2$, colouring, array FP with cumulative occurrence probabilities between black positions, and array BP with black positions indices; the letter t at position 10 has already been filtered out since $0.4 < 1/z$

We consider a *maximal factor* of a weighted string to be a z-valid factor that cannot be extended to the right and preserve its validity.

Lemma 2 ([3]). *At most z maximal factors start at any position of x.*

Intuitively, this is because their probabilities are at least $1/z$ each and they sum up to at most 1.

Example 3. *Consider the following weighted string x and let $z = 10$.*

$$a[(a, 0.5), (c, 0.1), (g, 0.2), (t, 0.2)]\, t[(a, 0.5), (g, 0.5)]\, c.$$

We want to generate all maximal factors starting at position 0 of x. We start at position 0 with a. We extend at position 1 on the right and get aa, ac, ag, at. We extend at position 2 and get aat, act, agt, att. We extend at position 3 and get aata, aatg, agta, agtg, atta, attg. Note that factor act cannot be extended further on the right and preserve its validity. Finally we terminate the extension at position 4 and get aatac, aatgc, agtac, agtgc, attac, attgc. Therefore we get a total of seven maximal factors starting at position 0. These factors are:

$$aatac, aatgc, act, agtac, agtgc, attac, attgc.$$

Note that by Lemma 2, we can have no more than $z = 10$ maximal factors starting at position 0. □

For a weighted string x of length n, by $\mathsf{lcve}(x[i\,..\,n-1], x[j\,..\,n-1])$ we denote the length of the longest common z-valid prefix (or longest common z-valid extension) of $x[i\,..\,n-1]$ and $x[j\,..\,n-1]$.

Proposition 1. *Given a factor v of x with two z-valid occurrences $< i, j, s >$ and $< i', j', s' >$, $\mathsf{lcve}(x[i\,..\,n-1], x[i'\,..\,n-1])$ having v as a prefix can be computed in time $O(\ell + \ell z m)$, where $m = \mathsf{lcve}(x[i\,..\,n-1], x[i'\,..\,n-1]) - |v|$.*

Proof Let π_i and $\pi_{i'}$ be the probability of occurrence of the common z-valid factor v, starting at positions i and i' of x, respectively. We can compute π_i and $\pi_{i'}$ in time $O(\ell)$ using $< i, j, s >$ and $< i', j', s' >$ with the technique outlined above (using array FP). We may then proceed by comparing letters from positions $j + 1$ and $j' + 1$ onwards. We have a number of cases to consider. Suppose that of the two positions we compare:

- *no* position is black; then we carry on comparing the letters and updating the probabilities accordingly.
- *one* position is black and the other position is either white or grey; we check if the single letter at the white or grey position occurs at the black position and update the probabilities accordingly.
- *both* positions are black; we consider all occurring letters that match and continue extending all corresponding z-valid factors. By Lemma 2, the number of these combinations of letters at black positions is at most z; by Lemma 1 the number of black positions in each combination is at most ℓ.

We terminate this procedure when we have no match or the probability threshold is violated. It is clear to see that the work at any position is no more than $O(\ell z)$. □

Suppose that we want to compute $\mathsf{lcve}(x[i\,..\,n-1], x[i'\,..\,n-1])$. Proposition 1 can be implemented via maintaing two lists E_i, $E_{i'}$ of tuples $< s, \pi, ptr >$, where s is a set of ordered pairs (c, p) denoting that letter $c \in \Sigma$ occurs at black position p of x, π is the occurrence probability of the z-valid factor represented by the tuple, and *ptr* is a pointer from the tuple of E_i (resp. $E_{i'}$) to the tuple representing the same factor in list $E_{i'}$ (resp. E_i). Note that both lists are maintained sorted according to element s due to the fact that the set of pairs (p, c) are ordered by the definition of the weighted string. By Lemmas 1 and 2 the size of each list is $O(\ell z)$.

Example 4. *Consider the following weighted strings and let $z = 16$.*

$$x[i\,..\,n-1] = [(a, 0.5), (c, 0.5)]\,a[(a, 0.5), (c, 0.5)]\,c[(a, 0.5), (c, 0.5)]\ldots$$
$$x[i'\,..\,n-1] = [(a, 0.5), (c, 0.5)][(a, 0.5), (c, 0.5)]\,c[(a, 0.5), (c, 0.5)]\,c\ldots$$

For the first five positions the lists are as follows.

$$E_i =< \{(a, 0), (c, 2), (c, 4)\}, 1/8, 0 >, < \{(c, 0), (c, 2), (c, 4)\}, 1/8, 1 >$$
$$E_{i'} =< \{(a, 0), (a, 1), (c, 3)\}, 1/8, 0 >, < \{(c, 0), (a, 1), (c, 3)\}, 1/8, 1 > .$$

Hence we know that the longest common z-valid extension is 5 up to this point. Let the next position be as follows.

$$x[i + 5] = [(a, 0.5), (c, 0.5)]$$
$$x[i' + 5] = [(a, 0.5), (c, 0.5)].$$

We need to update the lists as follows.

$$E_i =< \{(a, 0), (c, 2), (c, 4), (a, 5)\}, 1/16, 0 >, < \{(a, 0), (c, 2), (c, 4), (c, 5)\}, 1/16, 1 >,$$
$$< \{(c, 0), (c, 2), (c, 4), (a, 5)\}, 1/16, 2 >, < \{(c, 0), (c, 2), (c, 4), (c, 5)\}, 1/16, 3 > \quad (1)$$

$$E_{i'} =< \{(a, 0), (a, 1), (c, 3), (a, 5)\}, 1/16, 0 >, < \{(a, 0), (a, 1), (c, 3), (c, 5)\}, 1/16, 1 >,$$
$$< \{(c, 0), (a, 1), (c, 3), (a, 5)\}, 1/16, 2 >, < \{(c, 0), (a, 1), (c, 3), (c, 5)\}, 1/16, 3 > . \quad (2)$$

Hence we know that the longest common z-valid extension is 6 up to this point. □

The following crucial observation follows directly from Fact 1.

Observation 1. *Let $u = av$, for some $a \in \Sigma$ and some string v, be a z-valid factor of a weighted string x of length n, starting at position i, $0 \le i < n - 1$, of x. Then factor v starting at position $i + 1$ is also z-valid.*

By using Observation 1, the length of the longest z-valid factor starting from each position of x for each combination of letters at black positions can be easily maintained. That is, for each position i, we maintain a list L_i of tuples $< s, \pi, len >$, where s is a set of ordered pairs (c, p) denoting that letter $c \in \Sigma$ occurs at black position p of x, π is the occurrence probability of the z-valid factor represented by the tuple, and *len* is the length of the z-valid factor represented by the tuple. List L_i is sorted according to element s. List L_0 can be constructed by generating all maximal factors starting at position 0. It is sorted according to element s due to the fact that the set of pairs (p, c) are ordered by the definition of the weighted string. We can easily obtain list L_i from L_{i-1}, for all $1 \le i < n$, using Observation 1. By Lemmas 1 and 2 the size of each list is $O(\ell z)$. After obtaining list L_i, we can sort it according to element s using string mergesort in time $O(z\ell + z \log z)$ [15]. Note that we can easily remove any duplicate elements from this sorted list if required. By Lemma 10 (see Section 3), $O(z\ell + z \log z) = O(z\ell)$. We obtain the following lemma.

Lemma 3. *Lists $L_0, L_1, \ldots, L_{n-1}$ can be computed in time and space $O(z\ell n)$.*

Example 5. *Consider the following weighted string x and let $z = 8$.*

$$[(a, 0.5), (c, 0.5)]\,a[(a, 0.5), (c, 0.5)]\,c[(a, 0.5), (c, 0.5)][(g, 0.5), (t, 0.5)][(a, 0.1), (t, 0.9)]\ldots$$

List L_0 is as follows.

$$\begin{aligned} L_0 =< &\{(a,0),(a,2),(a,4)\},1/8,5>,<\{(a,0),(a,2),(c,4)\},1/8,5>,\\ &<\{(a,0),(c,2),(a,4)\},1/8,5>,<\{(a,0),(c,2),(c,4)\},1/8,5>,\\ &<\{(c,0),(a,2),(a,4)\},1/8,5>,<\{(c,0),(a,2),(c,4)\},1/8,5>,\\ &<\{(c,0),(c,2),(a,4)\},1/8,5>,<\{(c,0),(c,2),(c,4)\},1/8,5>. \end{aligned} \quad (3)$$

Hence we know that the longest z-valid factor starting at position 0 of x for each combination of letters at black positions is 5. We obtain list L_1 from L_0 via removing the leftmost (black) position, updating the probabilities, and extending to the right.

$$\begin{aligned} L_1 =< &\{(a,2),(a,4),(g,5)\},1/8,5>,<\{(a,2),(c,4),(g,5)\},1/8,5>,\\ &<\{(c,2),(a,4),(g,5)\},1/8,5>,<\{(c,2),(c,4),(g,5)\},1/8,5>,\\ &<\{(a,2),(a,4),(t,5)\},1/8,5>,<\{(a,2),(c,4),(t,5)\},1/8,5>,\\ &<\{(c,2),(a,4),(t,5)\},1/8,5>,<\{(c,2),(c,4,(t,5))\},1/8,5>. \end{aligned} \quad (4)$$

Hence we know that the longest z-valid factor starting at position 1 of x for each combination of letters at black positions is 5. After obtaining list L_1, we can sort it according to element s using string mergesort. □

The main idea of our algorithm is based on the efficient computation of an auxiliary array P of n integers, which for some position i stores the length of the longest common prefix (or longest common extension) of strings $x[i\,..\,n-1]$ and $x[0\,..\,n-1]$ as if in all black positions of x we had a *don't care* letter—a letter that does not belong in Σ and it matches itself and any other letter in Σ.

Computing array P in a naïve way could take as much as $O(n^2)$ time; the transitive properties used in the standard setting [12] do not hold due to don't care letters. We will show here that it is possible to compute array P in time $O(\ell n)$. The *critical* observation for this computation is that no entry in WP can be greater than the length of the longest z-valid prefix of x containing at most ℓ black positions: this is clear from Lemma 1. This means that we only need to compute the values of P for this longest z-valid prefix; this will allow us to efficiently compute the values of WP for all positions later on.

We now describe the method for the efficient computation of P. Let x' be the string obtained by replacing in x: (a) each black position with a *unique* letter $\$_h$ *not* in Σ; and (b) each grey position with the only letter in that position with probability of occurrence greater than $1 - 1/z$. Hence x' is of the form $x'_0 b_0 x'_1 b_1 \ldots x'_{k-1} b_{k-1} x'_k$, such that x'_0, x'_k are (possibly empty) strings on Σ, $x'_1, \ldots, x'_{k-1}$ are non-empty strings on Σ, and $b_0, \ldots, b_{k-1}$ are maximal sequences consisting only of letters $\$_h$. Notice that a letter comparison involving any $\$_h$ causes a *mismatch* since they are unique letters that do not belong in Σ. For the weighted string x and string x', array P can be formally defined as follows.

$$\mathsf{P}[i] = \begin{cases} n & \text{if } i = 0. \\ |v| & \text{if } 0 < i < n \text{ and } v \text{ is the longest prefix of } x' \text{ with an occurrence at } i \text{ with} \\ & \text{at most } \ell \text{ mismatches and } |v| \le |u|, \text{ where } u \text{ is the longest } z\text{-valid prefix} \\ & \text{of } x. \\ \infty & \text{if } 0 < i < n \text{ and } v \text{ is the longest prefix of } x' \text{ with an occurrence at } i \text{ with} \\ & \text{at most } \ell \text{ mismatches and } |v| > |u|, \text{ where } u \text{ is the longest } z\text{-valid prefix} \\ & \text{of } x. \end{cases}$$

For a string y of length n, by $\mathsf{lce}(y[i\,..\,n-1], y[j\,..\,n-1])$ we denote the length of the longest common prefix (longest common extension) of factors $y[i\,..\,n-1]$ and $y[j\,..\,n-1]$. The following fact specifies a well-known efficient data structure over y answering such queries. It consists of the suffix array with its inverse [16], the longest common prefix array [17], and a succinct data structure for range minimum queries over the longest common prefix array [18].

Fact 2. *Let y be a string of length n. After $O(n)$-time and $O(n)$-space preprocessing, one can compute $\mathsf{lce}(y[i\,..\,n-1], y[j\,..\,n-1])$ for all $0 \le i < n$, $0 \le j < n$ in time $O(1)$.*

Fact 2 can be applied on string x' to efficiently compute array P. We therefore obtain the following.

Lemma 4. *Array P can be computed in time $O(\ell n)$ and additional space $O(n)$.*

Proof The value of $\mathsf{P}[0]$ is trivially set to n. The computation of the longest z-valid prefix of x can be done in time $O(zn)$: we pick the highest probability per position to determine this length. For any position $i > 0$ of array P, we make at most ℓ lce-queries to find subsequent mismatching positions of $x'[0\,..\,n-1]$ and $x'[i\,..\,n-1]$, starting from $\mathsf{lce}(x'[0\,..\,n-1], x'[i\,..\,n-1])$. If at least one of the letters is a $\$_h$ letter we continue extending as this does not constitute a legitimate mismatch. If both letters are in Σ we stop extending as this constitutes a legitimate mismatch. By Fact 2 each query requires time $O(1)$ after $O(n)$-time and $O(n)$-space preprocessing. We immediately stop if any of the queries exceeds the boundary of x'. Otherwise, we make a final lce-query to check if the remaining parts match exactly. If, for any position $i > 0$, the length of the extension becomes greater than the length of the longest z-valid prefix of x', we terminate the extension and set $\mathsf{P}[i] = \infty$. Each entry in P is updated only once, and so we achieve the claim. □

Lemma 5. *$P[i] \ge WP[i]$, for all $0 \le i < n$.*

Proof For those entries in P that are ∞ the claim is obvious. Those others entries in P are computed ignoring letters at black positions: should all those black positions match their corresponding positions and the probability threshold is not violated then $\mathsf{P}[i] = \mathsf{WP}[i]$; should any of those positions not match or the probability threshold is violated then $\mathsf{P}[i] > \mathsf{WP}[i]$. □

2.2. Properties of the Prefix Table

The method for computing table WP proceeds by determining $\mathsf{WP}[i]$ by increasing values of the position i on x. We introduce, the index i being fixed, two values g and f that constitute the key elements of our method. They satisfy the following relations

$$g = \max\{j + \mathsf{WP}[j] : 0 < j < i\} \tag{5}$$

and

$$f \in \{j : 0 < j < i \text{ and } j + \mathsf{WP}[j] = g\}. \tag{6}$$

We note that g and f are defined when $i > 1$. We note, moreover, that if $g < i$ we have then $g = i - 1$, and that on the contrary, by definition of f, we have $f < i \le g$.

Lemma 6. *Let $f < i < g$, u be a factor of x with two z-valid occurrences at positions 0 and f, $|u| = g - f$, and $WP[i - f] < g - i$. Then we can compute $\mathsf{lcve}(x, x[i\,..\,g-1])$ in time $O(\ell z)$.*

Proof If $\mathsf{WP}[i - f] < g - i$ then there exist two factors v_1 and v_2, possibly $v_1 \neq v_2$, of u, $|v_1| = |v_2| = \mathsf{WP}[i - f]$, occurring at positions 0, f and $i - f$ and i, respectively. By Fact 1 factors v_1 and v_2 are z-valid. By the definition of table WP there does not exist another z-valid factor v, $|v| > |v_1|$, occurring at positions 0 and $i - f$. Let v_3 be the longest common z-valid prefix of x and $x[i\,..\,g-1]$, i.e. $\mathsf{lcve}(x, x[i\,..\,g-1]) = |v_3|$. We have two cases: $v_1 = v_2$ and $v_1 \neq v_2$.

In case $v_1 = v_2$, it holds that v_3, $|v_1| \le |v_3| \le g - i$, occurs at positions 0 and i. By Lemmas 1 and 2, we can determine, in time $O(\ell z)$, the length of v_3 using array P, $|v_3| \le \mathsf{P}[i]$ (see Lemma 5), letter comparisons *only* at black positions, and the sorted lists L_0 and L_i (see Lemma 3). Specifically, by Lemma 5 we know where is the next legitimate mismatch, hence this deals with the *white positions*. This check can be done in constant time. For all *black positions*, we need to make letter comparisons. This can be done in time $O(\ell z)$ by the combination of Lemmas 1 and 2. After we have decided which combination of letters at black positions match, we need to check what is the maximal length of the factors involving these matches. As these combinations can be by the definition of the weighted string sorted and lists L_0 and L_i are also sorted, this takes time no more than $O(\ell z)$. This deals with the fact that we also have *grey positions* which are treated as white by array P.

In case $v_1 \neq v_2$ letter comparisons are required to determine the length of v_3. By Lemma 1 and triangle inequality, since v_1, v_2, and v_3 are z-valid factors, a prefix of v_1 may differ to a prefix of v_3 by at most 2ℓ (black) positions: v_1,

occurring at position 0, differs to v_2, occurring at position $i-f$, by at most ℓ positions; and a prefix of v_2, occurring at position i, differs to a prefix of v_3, occurring at position i, by at most ℓ positions. Each position has *at most* z occurring letters. Therefore, by Lemmas 1 and 2, we can either determine the actual value of $\mathsf{lcve}(x, x[i\,..\,g-1]) = |v_3| < |v_1|$ or determine that $|v_1| \le |v_3| \le g-i$ in time $O(\ell z)$. In case $|v_1| \le |v_3| \le g-i$, by Lemmas 1 and 2, we can determine, in time $O(\ell z)$, the length of v_3 using array P, $|v_3| \le \mathsf{P}[i]$ (see Lemma 5), letter comparisons *only* at black positions, and the sorted lists L_0 and L_i (see Lemma 3).

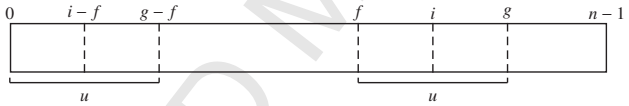
Example 6. *Consider the following weighted string x and let $z = 64$.*

								f							g
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$x[i]$	g	(a, 0.5) (c, 0.5)	g	(g, 0.5) (t, 0.5)	(a, 0.5) (t, 0.5)	g	a	g	c	g	(g, 0.5) (t, 0.5)	(c, 0.5) (t, 0.5)	g	(a, 0.5) (t, 0.5)	c
$\mathsf{P}[i]$	15	5	6	5	2	9	0	7	0	5	5	4	2	2	0
$\mathsf{WP}[i]$	15	0	1	3	0	3	0	7	0	1	4	0	2	0	0

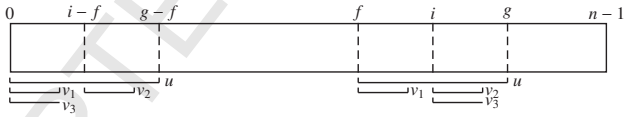
Further let $i = 10 < g = 14$, $u = \mathtt{gcgttga}$ be the z-valid factor occurring at positions 0 and $f = 7$, $|u| = g - f = 7$, and $\mathsf{WP}[i-f] = 3 < g - i = 4$. Factor $v_1 = \mathtt{gcg}$ occurs at positions 0 and $f = 7$ and factor $v_2 = \mathtt{ttg}$ occurs at positions $i - f = 3$ and $i = 10$. In this case $v_1 \neq v_2$. We apply Lemma 6.

Factor v_1, occurring at position 0, has a black position at index 1. Factor v_2, occurring at position 3, has two black positions at indices 3 and 4. Factors v_2 and v_3, starting at position 10, have two black positions at indices 10 and 11. Therefore we know that we have to compare the letters at positions 0 and 1 to the letters at positions 10 and 11, respectively, and that positions 2 and 12 are white and therefore $x[2] = x[12]$. There exist such letters that match and therefore the prefix of length $|v_1|$ of v_3 is $\mathtt{gcg}$. And since $\mathsf{P}[10] = 5$, by Lemma 5, we know that $|v_3| \le 5$. We can determine $|v_3|$ via comparing black positions: 3 to 13. Therefore we determine that $\mathsf{lcve}(x, x[i\,..\,g-1]) = 4$. □

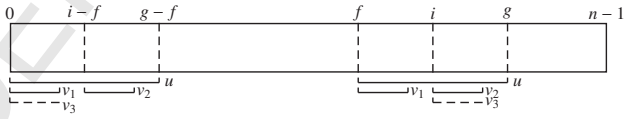
For a graphical illustration of the proof inspect Figure 1. □



(a) String $x = x[0\,..\,n-1]$, u occurs at positions 0 and f, and $\mathsf{WP}[f] = |u|$



(b) If $v_1 = v_2$ then $|v_3| = \mathsf{lcve}(x, x[i\,..\,g-1]) \ge |v_1|$



(c) If $v_1 \neq v_2$ then letter comparisons at black positions are required to determine the length of v_3

Figure 1: Illustration of Lemma 6

Lemma 7. *Let $f < i < g$, u be a factor of x with two z-valid occurrences at positions 0 and f, $|u| = g - f$, and $\mathsf{WP}[i-f] > g - i$. Then we can compute $\mathsf{lcve}(x, x[i\,..\,g-1])$ in time $O(\ell z)$.*

Proof If $\mathsf{WP}[i-f] > g-i$ then there exists a factor v, possibly $v \neq u$, $|v| = \mathsf{WP}[i-f]$, occurring at positions 0 and $i-f$. By the definition of WP factor v is z-valid and there does not exist another z-valid factor v', $|v'| > |v|$, occurring at

positions 0 and $i - f$. Let v_1 be the longest common z-valid prefix of x and $x[i\,..\,g-1]$, i.e. $\mathsf{lcve}(x, x[i\,..\,g-1]) = |v_1|$. We have two cases: $v = u$ and $v \neq u$.

In case $v = u$, it holds that v_1, $|v_1| = g - i$, occurs at positions 0 and i, and hence $\mathsf{lcve}(x, x[i\,..\,g-1]) = |v_1|$.

In case $v \neq u$, letter comparisons are required to determine the length of v_1. By Lemma 1 and triangle inequality, since u, v, and v_1 are z-valid factors, v_1 may differ to some prefix of u by at most 2ℓ (black) positions: the prefix of length $g - i$ of u, occurring at position 0, differs to the suffix of length $g - i$ of u in at most ℓ positions; and the suffix of length $g - i$ of u, occurring at position i, differs to v_1 in at most ℓ positions. Each position has *at most* z occurring letters. Therefore, by Lemmas 1 and 2, and using the sorted lists L_0 and L_i (see Lemma 3) we can either determine the actual value of $\mathsf{WP}[i] = \mathsf{lcve}(x, x[i\,..\,g-1]) < g - i$ or determine that $\mathsf{lcve}(x, x[i\,..\,g-1]) = g - i$ in time $O(\ell z)$.

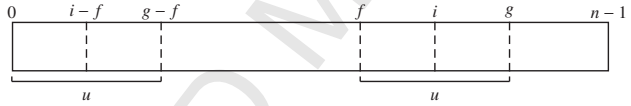
Example 7. *Consider the following weighted string x and let $z = 8$.*

										f							g
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$x[i]$	a	c	g	a	(c, 0.5)	(a, 0.5)	a	t	c	a	c	g	a	(c, 0.5)	(a, 0.5)	c
					(t, 0.5)	(g, 0.5)								(t, 0.5)	(g,0.5)	

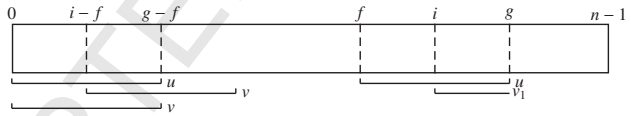
Further let $i = 12 < g = 15$, $u = \mathtt{acgata}$ be the z-valid factor occurring at positions 0 and $f = 9$, $|u| = g - f = 6$. Factor $v = \mathtt{acgat}$, $|v| = \mathsf{WP}[i - f] = 5 > g - i = 3$, occurs at positions 0 and $i - f = 3$. In this case $u \neq v$. We apply Lemma 7.

The prefix of length $g - i = 3$ of $u = \mathtt{acgata}$, occurring at position 0, differs to the suffix of length $g - i = 3$ of u in two positions. The suffix of length $g - i = 3$ of u, occurring at position $i = 12$, differs to factor v_1, occurring at positions 0 and $i = 12$, in two positions. Therefore we know that we have to compare the letters at positions 1 and 2 to the letters at positions 13 and 14, respectively, and that positions 0 and 12 are white and therefore $x[0] = x[12]$. There exist such letters that match and therefore we determine that $\mathsf{lcve}(x, x[i\,..\,g-1]) = g - i = 3$. □

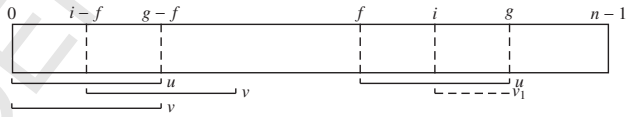
For a graphical illustration of the proof inspect Figure 2. □



(a) String $x = x[0\,..\,n-1]$, u occurs at positions 0 and f, and $\mathsf{WP}[f] = |u|$



(b) If $u = v$ then $|v_1| = \mathsf{lcve}(x, x[i\,..\,g-1])$



(c) If $u \neq v$ then letter comparisons at black positions are required to determine the length of v_1

Figure 2: Illustration of Lemma 7

Similar to Lemma 7 we can obtain the following.

Lemma 8. *Let $f < i < g$, u be a factor of x with two z-valid occurrences at positions 0 and f, $|u| = g - f$, and $\mathsf{WP}[i - f] = g - i$. Then we can compute $\mathsf{lcve}(x, x[i\,..\,g-1])$ in time $O(\ell z)$.*

3. Algorithm

We can now present Algorithm WeightedPrefixTable for computing table WP.

Algorithm *WeightedPrefixTable*$(x, n, 1/z)$
  WP$[0\,..\,n-1] \leftarrow 0$;
  WP$[0] \leftarrow |u|$ where u is the longest z-valid prefix of x;
  Compute lists $L_0, L_1, \ldots, L_{n-1}$ and array P;
  $g \leftarrow 0$;
  foreach $i \in \{1, \ldots, n-1\}$ **do**
    if $i < g$ **and** $\mathsf{lcve}(x, x[i\,..\,g-1]) < g - i$ **then**
      WP$[i] \leftarrow \mathsf{lcve}(x, x[i\,..\,g-1])$;
    else
      $f \leftarrow i$; $g \leftarrow \max\{g, i\}$;
      while $g < n$ **and** *there exists* $c \in \Sigma$ *occurring at positions* g *and* $g - f$ **and** *there exists a common z-valid prefix, say v,* $|v| = g - f$*, of* $x[0\,..\,n-1]$ *and* $x[i\,..\,n-1]$*, such that vc is a z-valid factor starting at position* 0 *and* i **do**
        $g \leftarrow g + 1$;
      WP$[i] \leftarrow g - f$;
  return *WP*;

Lemma 9. *Algorithm* WeightedPrefixTable *correctly computes table* WP.

Proof The computation of WP[0] is, by definition, correct. The variables f and g satisfy Equations 5 and 6 at each step of the execution of the loop. For i being fixed and satisfying the conditions $i < g$ *and* $\mathsf{lcve}(x, x[i\,..\,g-1]) < g - i$, the algorithm applies the Lemmas 6-8 which produce a correct computation: $\mathsf{lcve}(x, x[i\,..\,n-1]) = \mathsf{lcve}(x, x[i\,..\,g-1])$. It remains thus to check that the computation is correct otherwise. But in this case, we compute $\mathsf{lcve}(x, x[i\,..\,n-1]) = |x[f\,..\,g-1]| = g - f$ which is, by definition, the value of WP$[i]$. Therefore, Algorithm WeightedPrefixTable correctly computes table WP. □

In [3] it was shown that $\ell = O(z \log z)$. Here we refine this to an exact bound.

Lemma 10. *Let* $z \geq 2$*. Then* $\ell \leq z \log z$.

Proof By Lemma 1 we know that $\ell = \lfloor \frac{\log z}{\log(\frac{z}{z-1})} \rfloor$. For $z > 1$, we must show that:

$$\ell = \Big\lfloor \frac{\log z}{\log(\frac{z}{z-1})} \Big\rfloor = \Big\lfloor \frac{\log z}{\log(z) - \log(z-1)} \Big\rfloor \leq z \log z. \text{ Or equivalently that:}$$

$$\frac{\log z(z \log z - z \log(z-1) - 1)}{\log z - \log(z-1)} > 0.$$

Clearly the above is true if and only if: $z \log z - z \log(z-1) - 1 > 0$. There is a discontinuity at $z = 1$; after this it is *always* positive and the following holds:

$$\lim_{z \to \infty} z \log z - z \log(z-1) - 1 = 0.$$

□

Lemma 11. *Given a weighted string* x *of length* n *and a constant cumulative weight threshold* $1/z$ *Algorithm* WeightedPrefixTable *requires time* $O(n)$.

Proof The computation of WP[0] can be trivially done in time $O(n)$. Lists $L_0, L_1, \ldots, L_{n-1}$ can be computed in time $O(n)$, for $\ell = O(1)$, by Lemma 3 and Lemma 10. Array P can be computed in time $O(n)$, for $\ell = O(1)$, by Lemma 4 and Lemma 10. As the value of g never decreases and that it varies from 0 to at most n, we have at most $O(n)$ times where the condition of the inner loop is satisfied; and by applying Proposition 1 (see also Example 4) the total time required for these checks is $O(n)$. Each time the condition is not satisfied leads to the next step of the outer loop; and there are at most $n-1$ of them. Thus $O(n)$ checks on the overall. All other instructions, by Lemmas 6-8, take constant time for each value of i giving a total time of $O(n)$. □

Lemmas 9 and 11 imply the main result of this article (Theorem 1). By Lemma 10 we obtain the following.

Corollary 1. *Given a weighted string x of length n and a cumulative weight threshold $1/z$ Algorithm* WeightedPrefixTable *requires time $O(nz^2 \log z)$.*

Notice that the pre-computation of lists $L_0, L_1, \ldots, L_{n-1}$ and array P allows for a simpler formulation of this algorithm, where for each $i \in \{1, \ldots, n-1\}$ we compute WP[i] separately, without the use of variables g anf f. The structure of Algorithm WeightedPrefixTable, however, allows for using previous computations (see Lemma 7) and for an efficient (in practical terms) implementation where the information implied by lists $L_0, L_1, \ldots, L_{n-1}$ and array P is computed on the fly.

4. Applications

In this section, we outline a number of applications of Theorem 1 for solving various problems on *non-standard* strings.

General Pattern Matching on Weighted Strings

String pattern matching is a fundamental task in text processing, bioinformatics, and numerous other applications. It consists in finding all factors of a text string y of length n that match (or are similar to) a pattern string x of length m. Pattern matching on weighted strings [3, 4] and the related *probabilistic profile* matching (for a definition, see [19]) have been considered in a number of contexts.

In particular, this kind of non-standard strings have found important applications to model probabilistic ancestral sequences at internal nodes of a phylogenetic tree—currently the best-known method to obtain information about the ancestral sequence [20, 21] and for generating probabilistic profiles by summarising the common regions of a set of related sequences, which can then be used to search an entire database efficiently [22, 23].

In the literature, the authors have considered the case where either *only* the text is weighted or *only* the pattern is weighted. Here we introduce the *General Weighted Pattern Matching* problem, where both the pattern and the text may be weighted, as follows. Given a weighted string x of length m, a weighted string y of length $n > m$, and a cumulative weight threshold $1/z$, the problem consists in finding all positions in y where some z-valid factor of length m of x has a z-valid occurrence. It can be solved as follows. We first compute the prefix table WP of the weighted string xy in time $O(m+n)$, for *constant* z, by applying Theorem 1. The space complexity is also $O(m+n)$ for constant z. All matching positions in y can then be found by going through WP from left to right. There exists a z-valid occurrence of some z-valid factor of length m of x starting at position $i-m$ in y, for all $m \le i \le n$, *iff* WP[i] $\ge m$.

Prefix Table to Border Table for Weighted Strings

A border of a nonempty string x of length n is a factor u, $|u| < n$, of x that is both a prefix and a suffix of x. The *border table* B = B[$0\,..\,n-1$] of x gives the length B[i] of the longest border of every prefix $x[0\,..\,i]$, $0 \le i < n$, of x. It is computed by an elegant algorithm in time $O(n)$ [24].

The border table is a fundamental data structure used for pattern matching [25], for finding regularities [26, 27], for computing overlaps between strings [13], and for many other applications. Here we define the border table of a weighted string and show how it can be computed efficiently.

Given a weighted string x of length n and a cumulative weight threshold $1/z$, we define a *border* of x to be a z-valid factor u of x, $|u| < n$, that has a z-valid occurrence *both* as a prefix and as a suffix of x. The *border table* WB = WB[$0\,..\,n-1$] of a weighted string x gives the length WB[i] of the longest border of every prefix $x[0\,..\,i]$,

$0 \leq i < n$, of x. It can be computed as follows. We first compute table WP of the weighted string x in time $O(n)$, for *constant* z, by applying Theorem 1, and then compute table WB of x in time $O(n)$ by using the prefix table to border table conversion algorithm by Barton *et al* [13] or by Bland *et al* [28].

Suffix/Prefix Overlap of Two Weighted Strings

In paired-end sequencing, a fragment of DNA is sequenced forward and backwards, resulting in two short reads with an unknown sequence in the middle of approximately known length. Paired-end reads provide some benefits of longer read lengths without actually requiring the technology to accurately sequence reads of that length. Paired-end reads can also simplify the detection of certain genetic variations by seeing how and where each read in the pair is aligned. When performing paired-end sequencing it had previously been the case that the length of each read would be less than half the length of the DNA fragment being sequenced. Recent improvements have increased the read lengths such that when a DNA fragment is sequenced the resulting reads may overlap. The overlap between pairs means that it is possible to merge the two reads into a single long read of high quality [29]. The chance that the sequencer has introduced an error increases exponentially the further along the read, so merged paired-end reads allow for far greater confidence in the latter half of the read than would normally be possible. Merging paired-end reads may also be beneficial for reasons including correcting sequencing errors, higher quality single reads, and larger initial fragments for *de novo* assembly. It is therefore important to consider the problem of finding the longest suffix/prefix overlap between two strings [30, 31].

Using the corresponding quality scores [29], two reads can be modeled as weighted strings, where at each position of the sequences, a probability of occurrence can be assigned to each letter of the nucleotide alphabet. Given weighted strings x and y of lengths m and $n \geq m$, respectively, and a cumulative weight threshold $1/z$, the suffix/prefix overlap problem consists in finding the longest z-valid suffix of x that has a z-valid occurrence as a prefix of y. It can be solved as follows. We first compute table WP of the weighted string yx in time $O(m + n)$, for *constant* z, by applying Theorem 1, and then compute table WB of yx in time $O(m + n)$ by using the prefix table to border table conversion algorithm by Barton *et al* [13] or by Bland *et al* [28]. The longest z-valid suffix of x having a z-valid occurrence as a prefix of y is then specified by $\mathsf{WB}[m + n - 1]$. Note that we may need to trim the resulting border to ensure that it is not longer than $\min\{|x|, |y|\}$.

Prefix Table for Indeterminate Strings

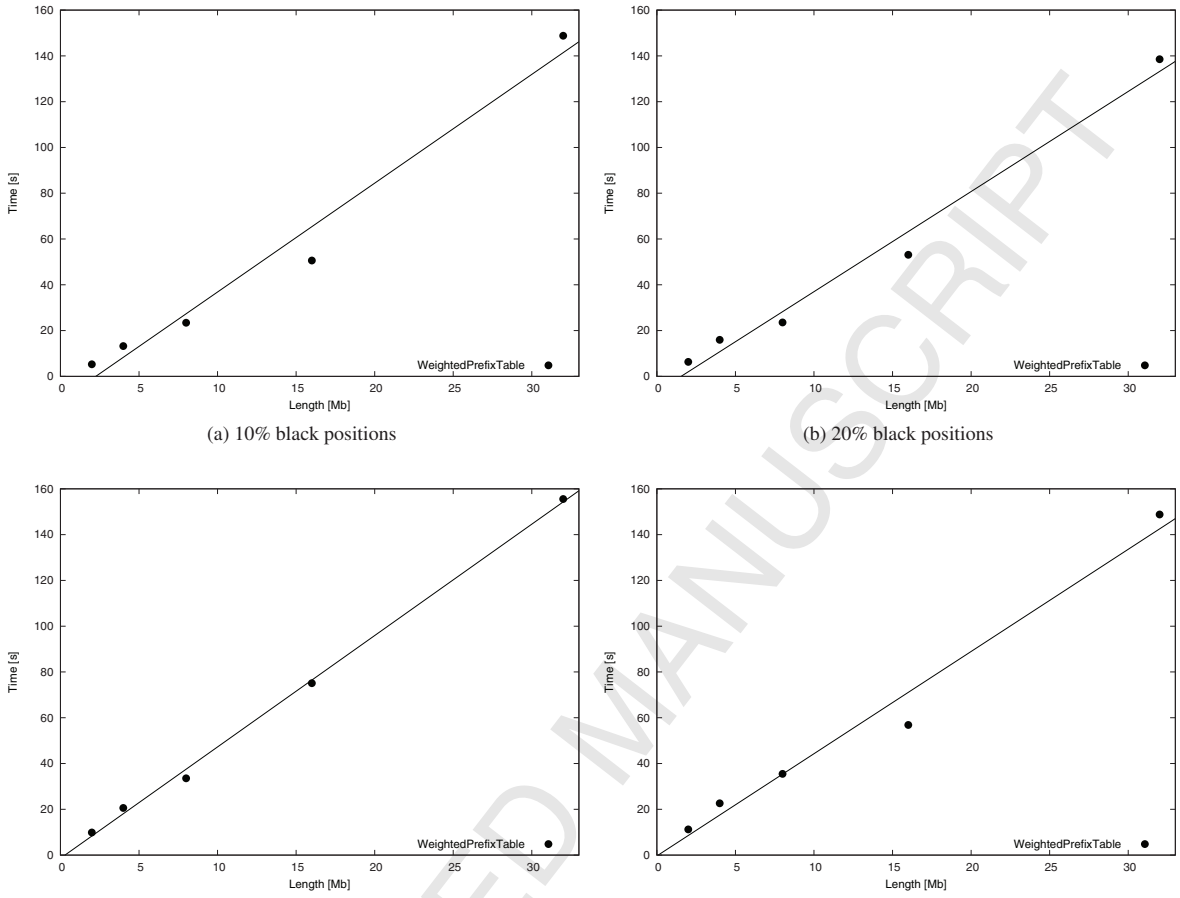
An *indeterminate string* x of length n on an alphabet Σ is a finite sequence of n sets, such that $x[i] \subseteq \Sigma$, $x[i] \neq \emptyset$, for all $0 \leq i < n$ [1, 32]. Similar to weighted strings, if $|x[i]| = 1$, that is, $x[i]$ represents a single letter of Σ, we say that i is a *solid* position. We say that two indeterminate strings x and y *match*, denoted by $x \approx y$, if $|x| = |y|$ and for each $i = 0, \ldots, |x| - 1$, we have $x[i] \cap y[i] \neq \emptyset$.

The *prefix table* $\mathsf{IP} = \mathsf{IP}[0\,..\,n-1]$ of an indeterminate string x is defined as an array of integers, such that $\mathsf{IP}[i] = p$ *iff* p is the largest integer such that $x[i\,..\,i + p - 1] \approx x[0\,..\,p - 1]$. A *conservative indeterminate string* x is an indeterminate string whose maximum number of non-solid positions is bounded by a *constant* integer $k > 0$. Here we show how to compute the prefix table for conservative indeterminate strings efficiently.

Any (conservative) indeterminate string x of length n can be represented by a weighted string x' of length n such that $(a, \frac{1}{|x[i]|}) \in x'[i]$ *iff* $a \in x[i]$ and setting $z = \sigma^k$. We can compute table WP of the weighted string x' for $\sigma = O(1)$, $k = O(1)$, and $z = \sigma^k = O(1)$ in time $O(n)$ by applying Theorem 1; then table WP is the prefix table IP of x. Having computed IP, all aforementioned applications follow for conservative indeterminate strings as well.

5. Implementation and Experimental Results

Algorithm WeightedPrefixTable was implemented as a program to compute the prefix table of a weighted string. The program was implemented in the `C++` programming language and developed under GNU/Linux operating system. The input parameters are a weighted string x of length n in the form of a $\sigma \times n$ matrix of probabilities, and integer z, to be used as the cumulative weight threshold $1/z$. The output is the prefix table of x. The implementation is distributed under the GNU General Public License (GPL), and it is available at `http://github.com/YagaoLiu/WPT`. The experiments were conducted on a Desktop PC using one core of Intel Core i5-4690 CPU at 3.50GHz under GNU/Linux. The program is compiled with g++ version 4.8.4 at optimisation level 3 (-O3).



(a) 10% black positions

(b) 20% black positions

(c) 30% black positions

(d) 40% black positions

Figure 3: Elapsed time of Algorithm WeightedPrefixTable using synthetic DNA data ($\sigma = 4$) for black positions percentage 10% to 40% and text length 2MB to 32MB, for $z = 65,536$.

To evaluate the time performance of our implementation, synthetic DNA data ($\sigma = 4$) were used in the experiments. As input datasets we used four sets of weighted strings with different black positions percentages, ranging from 10% to 40%. For each set of strings, we used five different lengths for n: 2, 4, 8, 16, and 32 Megabytes (MB). The weighted strings were generated using a randomised (uniform distribution) script. We used a constant threshold $z = 65,536$ in all the experiments. The results, for each black positions percentage, are plotted in Fig 3. When the cumulative weight threshold $1/z$ is constant, for the weighted strings with the same black position percentage, it is demonstrated by the results that the elapsed time grows linearly with respect to the length of weighted strings. The experimental results confirm our theoretical findings (see Theorem 1). Given the relatively high value for z, the fast execution of the program is explained by the randomised (uniform distribution) data generation. This generation ensures that it is unlikely to have too many black positions close to each other, which would then result in a high number of z-valid factors being generated.

6. Final Remarks

In this article, we presented a linear-time algorithm for computing the prefix table for weighted strings with a very low constant factor (Corollary 1). This implies an $O(nz^2 \log z)$-time algorithm for pattern matching on weighted

strings for arbitrary z, which is simple and matches the best-known time complexity for this problem [3]. Furthermore, we showed how this structure can be of use in other problems on weighted strings such as computing the border table of a weighted string and computing the suffix/prefix overlap of two weighted strings; as well as for computing the prefix table of a conservative indeterminate string.

References

[1] C. S. Iliopoulos, M. Mohamed, L. Mouchard, K. Perdikuri, W. F. Smyth, A. K. Tsakalidis, String regularities with don't cares, Nord. J. Comput. 10 (1) (2003) 40–51.

[2] Nomenclature Committee of the International Union of Biochemistry (NC-IUB). Nomenclature for incompletely specified bases in nucleic acid sequences. Recommendations 1984., European Journal of Biochemistry 150 (1) (1985) 1–5.

[3] A. Amir, E. Chencinski, C. Iliopoulos, T. Kopelowitz, H. Zhang, Property matching and weighted matching, in: M. Lewenstein, G. Valiente (Eds.), Combinatorial Pattern Matching, Vol. 4009 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 188–199.

[4] A. Amir, C. Iliopoulos, O. Kapah, E. Porat, Approximate matching in weighted sequences, in: M. Lewenstein, G. Valiente (Eds.), Combinatorial Pattern Matching, Vol. 4009 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 365–376.

[5] C. S. Iliopoulos, L. Mouchard, K. Perdikuri, A. K. Tsakalidis, Computing the repetitions in a biological weighted sequence, Journal of Automata, Languages and Combinatorics 10 (5/6) (2005) 687–696.

[6] M. Christodoulakis, C. S. Iliopoulos, L. Mouchard, K. Perdikuri, A. K. Tsakalidis, K. Tsichlas, Computation of repetitions and regularities of biologically weighted sequences, Journal of Computational Biology 13 (6) (2006) 1214–1231.

[7] C. Barton, S. P. Pissis, Optimal computation of all repetitions in a weighted string, in: C. S. Iliopoulos, A. Langiu (Eds.), International Conference on Algorithms for Big Data (ICABD2014), no. 1146 in CEUR-WS Proceedings, Aachen, 2014, pp. 9–15.

[8] C. Barton, C. S. Iliopoulos, S. P. Pissis, Optimal computation of all tandem repeats in a weighted sequence, Algorithms for Molecular Biology 9 (21).

[9] C. S. Iliopoulos, C. Makris, Y. Panagis, K. Perdikuri, E. Theodoridis, A. Tsakalidis, The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications, Fundam. Inf. 71 (2,3) (2006) 259–277.

[10] A. Amir, Z. Gotthilf, B. R. Shalom, Weighted LCS, Journal of Discrete Algorithms 8 (3) (2010) 273–281.

[11] M. Cygan, M. Kubica, J. Radoszewski, W. Rytter, T. Walen, Polynomial-time approximation algorithms for weighted LCS problem, in: R. Giancarlo, G. Manzini (Eds.), Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011. Proceedings, Vol. 6661 of Lecture Notes in Computer Science, Springer, 2011, pp. 455–466.

[12] W. Smyth, S. Wang, New perspectives on the prefix array, in: A. Amir, A. Turpin, A. Moffat (Eds.), String Processing and Information Retrieval, Vol. 5280 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 133–143.

[13] C. Barton, C. S. Iliopoulos, S. P. Pissis, W. F. Smyth, Fast and simple computations using prefix tables under Hamming and edit distance, in: K. Jan, M. Miller, D. Froncek (Eds.), Combinatorial Algorithms, Vol. 8986 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 49–61.

[14] C. Barton, S. P. Pissis, Linear-time computation of prefix table for weighted strings, in: D. Nowotka (Ed.), Combinatorics on Words - 10th International Conference, WORDS 2015, Kiel, Germany, September 14-18. Proceedings, Lecture Notes in Computer Science, Springer, 2015.

[15] W. Ng, K. Kakehi, Merging string sequences by longest common prefixes, IPSJ Digital Courier 4 (2008) 69–78.

[16] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM 53 (6) (2006) 918–936.

[17] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings, 2001, pp. 181–192.

[18] J. Fischer, V. Heun, Space-efficient preprocessing schemes for range minimum queries on static arrays, SIAM J. Comput. 40 (2) (2011) 465–492.

[19] C. Pizzi, E. Ukkonen, Fast profile matching algorithms — A survey, Theor. Comput. Sci. 395 (2-3) (2008) 137–157.

[20] G. H. Gonnet, S. A. Benner, Probabilistic ancestral sequences and multiple alignments., in: R. G. Karlsson, A. Lingas (Eds.), SWAT, Vol. 1097 of Lecture Notes in Computer Science, Springer, 1996, pp. 380–391.

[21] J. Koshi, R. Goldstein, Probabilistic reconstruction of ancestral protein sequences, Journal of Molecular Evolution 42 (2) (1996) 313–320.

[22] P. Bork, T. J. Gibson, Applying motif and profile searches, in: R. F. Doolittle (Ed.), Computer Methods for Macromolecular Sequence Analysis, Vol. 266 of Methods in Enzymology, Academic Press, 1996, pp. 162 – 184.

[23] T. Yan, D. Yoo, T. Z. Berardini, L. A. Mueller, D. C. Weems, S. Weng, J. M. Cherry, S. Y. Rhee, PatMatch: a program for finding patterns in peptide and nucleotide sequences, Nucleic Acids Research 33 (suppl 2) (2005) W262–W266.

[24] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, New York, NY, USA, 2007.

[25] D. E. Knuth, J. H. M. Jr., V. R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.

[26] D. Breslauer, An on-line string superprimitivity test, Inf. Process. Lett. 44 (6) (1992) 345–347.

[27] T. Flouri, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, S. J. Puglisi, W. Smyth, W. Tyczynski, Enhanced string covering, Theoretical Computer Science 506 (2013) 102–114.

[28] W. Bland, G. Kucherov, W. F. Smyth, Prefix table construction and conversion, in: T. Lecroq, L. Mouchard (Eds.), Combinatorial Algorithms - 24th International Workshop, IWOCA 2013, Rouen, France, July 10-12, 2013, Revised Selected Papers, Vol. 8288 of Lecture Notes in Computer Science, Springer, 2013, pp. 41–53.

[29] J. Zhang, K. Kobert, T. Flouri, A. Stamatakis, PEAR: a fast and accurate Illumina Paired-End reAd mergeR, Bioinformatics 30 (5) (2014) 614–620.

[30] N. Välimäki, S. Ladra, V. Mäkinen, Approximate all-pairs suffix/prefix overlaps, in: A. Amir, L. Parida (Eds.), Combinatorial Pattern Matching, Vol. 6129 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 76–87.

[31] G. Kucherov, D. Tsur, Improved filters for the approximate suffix-prefix overlap problem, in: E. Moura, M. Crochemore (Eds.), String Processing and Information Retrieval, Vol. 8799 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 139–148.

[32] J. Holub, W. Smyth, S. Wang, Fast pattern-matching on indeterminate strings, Journal of Discrete Algorithms 6 (1) (2008) 37–50, selected papers from AWOCA 2005 Sixteenth Australasian Workshop on Combinatorial Algorithms.