Property Suffix Array with Applications

Panagiotis Charalampopoulos, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis(✉)

Department of Informatics, King's College London, London, UK
{panagiotis.charalampopoulos,costas.iliopoulos,chang.2.liu, solon.pissis}@kcl.ac.uk

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. A suffix-tree-like index over these 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 $\mathcal{O}(nz)$.

1 Introduction

Property matching, introduced in [4], 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 $[a, b] \in \Pi$.

Most of the prevalent text indexing data structures are built over the suffixes of the text [22]. However, by introducing the property Π only some prefixes of each suffix are now valid. The authors in [4] 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. Recently, the authors in [5,6] have presented an $\mathcal{O}(n)$-time algorithm for the construction

P. Charalampopoulos—Supported by the Graduate Teaching Scholarship scheme of the Department of Informatics at King's College London and by the A.G. Leventis Foundation.

M. A. Bender et al. (Eds.): LATIN 2018, LNCS 10807, pp. 290–302, 2018.
https://doi.org/10.1007/978-3-319-77404-6_22

of the PST that also works for integer alphabets. This is based on a technique by Kociumaka, Radoszewski, Rytter and Waleń for answering off-line weighted ancestor queries in suffix trees (see the Appendix of [5]). A dynamic instance of Property Indexing has also been studied in [19], where the author also makes use of the suffix tree.

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 [14,15]. 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 [20]. 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 [1].

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 $x = \texttt{acababaab}$ 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
SA$[i]$	6	7	4	2	0	8	5	3	1
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 PSA$[r-1] = j$ and 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 [2] 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 $\frac{1}{z}$ [8,17]. 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 [4], where a reduction to Property Indexing of a text of size $\mathcal{O}(nz^2 \log z)$ was proposed. The authors in [6] 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 pattern matching queries on X in optimal time. The same index as the one in [6] was first presented in [7] 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 [6,10].

All these indexes [4,6,7] are based on constructing and traversing the suffix tree. Here, using our solution to problem Property Suffix Array *and* the main idea of [6], we show how to construct directly an array data structure within the same complexities. Moreover, we present experiments that show the advantage of our new data structure: as expected, it requires much less space than the one of [6,7]. 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 Sect. 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 $\mathcal{O}(n)$. In Sect. 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 Sect. 5, we present an experimental evaluation of the proposed algorithms.

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 $x[\mathsf{SA}[r-1]\,..\,n-1] < x[\mathsf{SA}[r]\,..\,n-1]$ [20]. 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 [21], iSA, and LCP [16] 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 [9], 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 [9] and the *doubling technique* [20] 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 [12]. 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 suffixes up to length 2^{k+1}; at each level, $\mathcal{O}(n \log n)$ time is required to sort these factors using any optimal comparison-based sorting algorithm [11].

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 a completely different 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 [20] 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*$(x, n, \mathcal{L})$
2 **for** $i \leftarrow 0$ to $n - 1$ **do**
3 $\mathsf{PSA}[i] \leftarrow i$;
4 Sort PSA using the following comparison rule for $\mathsf{PSA}[i]$ and $\mathsf{PSA}[j]$:
5 **if** $x[i] < x[j]$ **then** $\mathsf{PSA}[i] < \mathsf{PSA}[j]$;
6 **else if** $x[i] > x[j]$ **then** $\mathsf{PSA}[i] > \mathsf{PSA}[j]$;
7 **else** $\mathsf{PSA}[i] = \mathsf{PSA}[j]$;
8 Rank the elements of PSA and store their ranks in $\mathsf{Rank}_{\mathsf{PSA}}$;
9 $\mathsf{Rank}_{\mathsf{PREF}} \leftarrow \mathsf{Rank}_{\mathsf{PSA}}$;
10 **for** $k \leftarrow 1$ to $\lfloor \log n \rfloor$ **do**
11 Construct an array $\mathcal{A}$ of pairs: $\mathcal{A}[i] = (\mathsf{Rank}_{\mathsf{PREF}}[i], \mathsf{Rank}_{\mathsf{PREF}}[i + 2^{k-1}])$;
12 Sort the pairs in $\mathcal{A}$ using radix sort and store their ranks in $\mathsf{Rank}_{\mathsf{CURR}}$;
13 Sort PSA using $\mathcal{L}$, $\mathsf{Rank}_{\mathsf{PSA}}$, $\mathsf{Rank}_{\mathsf{CURR}}$, and Fact 2 for the comparison;
14 Rank the elements of PSA and store their new ranks in $\mathsf{Rank}_{\mathsf{PSA}}$;
15 $\mathsf{Rank}_{\mathsf{PREF}} \leftarrow \mathsf{Rank}_{\mathsf{CURR}}$;
16 **return** PSA;

3.2 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 [9,20]. We can then perform any optimal comparison-based sorting algorithm (use 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              PSA[i] < PSA[j];
10         else
11             if L[SA[i]] < L[SA[j]] then
12                 PSA[i] < PSA[j];
13             else
14                 PSA[i] > PSA[j];
15     return PSA;
```

3.3 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 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 if $(f_{i_1}, j_1 - i_1) \le (f_{i_2}, j_2 - i_2)$ then $x[i_1 \,..\, j_1] \le x[i_2 \,..\, j_2]$.*

Proof. Note that $x[i \,..\, j]$ is a prefix of $x[\mathsf{SA}[f_i] \,..\, n-1]$. Thus if

- either $f_{i_1} < f_{i_2}$
- or $f_{i_1} = f_{i_2}$ and $j_1 - i_1 \le j_2 - i_2$

then we have that $x[i_1 \,..\, j_1] \le x[i_2 \,..\, j_2]$. □

Example 5 (Running example).

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

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

The computational problem is to compute f_i efficiently for all i; for this we rely on the Union-Find data structure [11] in a similar manner as the authors in [18]. Our technique also resembles the technique by Kociumaka, Radoszewski, Rytter and Waleń for answering off-line weighted ancestor queries in trees; it can be found in the Appendix of [5]. Union-Find maintains a partition of $\{0, 1, \ldots, n-1\}$, where each set has a representative element, and supports three basic operations:

- $\mathsf{MakeSet}(n)$ creates n new sets $\{0\}, \{1\}, \ldots, \{n-1\}$, where the representative index of set $\{i\}$ is i.
- $\mathsf{Find}(i)$ returns the representative of the set containing i.
- $\mathsf{Union}(i, j)$ 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 ([13])**.** *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 $\mathsf{MakeSet}(n)$;
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
LCP$[i]$	0	1	2	3	1	0	1	2	0

i	0	1	2	3	4	5	6	7	8
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 Find(iSA[3]) = 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 [20] 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 $\mathcal{O}(n+q)$.*

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 $X[i] \dots X[j]$. 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 positive 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 [5]. 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 [5] 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 ([5]). *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.*

The idea is to create a new special weighted sequence Y by concatenating these $\lfloor z \rfloor$ special weighted sequences. At this point we view Y as the standard string y of length $n\lfloor z \rfloor$ (at most one letter per position has a positive 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. We then construct the PSA for y and $\mathcal{L}$.

We are not done yet since a string of length m occurring at a position i of X may occur at several positions $j_0, j_1, \dots, j_{k-1}$ in y, with $j_p = i \,(\mathrm{mod}\, n)$ and $\mathcal{L}[j_p] = m$ for all $0 \leq p < k$. We naturally want to keep *one* of these occurrences. We do that as follows: we identify maximal intervals $[r, s]$ in the PSA satisfying $\mathcal{L}[\mathsf{PSA}[q]] = \mathsf{pLCP}[t] = m$ for all $r - 1 \leq q \leq s$ and $r \leq t \leq s$; for each such interval, we consider all of the indices in $\{\mathsf{PSA}[q] | r - 1 \leq q \leq s\}$ modulo n, we bucket sort the residuals, and finally 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 [20].

Example 16. Let $X = [(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)]\mathtt{bab}[(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)][(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)]$ 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
WSA$[i]$	17	22	10	20	8	6	0	14	2	5	16	21	9	19	7	13	1	4	15	18	12	3		
$\mathcal{L}$[WSA$[i]$]	1	2	2	4	4	5	5	4	4	1	2	3	3	5	5	5	5	2	3	5	5	3		
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 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. The input parameters for both programs are a string of length n and an integer array of size n for the corresponding Π-valid intervals. The output of both 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 [6] 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 2.60 GHz. 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 [3]. 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 too slow for the same inputs.

To evaluate the time and space performance of our implementations, we used synthetic weighted DNA sequences ($\sigma = 4$). We used the weighted sequences to create a new standard string y and compute the integer array $\mathcal{L}$ as described in Sect. 4. Thus given a weighted sequence of length n and a probability threshold $\frac{1}{z}$, we obtained a new string of length nz. In our experiments, we used weighted sequences of length ranging from 125,000 to 4,000,000; the probability threshold was set to 1/8. The strings obtained from weighted sequences are thus of length ranging from 1,000,000 to 32,000,000. The results are plotted in Figs. 1 and 2. In Fig. 1 we see that: (i) UF-PSA and PST run in *linear* time; (ii) ST-PSA runs in (slightly) *super-linear* time; and (iii) UF-PSA is the *fastest* of the three implementations. In Fig. 2 we see that: (i) all three implementations run in *linear* space; (ii) PST is by far the most space *inefficient* of the three implementations; and (iii) ST-PSA is the most space *efficient* of the three implementations. The presented experimental results confirm *fully* our theoretical findings and justify the motivation for the contributions of this paper.

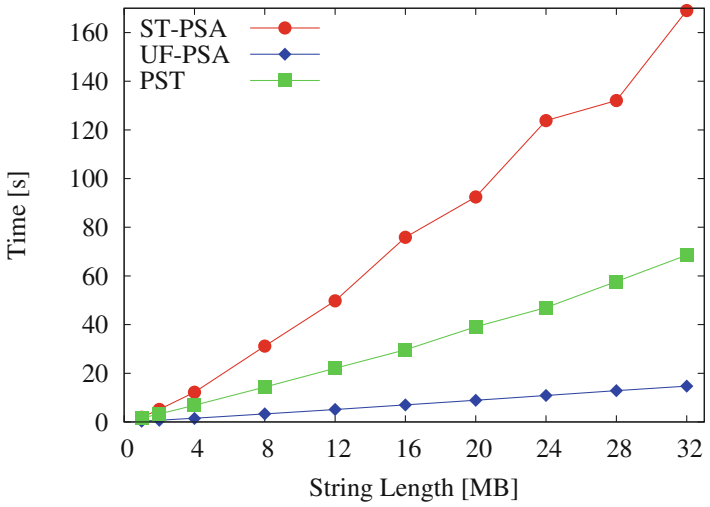

Fig. 1. Elapsed time of ST-PSA, UF-PSA, and PST using synthetic texts of length ranging from 1 MB to 32 MB over the DNA alphabet.

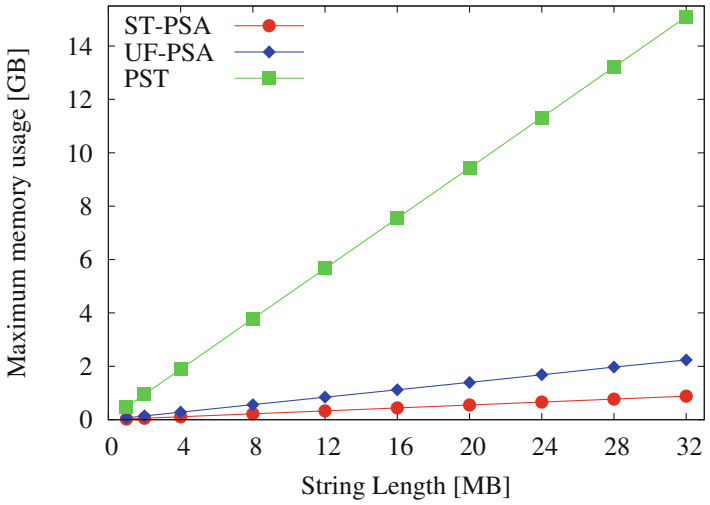

Fig. 2. Peak memory usage of ST-PSA, UF-PSA, and PST using synthetic texts of length ranging from 1 MB to 32 MB over the DNA alphabet.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**(1), 53–86 (2004)
2. Aggarwal, C.C., Yu, P.S.: A survey of uncertain data algorithms and applications. IEEE Trans. Knowl. Data Eng. **21**(5), 609–623 (2009)

3. Alzamel, M., Charalampopoulos, P., Iliopoulos, C.S., Pissis, S.P.: How to answer a small batch of RMQs or LCA queries in practice. In: IWOCA. LNCS. Springer International Publishing (2017, in press)
4. Amir, A., Chencinski, E., Iliopoulos, C., Kopelowitz, T., Zhang, H.: Property matching and weighted matching. Theor. Comput. Sci. **395**(2–3), 298–310 (2008)
5. Barton, C., Kociumaka, T., Liu, C., Pissis, S.P., Radoszewski, J.: Indexing weighted sequences: neat and efficient. CoRR abs/1704.07625v1 (2017)
6. Barton, C., Kociumaka, T., Liu, C., Pissis, S.P., Radoszewski, J.: Indexing weighted sequences: neat and efficient. CoRR abs/1704.07625v2 (2017)
7. Barton, C., Kociumaka, T., Pissis, S.P., Radoszewski, J.: Efficient index for weighted sequences. In: CPM. LIPIcs, vol. 54, pp. 4:1–4:13. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
8. Barton, C., Liu, C., Pissis, S.P.: On-line pattern matching on uncertain sequences and applications. In: Chan, T.-H.H., Li, M., Wang, L. (eds.) COCOA 2016. LNCS, vol. 10043, pp. 547–562. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48749-6_40
9. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000). https://doi.org/10.1007/10719839_9
10. Biswas, S., Patil, M., Thankachan, S.V., Shah, R.: Probabilistic threshold indexing for uncertain strings. In: EDBT. pp. 401–412 (2016). OpenProceedings.org
11. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms, 2nd edn. McGraw-Hill Higher Education, Pennsylvania (2001)
12. Crochemore, M., Iliopoulos, C., Kubica, M., Radoszewski, J., Rytter, W., Stencel, K., Walen, T.: New simple efficient algorithms computing powers and runs in strings. Discrete Appl. Math. **163**(Part 3), 258–267 (2014)
13. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. J. Comput. Syst. Sci. **30**(2), 209–221 (1985)
14. Iliopoulos, C.S., Rahman, M.S.: Faster index for property matching. Inf. Process. Lett. **105**(6), 218–223 (2008)
15. Juan, M.T., Liu, J.J., Wang, Y.L.: Errata for "faster index for property matching". Inf. Process. Lett. **109**(18), 1027–1029 (2009)
16. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A. (ed.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48194-X_17
17. Kociumaka, T., Pissis, S.P., Radoszewski, J.: Pattern matching and consensus problems on weighted sequences and profiles. In: ISAAC. LIPIcs, vol. 64, pp. 46:1–46:12. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
18. Kociumaka, T., Pissis, S.P., Radoszewski, J., Rytter, W., Walen, T.: Efficient algorithms for shortest partial seeds in words. Theor. Comput. Sci. **710**, 139–147 (2018)
19. Kopelowitz, T.: The property suffix tree with dynamic properties. Theor. Comput. Sci. **638**(C), 44–51 (2016)
20. Manber, U., Myers, E.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
21. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: DCC, pp. 193–202. IEEE (2009)
22. Weiner, P.: Linear pattern matching algorithms. In: SWAT (FOCS), pp. 1–11. IEEE Computer Society (1973)