Property Suffix Array with Applications in Indexing Weighted Sequences

Panagiotis Charalampopoulos[1,2], Costas S. Iliopoulos[1], Chang Liu[3], and Solon P. Pissis[4]

[1]Department of Informatics, King's College London, London, UK
[2]Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warsaw, Poland
[3]Cancer Bioinformatics, School of Cancer and Pharmaceutical Science, King's College London, London, UK
[4]CWI, Amsterdam, The Netherlands

Abstract

The suffix array is one of the most prevalent data structures for string indexing; it stores the lexicographically sorted list of suffixes of a given string. Its practical advantage compared to the suffix tree is space efficiency. In *Property Indexing*, we are given a string x of length n and a property Π, i.e. a set of Π-*valid* intervals over x so that a pattern p occurs in x if and only if x has an occurrence of p that lies entirely within an interval of Π. A suffix-tree-like index over the valid prefixes of suffixes of x can be built in time and space $\mathcal{O}(n)$. We show here how to directly build a suffix-array-like index, the *Property Suffix Array* (PSA), in time and space $\mathcal{O}(n)$. We mainly draw our motivation from weighted (probabilistic) sequences: sequences of probability distributions over a given alphabet. Given a probability threshold $\frac{1}{z}$, we say that a string p of length m matches a weighted sequence X of length n at starting position i if the product of probabilities of the letters of p at positions $i,\ldots,i+m-1$ in X is at least $\frac{1}{z}$. Our algorithm for building the PSA can be directly applied to build an $\mathcal{O}(nz)$-sized suffix-array-like index over X in time and space $\mathcal{O}(nz)$. Finally, we present extensive experimental results showing that our new indexing data structure is well suited for real-world applications.

1 Introduction

Property matching, introduced in [6], comprises of matching a pattern to a text of which only certain intervals are valid. The on-line version of this problem is trivial and thus the indexing version has received much more attention. In the *Property Indexing* problem, we are given a text x of length n over an alphabet of size σ and a *property* Π; Π is a set of subintervals of $[0,n-1]$ with integer endpoints. The goal is to then preprocess the text so that given a pattern p we can return its occurrences in the Π-valid intervals of x, i.e. we want to report $x[i\,..\,j]$ if and only if it is equal to p and $[i,j]$ is a subinterval of some $[a,b]\in\Pi$. Note that if an interval $A\in\Pi$ is a subinterval of an interval $B\in\Pi$, then we can just discard A. Further note, that for an inclusion-free family of intervals Π we have $|\Pi|=\mathcal{O}(n)$.

Most of the prevalent text indexing data structures are built over the suffixes of the text [38]. However, by introducing the property Π only some prefixes of each suffix are now valid. The authors in [6] presented an algorithm for building the *Property Suffix Tree* (PST) in $\mathcal{O}(n\log\sigma+n\log\log n)$ time for integer alphabets, implicitly sorting the prefixes of the suffixes that are valid. The PST answers pattern matching queries in time proportional to the length of the pattern and reports all occurrences in time proportional to their number. Recently, the authors in [7] have presented an $\mathcal{O}(n)$-time algorithm for the construction of the PST that also works for integer alphabets. The algorithm for integer alphabets is partially based on a technique by Kociumaka et al. for answering off-line weighted ancestor queries on trees [30]. A dynamic instance of Property Indexing has also been studied in [33], where the author also makes use of the suffix tree. A (dynamic) compressed suffix-tree-based data structure for Property Indexing has been proposed in [22].

An $\mathcal{O}(n)$-time algorithm for building an index over the suffix tree of x for integer alphabets that allows for property matching queries was proposed by the authors of [23, 24]. This solution, however, does not sort the prefixes of suffixes that are valid (which is an interesting problem per se); it offloads the difficulty of the computation from the construction to the queries.

The *suffix array* (SA) of a text x of length n is an integer array of size n that stores the lexicographically sorted list of suffixes of x [34]. In order to construct the *Property Suffix Array*, which we denote by PSA, we essentially need to lexicographically sort a multiset consisting of substrings of x; this multiset contains at most one prefix of each suffix of x. This can be achieved in linear time by traversing the PST, however our aim here is to do it directly—we do not want to construct or store the PST. It is well-known from the setting of standard strings that the SA is more space efficient than the suffix tree [2].

Note that for clarity of presentation we represent Π—and assume the input is given in this form—by an integer array $\mathcal{L}$ of size n, such that

$$\mathcal{L}[i] = \max\{j | (k, j) \in \Pi, k \leq i\} - i + 1$$

is the length of the longest prefix of $x[i\,..\,n-1]$ that is valid. It should be clear that $\mathcal{L}$ can be obtained from Π in $\mathcal{O}(n + |\Pi|)$ time. We also assume that $\mathcal{L}[i] > 0$ for all i; the case that $\mathcal{L}[i] = 0$ can be handled easily as the resulting substring would just be the empty string.

Example 1 (Running example)**.** *Consider the string* $x =$ `acababaab` *and property* $\Pi = \{(0, 3), (4, 6), (6, 8)\}$*:*

i	0	1	2	3	4	5	6	7	8
$x[i]$	a	c	a	b	a	b	a	a	b
$\mathcal{L}[i]$	4	3	2	1	3	2	3	2	1
$\mathsf{SA}[i]$	6	7	4	2	0	8	5	3	1
$\mathsf{PSA}[i]$	6	2	7	4	0	3	8	5	1

Our main result is an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space direct construction of the PSA for integer alphabets. The problem can be formally defined as follows.

Property Suffix Array
Input: A string x of length n and an integer array $\mathcal{L}$ of size n, satisfying $0 < \mathcal{L}[i] \leq n - i$ and $\mathcal{L}[i] \geq \mathcal{L}[i-1] - 1$.
Output: An array PSA that stores a permutation of $0, \ldots, n-1$ and for all $1 \leq r < n$, letting $\mathsf{PSA}[r-1] = j$ and $\mathsf{PSA}[r] = k$, we have $x[j\,..\,j + \mathcal{L}[j] - 1] \leq x[k\,..\,k + \mathcal{L}[k] - 1]$.

Application. We apply our solution to this problem in the setting of weighted sequences. In a weighted sequence every position contains a subset of the alphabet and every letter of this subset is associated with a probability of occurrence such that the sum of probabilities at each position equals 1. This data representation is common in a wide range of applications: (i) imprecise sensor data measurements; (ii) flexible sequence modeling, such as binding profiles of DNA sequences; (iii) observations that are private and thus sequences of observations may have artificial uncertainty introduced deliberately (see [3] for a survey). Pattern matching (or substring matching) is a core operation in a wide variety of applications including bioinformatics, information retrieval, text mining, and pattern recognition. Many pattern matching applications generalize naturally to the weighted case as much of this data is more commonly uncertain (e.g. genomes with incorporated SNPs from a population) than certain.

In the *weighted pattern matching* (WPM) problem we are given a string p of length m called a pattern, a weighted sequence X 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 i in X where the product of probabilities of the letters of p at positions $i, \ldots, i+m-1$ in X is at least $\frac{1}{z}$ [15, 31, 9, 10, 37]. Each such position is called an *occurrence* of the pattern; we also say that the fragment and the pattern *match*.

Here we consider the problem of indexing a weighted sequence. We are given a weighted sequence X of length n and a probability threshold $\frac{1}{z}$, and we are asked to construct an index which will allow us to

efficiently answer queries with respect to the contents of X. This problem was considered in [6], where a reduction to Property Indexing of a text of size $\mathcal{O}(nz^2 \log z)$ was proposed. The authors in [7] reduced this to a text of size nz, thus presenting an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space construction of an $\mathcal{O}(nz)$-sized index that answers weighted pattern matching queries on X in optimal time. The same index as the one in [7] was first presented in [8] but with a different $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space construction algorithm. Approximate variants of these indexes have also been considered in [13, 7] with the aim of reducing the construction time. In these approximate variants, we may report additional occurrences that have probability $\frac{1}{z} - \epsilon$, for a preselected value of $\epsilon > 0$.

All these indexes [6, 7, 8] are based on constructing and traversing the suffix tree. Here, using our solution to problem Property Suffix Array *and* the main idea of [7], we show how to construct directly an array data structure for weighted pattern matching within the same complexities. Moreover, we present experiments that show the advantage of our new data structure: as expected, it requires much less space in practice than the one of [7, 8]. Our index, apart from being *simple* and *small* in practice, is *asymptotically smaller* than the input weighted sequence when $z = o(\sigma)$.

Structure of the paper. In Section 3, we provide three $\mathcal{O}(n)$-space algorithms for computing the PSA directly, with time complexities $\mathcal{O}(n \log^2 n)$, $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$; we provide yet another $\mathcal{O}(n)$-space algorithm with $\mathcal{O}(n)$ average-case time complexity. In Section 4, we apply our solution to this general problem in the setting of weighted sequences to obtain an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm for constructing a new $\mathcal{O}(nz)$-sized array index for weighted sequences. Finally, in Section 5, we present an extensive experimental evaluation showing that our new indexing data structure is indeed better suited for real-world applications than the property suffix tree.

A preliminary version of this paper appeared in [14].

2 Preliminaries

Let $x = x[0]x[1] \dots x[n-1]$ be a *string* of length $|x| = n$ over a finite ordered *alphabet* Σ of size σ, i.e. $\sigma = |\Sigma|$. In particular, we consider the case of an *integer alphabet*; in this case each letter is replaced by its rank such that the resulting string consists of integers in the range $\{1, \dots, n\}$.

For two positions i and j on x, we denote by $x[i\,..\,j] = x[i] \dots x[j]$ the *factor* (sometimes called *substring*) of x that starts at position i and ends at position j. We recall that a *prefix* of x is a factor that starts at position 0 ($x[0\,..\,j]$) and a *suffix* of x is a factor that ends at position $n-1$ ($x[i\,..\,n-1]$). We denote a string x that is lexicographically smaller than (resp. smaller than or equal to) a string y by $x < y$ ($x \leq y$).

2.1 Suffix array

We denote by SA the *suffix array* of a non-empty string x of length n. SA is an integer array of size n storing the starting positions of all (lexicographically) sorted non-empty suffixes of x, i.e. for all $1 \leq r < n$ we have $x[\mathsf{SA}[r-1]\,..\,n-1] < x[\mathsf{SA}[r]\,..\,n-1]$ [34]. Let $\mathsf{lcp}(r, s)$ denote the length of the longest common prefix between $x[\mathsf{SA}[r]\,..\,n-1]$ and $x[\mathsf{SA}[s]\,..\,n-1]$ for all positions r, s on x, and 0 otherwise. We denote by LCP the *longest common prefix* array of y defined by $\mathsf{LCP}[r] = \mathsf{lcp}(r-1, r)$ for all $1 \leq r < n$, and $\mathsf{LCP}[0] = 0$. The inverse iSA of the array SA is defined by $\mathsf{iSA}[\mathsf{SA}[r]] = r$, for all $0 \leq r < n$. It is known that SA [27, 36, 35, 25], iSA, and ,LCP [29, 26] of a string of length n, over an integer alphabet, can be computed in time and space $\mathcal{O}(n)$. It is then known that a range minimum query (RMQ) data structure over the LCP array, that can be constructed in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space [12, 20, 19, 11], can answer lcp queries in $\mathcal{O}(1)$ time per query by returning the index of a minimal value in the respective range of the SA.

3 $\mathcal{O}(n)$-space algorithms for computing PSA

3.1 Sparse Table-based $\mathcal{O}(n \log^2 n)$-time algorithm

The algorithm presented in this subsection applies a combination of the *Sparse Table* idea for answering RMQs [12] and the *doubling technique* [34] to the context of sorting prefixes of suffixes (factors) of x. Using this combination, one may easily obtain an $\mathcal{O}(n \log n)$-time and $\mathcal{O}(n \log n)$-space algorithm for constructing the PSA [18]. Specifically, Crochemore et al. show that by precomputing the *dictionary of basic factors* of x (basic factors are factors whose lengths are powers of 2), any two factors of x can be compared in constant time. We tweak this solution to require only $\mathcal{O}(n)$ space, suffering an additional multiplicative $\log n$ factor in the time complexity. There are $\mathcal{O}(\log n)$ levels: at the kth level, we sort prefixes of length up to 2^{k+1} of suffixes; at each level, $\mathcal{O}(n \log n)$ time is required to sort these factors using any optimal comparison-based sorting algorithm [17].

The aforementioned scheme assumes that we can compare two factors in constant time. To this end, we borrow the Sparse Table algorithm idea for answering RMQs: the minimum value in a given range r is the minimum between the minimums of any two, potentially overlapping, subranges whose union is r. The same idea can be applied in our context:

Fact 2. *Given two strings x and y, with $|x| \leq |y|$, and $k = \lfloor \log |x| \rfloor$, $x \leq y$ if and only if $(x[0\,..\,2^k], x[|x| - 2^k\,..\,|x| - 1]) \leq (y[0\,..\,2^k], y[|x| - 2^k\,..\,|x| - 1])$.*

We thus compute the ranks of prefixes of suffixes whose lengths are multiples of two using the doubling technique [34] and then use these ranks to sort prefixes whose lengths may not be multiples of two by applying Fact 2. Note that this computation can be done level by level in a total of $\mathcal{O}(\log n)$ levels, and therefore the working space is $\mathcal{O}(n)$. We formalize this algorithm, denoted by ST-PSA, in the pseudocode below. We start by initializing the elements in the PSA by sorting and ranking the letters of x (Lines 2–8). We store these ranks in an array (Line 9). Then, at level k (Line 10), we compute the ranks of prefixes whose lengths are multiples of two using the previous level information and radix sort in $\mathcal{O}(n)$ time (Lines 11–12). Next, we sort and rank *all* prefixes up to length 2^{k+1} using a comparison-based sorting algorithm and Fact 2 in $\mathcal{O}(n \log n)$ time (Lines 13–14). We store these ranks in an array (Line 15) and proceed to the next level. Thus the total time required is $\mathcal{O}(n \log^2 n)$ and the space is $\mathcal{O}(n)$. The value of this algorithm is its practicality: (a) it requires very little space; (b) the number of levels required is in fact $\lfloor \log \ell \rfloor$, where ℓ is the maximum value in $\mathcal{L}$; and (c) at level k it suffices to sort groups of elements having the same rank at level $k - 1$.

```
1  Algorithm ST-PSA(x, n, L)
2      for i ← 0 to n − 1 do
3          PSA[i] ← i;
4      Sort PSA using the following comparison rule for PSA[i] and PSA[j]:
5          if x[i] < x[j] then PSA[i] < PSA[j];
6          else if x[i] > x[j] then PSA[i] > PSA[j];
7          else PSA[i] = PSA[j];
8      Rank the elements of PSA and store their ranks in Rank_PSA;
9      Rank_PREF ← Rank_PSA;
10     for k ← 1 to ⌊log n⌋ do
11         Construct an array A of pairs: A[i] = (Rank_PREF[i], Rank_PREF[i + 2^(k−1)]);
12         Sort the pairs in A using radix sort and store their ranks in Rank_CURR;
13         Sort PSA using L, Rank_PSA, Rank_CURR, and Fact 2 for the comparison;
14         Rank the elements of PSA and store their new ranks in Rank_PSA;
15         Rank_PREF ← Rank_CURR;
16     return PSA;
```

For a parallel implementation of this algorithm, we make use of Cole's merge sort algorithm for parallel sorting [16]. For an input of size n and P processors, Cole's algorithm takes time $\mathcal{O}((n \log n)/P + \log P)$. We plug this parallel merge sort in the sorting part of algorithm ST-PSA. Specifically, we have $\mathcal{O}(\log n)$ levels and each level can be implemented using parallel merge sort. We get an overall execution time of $\mathcal{O}((n \log^2 n)/P + \log P \log n)$. Thus the Property Suffix Array problem can be solved in $\mathcal{O}(\log^2 n)$ time with $\mathcal{O}(n \log^2 n)$ work in the EREW PRAM model.

3.2 Average-case $\mathcal{O}(n)$-time algorithm

It should be clear that algorithm ST-PSA attains the time bound of $\mathcal{O}(n \log^2 n)$ for string $x[i] = a$ and $\mathcal{L}[i] = \min\{i+1, n-i\}$, for all $0 \leq i < n$. Let us make a more careful analysis of this algorithm in the average-case setting.

Our analysis model. We assume that the input is a string x of length n over an alphabet Σ of size $\sigma > 1$ with the letters of x being independent and identically distributed uniform random variables over Σ.

Under this model, the expected length of the longest repeated substring of x is known to be $2\log_\sigma n + \mathcal{O}(1)$ [28]. Hence, similar to the average-case analysis for computing the standard suffix array [34], the average-case time complexity of algorithm ST-PSA is $\mathcal{O}(n \log n \cdot \log\log_\sigma n)$ since we have $\mathcal{O}(\log\log_\sigma n)$ levels.

Note that we cannot directly apply the rest of the tricks presented in [34] to shave the $\log\log_\sigma n$ factor. Intuitively, the reason for this extra hardness, compared to the standard setting, is the fact that we need to account for property Π: not all values in $\mathcal{L}$ are at least $2\log_\sigma n + \mathcal{O}(1)$. Notably, we manage here to shave not only the $\log\log_\sigma n$ factor but the $\log n$ factor as well. Let us denote this new algorithm by AC-PSA.

The main idea comes from [34]. Let $t = \lfloor \log_\sigma n \rfloor$ and consider mapping each string of length t over Σ to a unique integer obtained when the string is viewed as a t-digit number. This is an isomorphism onto the range $[0, \sigma^t - 1] \subseteq [0, n-1]$. We define $s(x[i\,..\,i+t-1])$ as the integer signature of $x[i\,..\,i+t-1]$. We have that

$$s(x[i\,..\,i+t-1]) = s(x[i-1\,..\,i+t-2]) \cdot \sigma - x[i-1]\sigma^t + x[i+t-1].$$

It should thus be clear that in time $\mathcal{O}(n)$ we can compute $s(x[i\,..\,i+t-1])$ for all i. In our setting, however, we need to account for property Π. We thus go on to generalize this technique as follows.

Let $\Sigma' = \Sigma \sqcup \{\$\}$, where $\$$ is a letter (lexicographically) smaller than all letters in Σ. Further let $t' = \lfloor \log_{\sigma+1} n \rfloor$ and consider an analogous mapping onto the range $[0, (\sigma+1)^{t'} - 1] \subseteq [0, n-1]$. We define the integer signature s' of $x[i\,..\,i+t'-1]]$ as follows:

$$s'(x[i\,..\,i+t'-1]]) = \begin{cases} x[i] \cdot (\sigma+1)^{t'-1} + x[i+1] \cdot (\sigma+1)^{t'-2} + \ldots + x[i+t'-1] & : \mathcal{L}[i] \geq t' \\ x[i] \cdot (\sigma+1)^{t'-1} + \ldots + x[i+\mathcal{L}[i]-1](\sigma+1)^{t'-\mathcal{L}[i]} + & \\ \$ \cdot (\sigma+1)^{t'-\mathcal{L}[i]-1} + \ldots + \$ \cdot (\sigma+1) & : \mathcal{L}[i] < t'. \end{cases} \quad (1)$$

The initialization step consists of computing $s'(x[0\,..\,t'-1])$ trivially in time $\mathcal{O}(t')$.

We first consider the case $\mathcal{L}[i] \geq t'$. We make a first pass, from left to right, and compute $s'(x[i\,..\,i+t'-1])$, by ignoring the properties, as follows.

$$s'(x[i\,..\,i+t'-1]) = (s(x[i-1\,..\,i+t'-2]) - x[i-1] \cdot (\sigma+1)^{t'-1}) \cdot (\sigma+1) + x[i+t'-1].$$

At this point all signatures are computed correctly for all i such that $\mathcal{L}[i] \geq t'$. It should be clear that this can be implemented in time $\mathcal{O}(n)$.

We then consider the case $\mathcal{L}[i] < t'$. We make a second pass, from left to right. Assuming that $\$$ is mapped to 0, we mask the $t' - \mathcal{L}[i]$ letters that are not there, due to the property, with 0's; we use the previously

computed signatures and standard word-level operations to achieve that. Let $r = \lceil \log(\sigma + 1) \rceil$. We have that:

$$s'(x[i\,..\,i+t'-1]) := \\ s'(x[i\,..\,i+t'-1])\ \texttt{AND}\ ((2^{r\cdot t'}-1)-(2^{r\cdot(t'-\mathcal{L}[i])}-1)) = \\ s'(x[i\,..\,i+t'-1])\ \texttt{AND}\ (2^{r\cdot t'}-2^{r\cdot(t'-\mathcal{L}[i])}). \quad (2)$$

This computation can be executed in $\mathcal{O}(1)$ time since $t' = \lfloor \log_{\sigma+1} n \rfloor$. The whole procedure thus takes time $\mathcal{O}(n)$. This completes the computation of $s'(x[i\,..\,i+t-1])$ for all i.

At this point, we apply the doubling technique of [34] on string $x \cdot \$ \ldots \$$ of length $2n$; namely, we append n \$'s to x. Instead of performing a radix sort on the first letter of each suffix, we perform it on the t'-length prefixes of suffixes using their signatures. This radix sort requires time $\mathcal{O}(n)$ because $t' = \lfloor \log_{\sigma+1} n \rfloor$. For the second stage, instead of performing a radix sort on the signatures of all $2t'$-length prefixes of suffixes, each suffix is represented by a pair. The first element of this pair is the rank of the t'-length prefix; the second element is the signature of the succeeding substring of length t'. The former (rank) has been computed at the previous stage. For the latter (signature), note that, due to the properties, we cannot guarantee that these signatures have been computed at the previous stage. All signatures, however, can be computed in $\mathcal{O}(n)$ time for all suffixes using the aforementioned method. Sorting these pairs can be done in $\mathcal{O}(n)$ time. Since the expected length of the longest repeated substring of x is $t(2 + \mathcal{O}(1))$ and $t' = \Theta(t)$, a constant number of stages of the doubling technique are expected to be required to complete the sort. Thus algorithm AC-PSA solves the Property Suffix Array problem in $\mathcal{O}(n)$ time on average using $\mathcal{O}(n)$ working space.

3.3 LCP-based $\mathcal{O}(n \log n)$-time algorithm

The algorithm presented in this subsection is based on the following fact.

Fact 3. *Given two factors of x, $x[i_1\,..\,j_1]$ and $x[i_2\,..\,j_2]$, with $\mathsf{iSA}[i_1] < \mathsf{iSA}[i_2]$, we have that $x[i_2\,..\,j_2] \leq x[i_1\,..\,j_1]$ if and only if $j_2 - i_2 \leq \mathsf{lcp}(\mathsf{iSA}[i_1], \mathsf{iSA}[i_2])$ and $j_2 - i_2 \leq j_1 - i_1$.*

Recall that lcp queries for two arbitrary suffixes of x can be answered in time $\mathcal{O}(1)$ per query after an $\mathcal{O}(n)$-time preprocessing of the LCP array of x [34, 12]. We can then perform any optimal comparison-based sorting algorithm (using Fact 3 for the comparison) on the set of prefixes of suffixes. Thus the total time required is $\mathcal{O}(n \log n)$ and the working space is $\mathcal{O}(n)$. We formalize this algorithm, denoted by LCP-PSA, in the pseudocode below.

```
1  Algorithm LCP-PSA(x, n, L)
2      Compute SA, iSA, LCP, RMQ_LCP of x;
3      for i ← 0 to n − 1 do
4          PSA[i] ← SA[i];
5      Sort PSA using the following comparison rule for PSA[i] and PSA[j]:
6          if i < j then k ← RMQ_LCP(i + 1, j);
7          else k ← RMQ_LCP(j + 1, i);
8          if LCP[k] < min{L[SA[i]], L[SA[j]]} then
9              if i < j then PSA[i] < PSA[j];
10             else PSA[i] > PSA[j];
11         else
12             if L[SA[i]] < L[SA[j]] then  PSA[i] < PSA[j];
13             else  PSA[i] > PSA[j];
14     return PSA;
```

3.4 Union-Find-based $\mathcal{O}(n)$-time algorithm

In this section we assume the precomputation of SA, iSA and LCP of x. Given the iSA, the LCP array and $\mathcal{L}$, let $f_i = \max_{0 \le r \le \mathsf{iSA}[i]} \{r | \mathsf{LCP}[r] < \mathcal{L}[i]\}$. Informally, f_i tells us how many suffixes are lexicographically (strictly) smaller than $x[i \,..\, i + \mathcal{L}[i] - 1]$ (see also Example 5 in this regard). It follows from the following lemma that in order to construct the PSA it is enough to sort the ordered pairs $(f_i, \mathcal{L}[i])$.

Lemma 4. *Given two factors of x, $x[i_1 \,..\, j_1]$ and $x[i_2 \,..\, j_2]$, we have that $(f_{i_1}, j_1 - i_1) \le (f_{i_2}, j_2 - i_2)$ if and only if $x[i_1 \,..\, j_1] \le x[i_2 \,..\, j_2]$.*

Proof. ($\Rightarrow$): Note that, for all i, j, $x[i \,..\, j]$ is a prefix of $x[\mathsf{SA}[f_i] \,..\, n-1]$. Thus if either (a) $f_{i_1} < f_{i_2}$ or (b) $f_{i_1} = f_{i_2}$ and $j_1 - i_1 \le j_2 - i_2$ then we have that $x[i_1 \,..\, j_1] \le x[i_2 \,..\, j_2]$.

($\Leftarrow$): Conversely, if $f_{i_1} < f_{i_2}$ then $x[i_1 \,..\, j_1] < x[i_2 \,..\, j_2]$. Else, if $f_{i_1} < f_{i_2}$ and $j_1 - i_1 \le j_2 - i_2$ then $x[i_1 \,..\, j_1]$ is a prefix of $x[i_1 \,..\, j_2]$. □

Example 5 (Running example)**.** *For $i = 3$, we have that $\mathsf{iSA}[3] = 7$, and hence we obtain the pair $(f_3, \mathcal{L}[3]) = (5, 1)$:*

i	0	1	2	3	4	5	6	7	8
$\mathcal{L}[i]$	4	3	2	1	3	2	3	2	1
$\mathsf{SA}[i]$	6	7	4	2	0	8	5	3	1
$\mathsf{LCP}[i]$	0	1	2	3	1	0	1	2	0
$\mathcal{L}[\mathsf{SA}[i]]$	3	2	3	2	4	1	2	1	3
$f_{\mathsf{SA}[i]}$	0	1	2	1	4	5	6	5	8
$\mathsf{PSA}[i]$	6	2	7	4	0	3	8	5	1

The computational problem is to compute f_i efficiently for all i; for this we rely on the Union-Find data structure [17] in a similar manner as the authors in [32]. Our technique also resembles the technique by Kociumaka et al. for answering off-line weighted ancestor queries on trees [30] (see also [4]). Union-Find maintains a partition of $\{0, 1, \ldots, n-1\}$, where each set has a representative element, and supports three basic operations:

- MakeSet(n) creates n new sets $\{0\}, \{1\}, \ldots, \{n-1\}$, where the representative index of set $\{i\}$ is i.
- Find(i) returns the representative of the set containing i.
- Union(i, j) first finds the set S_i containing i and the set S_j containing j. If $S_i \neq S_j$, then they are replaced by the set $S_i \cup S_j$.

In the algorithm described below, we only encounter *linear* Union-Find instances, in which the sets of the partition consist of consecutive integers and the representative of each set is its smallest element. We rely on the following result.

Theorem 6 ([21])**.** *A sequence of q given linear Union and Find operations over a partition of $\{0, 1, \ldots, n-1\}$ can be performed in time $\mathcal{O}(n + q)$.*

We perform the following initialization steps in $\mathcal{O}(n)$ time:

1. Initialize an array $\mathcal{A}$ of linked lists of size n;
2. Initialize the Union-Find data structure by calling MakeSet(n);
3. Sort indices $\{0, 1, \ldots, n-1\}$ based on $\mathcal{L}[i]$ (store them in an array $\mathcal{M}_{\mathcal{L}}$);
4. Sort indices $\{0, 1, \ldots, n-1\}$ based on $\mathsf{LCP}[i]$ (store them in an array $\mathcal{M}_{\mathsf{LCP}}$).

Then, for all j from $k = \max\{\max_i\{\mathsf{LCP}[i]\}, \max_i\{\mathcal{L}[i]\}\}$ down to 1 we do the following:

1. $\mathsf{Union}(i-1, i)$ for each i such that $\mathsf{LCP}[i] = j$ using $\mathcal{M}_{\mathsf{LCP}}$;
2. We find all i for which $\mathcal{L}[i] = j$ using $\mathcal{M}_{\mathcal{L}}$ and conclude that $f_i = \mathsf{Find}(\mathsf{iSA}[i])$; we store i at the head of the linked list $\mathcal{A}[f_i]$.

Note that after performing the Union operations for some j, the representative element of the set containing α, $\mathsf{Find}(\alpha)$, is the greatest $\beta \leq \alpha$, for which $\mathsf{LCP}[\beta] \leq j-1$. Thus, in the end of the computation, $\mathcal{A}[j]$ stores the indices i, for which $f_i = j$. In addition, the elements of each list $\mathcal{A}[j]$ are in the order of non-decreasing $\mathcal{L}[i]$. We can thus just read the elements of the linked lists in $\mathcal{A}$ from the left to the right and from the head to the tail to obtain the PSA. We formalize this algorithm, denoted by UF-PSA, in the pseudocode below.

```
1  Algorithm UF-PSA(x, n, L)
2      Compute SA, iSA and LCP of x;
3      Construct a map M_LCP such that M_LCP[i] = {j | LCP[j] = i};
4      Construct a map M_L such that M_L[i] = {j | L[j] = i};
5      Initialize an array of lists A of size n;
6      Initialize a Union-Find data structure UF;
7      UF.MakeSet(n);
8      lcp_max ← max{LCP[0], LCP[1], ..., LCP[n-1]};
9      ℓ_max ← max{L[0], L[1], ..., L[n-1]};
10     for j ← k = max{lcp_max, ℓ_max} to 1 do
11         foreach i ∈ M_LCP[j] do
12             UF.Union(i-1, i);
13         foreach i ∈ M_L[j] do
14             f ← UF.Find(iSA[i]);
15             Insert i at the head of A[f];
16     for j ← 0 to n-1 do
17         foreach i ∈ A[j] do
18             Insert(i, PSA);
19     return PSA;
```

Example 7 (Running example)**.** *The following two tables show the partition of* $\{0, 1, \ldots, n-1\}$ *before (top) and after (bottom) the* Union *operations performed for* $j = 1$*. Each monochromatic block represents a set in the partition.*

i	0	1	2	3	4	5	6	7	8
$\mathsf{LCP}[i]$	0	1	2	3	1	0	1	2	0

i	0	1	2	3	4	5	6	7	8
$\mathsf{LCP}[i]$	0	1	2	3	1	0	1	2	0
$\mathcal{L}[i]$	4	3	2	1	3	2	3	2	1

Find operations are then performed for those i *for which* $\mathcal{L}[i] = 1$*. For example for* $i = 3$ *we have that* $\mathsf{Find}(\mathsf{iSA}[3]) = \mathsf{Find}(7) = 5$*, since* 5 *is the smallest element in the set where* 7 *belongs. Hence* 3 *is added in the head of the linked list* $\mathcal{A}[5]$*.*

Putting together Lemma 4, Theorem 6 and the above description we obtain the following.

Theorem 8. *Problem* Property Suffix Array *can be solved in time and space* $\mathcal{O}(n)$*.*

In the standard setting, the SA is usually coupled with the LCP array to allow for efficient on-line pattern searches (see [34] for the details).

Definition 9. *The* property Longest Common Prefix array *(pLCP) for x and $\mathcal{L}$ is an integer array of size n such that, for all $1 \leq r < n$, $\mathsf{pLCP}[r]$ is the length of the longest common prefix of $x[i\,..\,i+\mathcal{L}[i]-1]$ and $x[j\,..\,j+\mathcal{L}[j]-1]$, where $i = \mathsf{PSA}[r]$ and $j = \mathsf{PSA}[r-1]$.*

Lemma 10. *We can compute the pLCP array in time $\mathcal{O}(n)$.*

Proof. We compute the pLCP array while constructing the PSA as follows. If we read both $\mathsf{PSA}[r-1]$ and $\mathsf{PSA}[r]$ from $\mathcal{A}[j]$, we set $\mathsf{pLCP}[r] = \mathcal{L}[\mathsf{PSA}[r-1]]$ since $x[i\,..\,i+\mathcal{L}[i]-1]]$ is a prefix of $x[i'\,..\,i'+\mathcal{L}[i']-1]]$. Otherwise, we read $\mathsf{PSA}[r-1]$ from $\mathcal{A}[j]$ and $\mathsf{PSA}[r] = i'$ from $\mathcal{A}[j']$ and proceed as follows:

1. If $\mathsf{iSA}[i'] < \mathsf{iSA}[i]$ then $x[i\,..\,i+\mathcal{L}[i]-1]$ is a prefix of $x[i'\,..\,i'+\mathcal{L}[i']-1]$ and hence we set $\mathsf{pLCP}[r] = \mathcal{L}[i]$;
2. else $\mathsf{iSA}[i] < \mathsf{iSA}[i']$, and since $\mathcal{L}[i] \leq \mathsf{lcp}(j, \mathsf{iSA}[i])$ and $\mathcal{L}[i'] \leq \mathsf{lcp}(j', \mathsf{iSA}[i'])$ we set

$$\mathsf{pLCP}[r] = \min\{\mathsf{lcp}(j, j'), \mathcal{L}[i], \mathcal{L}[i']\}.$$

We can compute $\mathsf{lcp}(j, j')$ for all consecutive non-empty lists $\mathcal{A}[j]$, $\mathcal{A}[j']$ in a simple scan of the LCP array in time $\mathcal{O}(n)$. □

Remark 11. *Alternatively, we can compute the pLCP array using lcp queries, since $\mathsf{pLCP}[r] = \min\{\mathsf{lcp}(\mathsf{PSA}[r-1], \mathsf{PSA}[r]), \mathcal{L}[\mathsf{PSA}[r-1]], \mathcal{L}[\mathsf{PSA}[r]]\}$.*

Finally, it is worth noting that the algorithms presented in this section for constructing the PSA depend neither on the fact that $\mathcal{L}[i] \geq \mathcal{L}[i-1]-1$ nor on the fact that we have (at most) one substring starting at each position. As a byproduct we thus obtain the following result *without* the aid of suffix tree, which is interesting in its own right.

Theorem 12. *Given q substrings of a string x of length n, encoded as intervals over x, we can sort them lexicographically in time and space $\mathcal{O}(n+q)$.*

4 Weighted Suffix Array

A *weighted sequence* X of length $|X| = n$ over an alphabet Σ is an $n \times \sigma$ matrix that specifies, for each position $i \in \{0, \ldots, n-1\}$ and letter $c \in \Sigma$, a probability $\pi_i^{(X)}(c)$ of c occurring at position i. If the considered weighted sequence is unambiguous, we write π_i instead of $\pi_i^{(X)}$. These values are non-negative and sum up to 1 for any given i.

The *probability of matching* of a string p with a weighted sequence X ($|p| = |X|$) equals

$$\mathcal{P}(p, X) = \prod_{i=0}^{|p|-1} \pi_i^{(X)}(p[i]).$$

We say that a string p *matches* a weighted sequence X with probability at least $\frac{1}{z}$ 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 $X[i] \ldots X[j]$. We then say that a string p *occurs* in X at position i if p matches the factor $X[i\,..\,i+|p|-1]$.

A weighted sequence is called *special* if, at each position, it contains at most one letter with non-zero probability. In this special case the assumption that the probabilities sum up to 1 for a given position is waived.

In this section, we present an algorithm for constructing a new index for a weighted sequence X of length n and a probability threshold $\frac{1}{z}$. We combine the ideas presented above with the following powerful combinatorial result (Theorem 13) presented in [7]. Informally, Theorem 13 tells us that one can construct in $\mathcal{O}(nz)$ time a family of $\lfloor z \rfloor$ special weighted sequences, each of length n, that carry all the information about all the strings occurring in X. More specifically, a string occurs with probability $\beta \geq \frac{1}{z}$ at position i in one of these $\lfloor z \rfloor$ special weighted sequences if and only if it occurs at position i of X with probability β.

The authors of [7] used this result to design an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm for constructing the *Weighted Index*: an $\mathcal{O}(nz)$-sized suffix-tree-like index for X. The Weighted Index is essentially the PST built over this family of strings after some appropriate property shifting.

Theorem 13 ([7]). *For a weighted sequence X of length n over an integer alphabet of size σ and a threshold $\frac{1}{z}$, one can construct in $\mathcal{O}(n\sigma + nz)$ time an equivalent collection of $\lfloor z \rfloor$ special weighted sequences.*

Definition 14. *The* Weighted Suffix Array *(WSA) for X and $\frac{1}{z}$ is an integer array (of size at most $n\lfloor z \rfloor$) storing the path-labels of the terminal nodes of the Weighted Index for X and $\frac{1}{z}$ in the order in which they are visited in a (lexicographic) depth first traversal.*

We create a new special weighted sequence Y by concatenating these $\lfloor z \rfloor$ special weighted sequences. Let us view Y as a standard string y of length $n\lfloor z \rfloor$ for simplicity (at most one letter per position has a non-zero probability). The probabilities at each position of Y and the ends of the original $\lfloor z \rfloor$ special weighted sequences give array $\mathcal{L}$ for y through a sliding window approach. For each of these $\lfloor z \rfloor$ strings, we start with a length-0 window at its left end. We extend the window to the right while the product of the probabilities is at least $\frac{1}{z}$ and we do not cross the right end of the string. When we cannot extend the window, we increment its left end and try to extend it further. We then construct the PSA for y and $\mathcal{L}$ using Theorem 8.

We also remove duplicates as follows. Let us assume that a string of length m occurring at a position i of X occurs at several positions $j_0, j_1, \ldots, j_{k-1}$ in y, with $j_p = i \pmod n$ and $\mathcal{L}[j_p] = m$ for all $0 \le p < k$. We naturally want to keep *one* of these occurrences.[1] We do that as follows: we identify maximal intervals $[r, s]$ in the PSA satisfying $\mathcal{L}[\mathsf{PSA}[q]] = \mathsf{pLCP}[t]$ for all $r - 1 \le q \le s$ and $r \le t \le s$; for each such interval, we consider all indices in $\{\mathsf{PSA}[q] | r - 1 \le q \le s\}$ modulo n, we bucket sort the residuals, and keep one representative for each of them. Doing this for the PSA of y and $\mathcal{L}$ from left to right, we end up with an array of size *at most* $n\lfloor z \rfloor$ that is the WSA for X and $\frac{1}{z}$.

Theorem 15. *The WSA for a weighted sequence X of length n over an integer alphabet of size σ and a threshold $\frac{1}{z}$ can be constructed in $\mathcal{O}(n\sigma + nz)$ time.*

The WSA for X and $\frac{1}{z}$, coupled with the naturally defined *weighted Longest Common Prefix array* (wLCP), which can be inferred directly from the pLCP array of y and $\mathcal{L}$, is an index with comparable capabilities as the ones of the SA coupled with the LCP array in the standard setting [34].

Example 16. *Let $X = [(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)]\,\mathtt{b}\,\mathtt{a}\,\mathtt{b}[(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)][(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)]$ and $\frac{1}{z} = 1/4$. The family of z strings and the corresponding index are as follows:*

i	0	1	2	3	4	5
	a	b	a	b	b	b
	a	b	a	b	a	b
	b	b	a	b	b	a
	b	b	a	b	a	a

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$y[i]$	a	b	a	b	b	b	a	b	a	b	a	b	b	b	a	b	b	a	b	b	a	b	a	a
$\mathsf{WSA}[i]$	17	22	10	20	8	6	0	14	2	5	16	21	9	19	7	13	1	4	15	18	12	3		
$\mathcal{L}[\mathsf{WSA}[i]]$	1	2	2	4	4	5	5	4	4	1	2	3	3	5	5	5	5	2	3	5	5	3		
$\mathsf{wLCP}[i]$	0	1	1	2	3	4	4	2	3	0	1	2	2	3	4	3	4	1	2	3	4	2		

5 Experimental Results

We have implemented algorithms ST-PSA, AC-PSA, and UF-PSA to compute the PSA. The programs have been implemented in the C++ programming language and developed under the GNU/Linux operating system.

[1] Another optimization—that would require a small tweak in the definition of the WSA—would be to consider, for each position $0 \le i \le n - 1$, the multiset of strings $F = \{y[j \,..\, \mathcal{L}[j]] : j = i \pmod n\}$ and repeatedly remove the shortest $f \in F$ that is a prefix of some other element of F until this is no longer possible.

The input parameters for all programs are a string of length n and an integer array of size n for the corresponding Π-valid intervals. The output of all programs is the PSA. The source code is distributed at `https://github.com/YagaoLiu/WSA` under the GNU General Public License. For comparison purposes, we used the implementation of the PST from [7] which has a similar interface (`https://bitbucket.org/kociumaka/weighted_index`). All experiments have been conducted on a Desktop PC using one core of Intel Xeon CPU E5-2640 at 2.60GHz and 64GB of RAM. All programs have been compiled with `g++` version 6.2.0 at optimization level 3 (-O3).

It is well-known, among practitioners and elsewhere, that optimal RMQ data structures for on-line $\mathcal{O}(1)$-time querying carry high constants in their preprocessing and querying time [5]. One would not thus expect that algorithm LCP-PSA performs well in practice. Indeed, we have implemented LCP-PSA but we omit its presentation here since it was not competitive for the same inputs.

To evaluate the time and space performance of our implementations, we used *synthetic* and *real* weighted sequences over the DNA alphabet ($\sigma = 4$). (In most real-world applications, weighted sequences are over the DNA alphabet.) Given a weighted sequence of length n and a probability threshold $1/z$, we constructed a single string of length nz and an integer array of size nz for the corresponding Π-valid intervals (Theorem 13), which we then used as our inputs.

Synthetic weighted DNA sequences. In the first experiment, we used synthetic weighted DNA sequences. The weighted sequences were of length ranging from $125,000$ to $4,000,000$. For each length, four different degeneracy percentages, denoted by δ, were used: 1%, 5%, 10% and 20% (percentage of positions where at least two letters with positive probability exist). The probability threshold was set to $1/8$. The strings obtained from the weighted sequences were thus of length ranging from $1,000,000$ to $32,000,000$. The results are plotted in Figures 1 and 2. In Figure 1 we observe that: (i) AC-PSA, UF-PSA and PST run in *linear* time; (ii) ST-PSA runs in (slightly) *super-linear* time; (iii) the array-based implementations run faster when δ increases. Observations (i) and (ii) confirm the theoretical findings. Observation (iii) is explained by the fact that when δ increases, the Π-valid intervals over the string are shorter on average, and thus fewer comparisons are generally required to resolve ties and thus to obtain the final order. In Figure 2 we observe that: (i) all four implementations run in *linear* space; (ii) PST is by far the most space *inefficient* of the four implementations; (iii) ST-PSA is the most space *efficient* of the three implementations. Observation (i) confirms the theoretical findings. Observations (ii) and (iii) are easily explained by the hidden constant factors in the corresponding space complexities.

Real weighted DNA sequences. In the second experiment, we created real weighted DNA sequences by combining the Genome Reference Consortium Human Build 37 (GRCh37) with the variants obtained from the 1000 Genomes Project (October 2011 Integrated Variant Set release) [1]. Specifically we made use of human chromosome 21 data. We randomly extracted fragments of length ranging from $125,000$ to $4,000,000$ from the generated weighted sequence. The probability threshold was set to $1/8$. The results are plotted in Figure 3. In both plots (time and space) we observe that the performance is analogous to the performance with the synthetic data of $\delta = 1\%$. This is explained by the fact that δ is found to be 0.7% in the weighted sequence of chromosome 21.

Whole-chromosome weighted DNA sequences. In the third experiment, we applied our three algorithms, ST-PSA, AC-PSA and UF-PSA, on whole-chromosome weighted DNA sequences. Specifically we combined human (GRCh37) chromosomes 18 to 22 and the corresponding variants (October 2011 Integrated Variant Set release). The weighted sequences were constructed in the same way as in the second experiment. In addition, long prefixes (or suffixes) of the chromosome sequences consisting solely of unknown bases (letter `N`) were omitted. The performance results are depicted in Table 1. As shown before, (i) UF-PSA is the fastest but the most space inefficient; (ii) ST-PSA is the most space efficient but the slowest; and (iii) AC-PSA lies in between as a reasonable trade off. We stress that it was not possible to apply PST due to its memory requirements which far exceed the capacity (64GB) of the machine used.

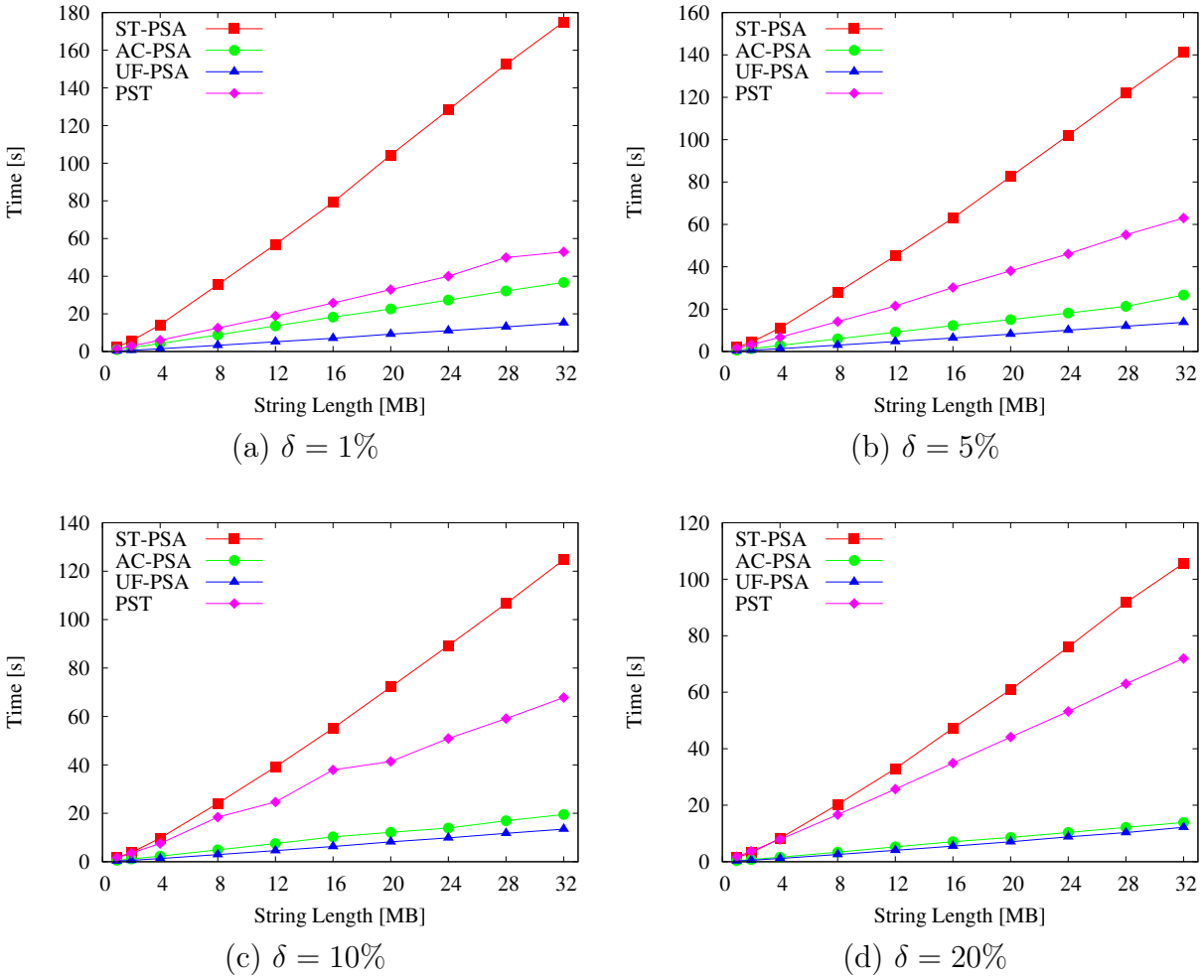


Figure 1: Elapsed time of ST-PSA, AC-PSA, UF-PSA, and PST on synthetic weighted DNA sequences of length ranging from 1MB to 32MB.

Human chromosome	String length (MB)	ST-PSA		AC-PSA		UF-PSA	
		Time (s)	Space (GB)	Time (s)	Space (GB)	Time (s)	Space (GB)
18	624	11512	17.066	4086	34.132	1446	60.533
19	472	7434	12.921	1704	25.842	1207	50.127
20	503	5013	13.763	714	27.524	422	36.052
21	309	2816	8.469	486	16.937	260	22.281
22	281	2706	7.701	371	15.401	160	19.778

Table 1: Elapsed time and peak memory usage of ST-PSA, AC-PSA and UF-PSA on whole-chromosome weighted DNA sequences.

Searching patterns on-line. The main scope of this paper is on the time- and space-efficient construction of the PSA; it is neither on its representation nor on its querying. As a data structure, the PST (resp. PSA) is essentially a suffix tree (resp. suffix array) with some of its nodes being lifted up (resp. with some LCP values being decreased). Thus, in practical terms, querying them is analogous to the standard setting and is

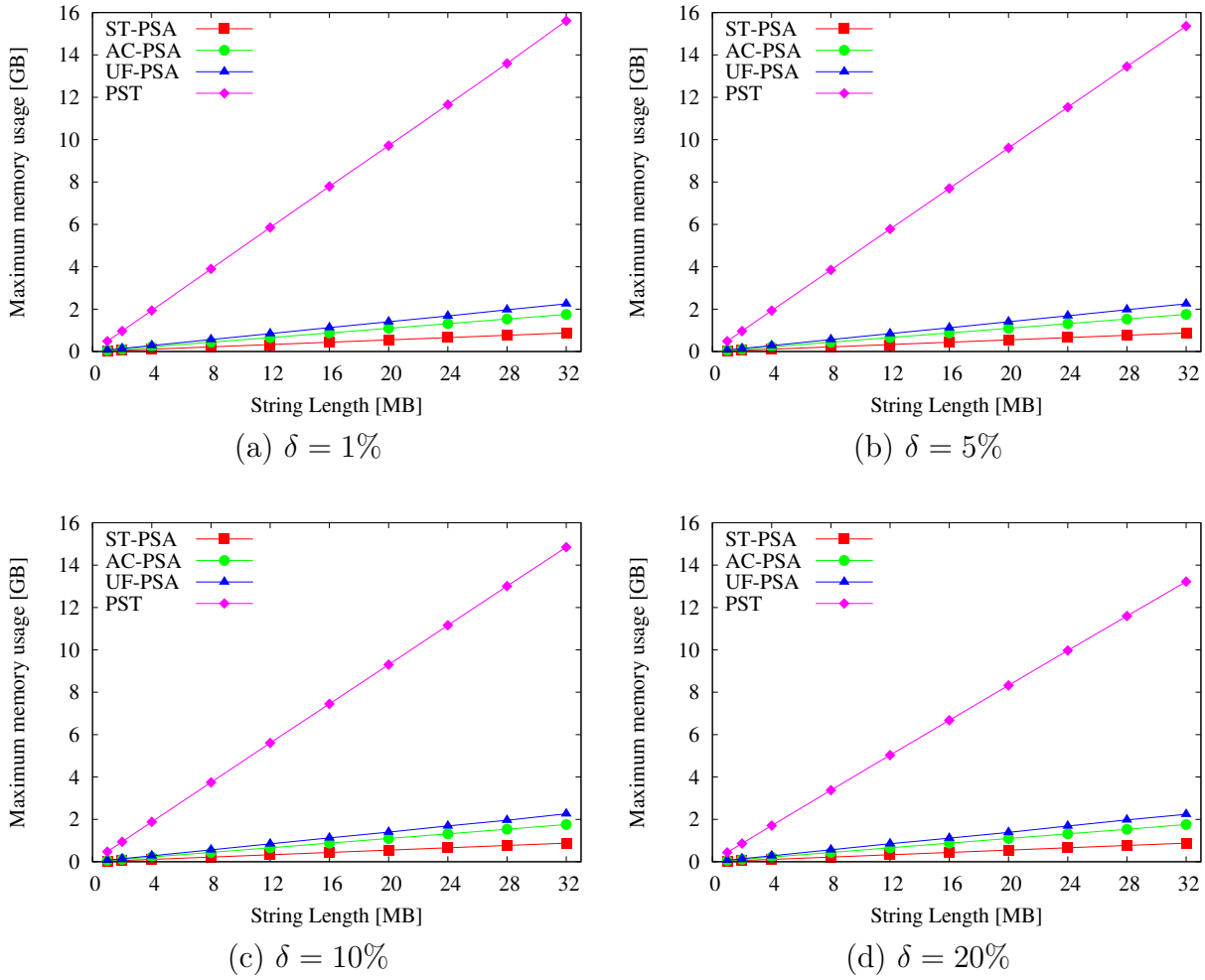
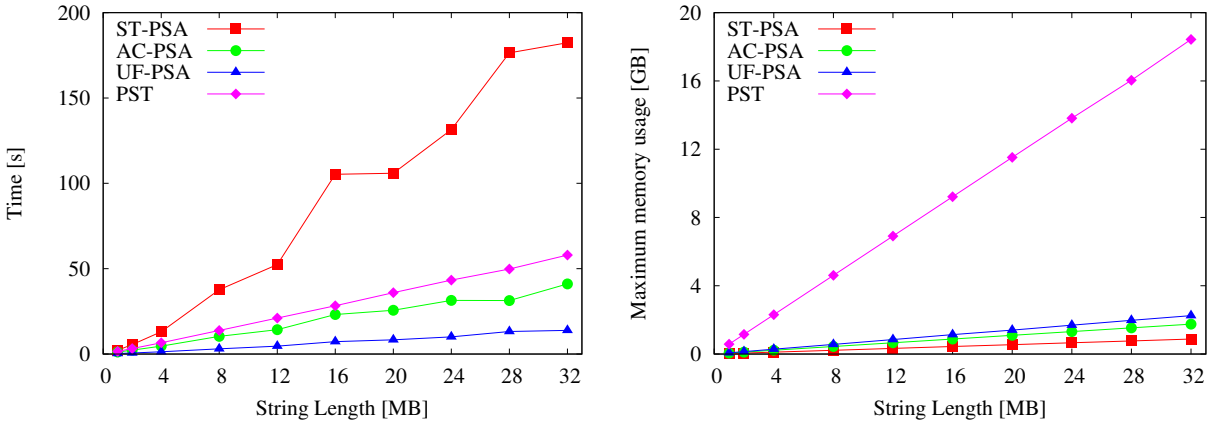


Figure 2: Peak memory usage of ST-PSA, AC-PSA, UF-PSA, and PST on synthetic weighted DNA sequences of length ranging from 1MB to 32MB.

thus beyond our focus. We conducted the following experiment as proof of concept. We used a synthetic weighted DNA sequence to generate a string of length $n = 2$MB. We generated $10,000$ uniformly random patterns each of length m, for all $m \in \{32, 64, \ldots, 1024\}$. We recorded the total elapsed time to search for all the patterns (decision queries), for each m separately, *after* constructing the PST and the PSA. For PST we used the $\mathcal{O}(m)$-time forward search and for PSA we used the $\mathcal{O}(\log n + m)$-time algorithm of [34] for the standard setting. The search time over PST was consistently one order of magnitude faster than searching over the PSA. This was expected as searching over suffix arrays using the algorithm of [34] entails a factor logarithmic in n as well as higher constant factors due to querying an RMQ data structure.

6 Conclusions

Given a string x of length n, the suffix array is the lexicographically sorted list of the n suffixes of x. It has been introduced as a space efficient alternative to suffix trees for indexing strings. In property indexing, we are additionally given a set of Π-valid intervals over x. The property suffix tree is an indexing data structure that



(a) Elapsed time (b) Peak memory usage

Figure 3: Elapsed time and peak memory usage of ST-PSA, AC-PSA, UF-PSA, and PST on real weighted DNA sequences of length ranging from 1MB to 32MB obtained from human chromosome 21.

takes property Π into account. It has been introduced as a data structure for indexing weighted sequences; it can be constructed in $\mathcal{O}(n)$ time.

The main contribution of this paper is the presentation of several time and space efficient algorithms to *directly* construct the property suffix array, the analogous array-based indexing data structure, culminating in an $\mathcal{O}(n)$-time construction. We also present an extensive experimental evaluation confirming *fully* our theoretical findings and justifying the motivation for the contributions of this paper. Specifically, we show that the peak memory usage of our implementations become at least one order of magnitude smaller than the one of property suffix tree when n grows.

Acknowledgments

Panagiotis Charalampopoulos was supported in part by the European Research Council grant TOTAL agreement no. 677651 and an A.G. Leventis Foundation educational grant.

References

[1] 1000 Genomes Project Consortium, Adam Auton, Lisa D. Brooks, Richard M. Durbin, Erik P. Garrison, Hyun Min M. Kang, Jan O. Korbel, Jonathan L. Marchini, Shane McCarthy, Gil A. McVean, and Gonçalo R. Abecasis. A global reference for human genetic variation. *Nature*, 526(7571):68–74, October 2015. doi:10.1038/nature15393.

[2] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004. URL: https://doi.org/10.1016/S1570-8667(03)00065-0, doi:10.1016/S1570-8667(03)00065-0.

[3] Charu C. Aggarwal and Philip S. Yu. A survey of uncertain data algorithms and applications. *IEEE Trans. Knowl. Data Eng.*, 21(5):609–623, 2009. URL: https://doi.org/10.1109/TKDE.2008.190, doi:10.1109/TKDE.2008.190.

[4] Hayam Alamro, Golnaz Badkobeh, Djamal Belazzougui, Costas S. Iliopoulos, and Simon J. Puglisi. Computing the antiperiod(s) of a string. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual*

Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy, volume 128 of *LIPIcs*, pages 32:1–32:11. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. URL: https://doi.org/10.4230/LIPIcs.CPM.2019.32, doi:10.4230/LIPIcs.CPM.2019.32.

[5] Mai Alzamel, Panagiotis Charalampopoulos, Costas S. Iliopoulos, and Solon P. Pissis. How to answer a small batch of rmqs or LCA queries in practice. In *IWOCA 2017*, pages 343–355, 2017. URL: https://doi.org/10.1007/978-3-319-78825-8_28, doi:10.1007/978-3-319-78825-8_28.

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

[7] Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P. Pissis, and Jakub Radoszewski. Indexing weighted sequences: Neat and efficient. *Information and Computation*, 270:104462, 2020. URL: http://www.sciencedirect.com/science/article/pii/S0890540119300781, doi:https://doi.org/10.1016/j.ic.2019.104462.

[8] Carl Barton, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Efficient index for weighted sequences. In *CPM 2016*, pages 4:1–4:13, 2016. URL: https://doi.org/10.4230/LIPIcs.CPM.2016.4, doi:10.4230/LIPIcs.CPM.2016.4.

[9] Carl Barton, Chang Liu, and Solon P. Pissis. On-line pattern matching on uncertain sequences and applications. In *COCOA 2016*, pages 547–562, 2016. URL: https://doi.org/10.1007/978-3-319-48749-6_40, doi:10.1007/978-3-319-48749-6_40.

[10] Carl Barton, Chang Liu, and Solon P. Pissis. Fast average-case pattern matching on weighted sequences. *Int. J. Found. Comput. Sci.*, 29(8):1331–1343, 2018. URL: https://doi.org/10.1142/S0129054118430062, doi:10.1142/S0129054118430062.

[11] Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit. Practical range minimum queries revisited. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, volume 75 of *LIPIcs*, pages 12:1–12:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. URL: https://doi.org/10.4230/LIPIcs.SEA.2017.12, doi:10.4230/LIPIcs.SEA.2017.12.

[12] Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005. URL: https://doi.org/10.1016/j.jalgor.2005.08.001, doi:10.1016/j.jalgor.2005.08.001.

[13] Sudip Biswas, Manish Patil, Sharma V. Thankachan, and Rahul Shah. Probabilistic threshold indexing for uncertain strings. In *EDBT 2016*, pages 401–412, 2016. URL: https://doi.org/10.5441/002/edbt.2016.37, doi:10.5441/002/edbt.2016.37.

[14] Panagiotis Charalampopoulos, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. Property suffix array with applications. In *LATIN 2018*, pages 290–302, 2018. URL: https://doi.org/10.1007/978-3-319-77404-6_22, doi:10.1007/978-3-319-77404-6_22.

[15] Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. On-line weighted pattern matching. *Inf. Comput.*, 266:49–59, 2019. URL: https://doi.org/10.1016/j.ic.2019.01.001, doi:10.1016/j.ic.2019.01.001.

[16] Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988. URL: https://doi.org/10.1137/0217049, doi:10.1137/0217049.

[17] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[18] Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Krzysztof Stencel, and Tomasz Walen. New simple efficient algorithms computing powers and runs in strings. *Discrete Applied Mathematics*, 163:258–267, 2014. URL: https://doi.org/10.1016/j.dam.2013.05.009, doi:10.1016/j.dam.2013.05.009.

[19] Héctor Ferrada and Gonzalo Navarro. Improved range minimum queries. *J. Discrete Algorithms*, 43:72–80, 2017. URL: https://doi.org/10.1016/j.jda.2016.09.002, doi:10.1016/j.jda.2016.09.002.

[20] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. URL: https://doi.org/10.1137/090779759, doi:10.1137/090779759.

[21] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. URL: https://doi.org/10.1016/0022-0000(85)90014-5, doi:10.1016/0022-0000(85)90014-5.

[22] Wing-Kai Hon, Manish Patil, Rahul Shah, and Sharma V. Thankachan. Compressed property suffix trees. *Inf. Comput.*, 232:10–18, 2013. doi:10.1016/j.ic.2013.09.001.

[23] Costas S. Iliopoulos and Mohammad Sohel Rahman. Faster index for property matching. *Inf. Process. Lett.*, 105(6):218–223, 2008. URL: https://doi.org/10.1016/j.ipl.2007.09.004, doi:10.1016/j.ipl.2007.09.004.

[24] M. T. Juan, J. J. Liu, and Y. L. Wang. Errata for "faster index for property matching". *Inf. Process. Lett.*, 109(18):1027–1029, 2009. URL: https://doi.org/10.1016/j.ipl.2009.06.009, doi:10.1016/j.ipl.2009.06.009.

[25] Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi, and Bella Zhukova. Engineering external memory induced suffix sorting. In Sándor P. Fekete and Vijaya Ramachandran, editors, *Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017*, pages 98–108. SIAM, 2017. URL: https://doi.org/10.1137/1.9781611974768.8, doi:10.1137/1.9781611974768.8.

[26] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In Gregory Kucherov and Esko Ukkonen, editors, *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Lille, France, June 22-24, 2009, Proceedings*, volume 5577 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2009. URL: https://doi.org/10.1007/978-3-642-02441-2_17, doi:10.1007/978-3-642-02441-2_17.

[27] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. URL: https://doi.org/10.1145/1217856.1217858, doi:10.1145/1217856.1217858.

[28] Samuel Karlin, Ghassan Ghandour, Friedemann Ost amd Simon Tavare, and Laurence J. Korn. New approaches for computer analysis of nucleic acid sequences. *Proceedings of the National Academy of Sciences of the United States of America*, 80(18):5660–5664, 1983. doi:10.1073/pnas.80.18.5660.

[29] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM 2001*, pages 181–192, 2001. URL: https://doi.org/10.1007/3-540-48194-X_17, doi:10.1007/3-540-48194-X_17.

[30] Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A linear time algorithm for seeds computation. *CoRR*, arXiv:1107.2422v2, 2011. URL: https://arxiv.org/abs/1107.2422v2.

[31] Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Pattern matching and consensus problems on weighted sequences and profiles. *Theory Comput. Syst.*, 63(3):506–542, 2019. URL: https://doi.org/10.1007/s00224-018-9881-2, doi:10.1007/s00224-018-9881-2.

[32] Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Efficient algorithms for shortest partial seeds in words. *Theor. Comput. Sci.*, 710:139–147, 2018. URL: https://doi.org/10.1016/j.tcs.2016.11.035, doi:10.1016/j.tcs.2016.11.035.

[33] Tsvi Kopelowitz. The property suffix tree with dynamic properties. *Theor. Comput. Sci.*, 638:44–51, 2016. URL: https://doi.org/10.1016/j.tcs.2016.02.033, doi:10.1016/j.tcs.2016.02.033.

[34] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. URL: https://doi.org/10.1137/0222058, doi:10.1137/0222058.

[35] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *DCC 2009*, pages 193–202, 2009. URL: https://doi.org/10.1109/DCC.2009.42, doi:10.1109/DCC.2009.42.

[36] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):4, 2007. URL: https://doi.org/10.1145/1242471.1242472, doi:10.1145/1242471.1242472.

[37] Jakub Radoszewski and Tatiana A. Starikovskaya. Streaming k-mismatch with error correcting and applications. In *DCC 2017*, pages 290–299, 2017. URL: https://doi.org/10.1109/DCC.2017.14, doi:10.1109/DCC.2017.14.

[38] Peter Weiner. Linear pattern matching algorithms. In *SWAT (FOCS), 1973*, pages 1–11, 1973. URL: https://doi.org/10.1109/SWAT.1973.13, doi:10.1109/SWAT.1973.13.