Information Processing Letters ••• (••••) ••••••

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

All-pairs suffix/prefix in optimal time using Aho-Corasick space

Grigorios Loukides [a], Solon P. Pissis [b,c,*]

[a] Department of Informatics, King's College London, London, UK
[b] CWI, Amsterdam, the Netherlands
[c] Vrije Universiteit, Amsterdam, the Netherlands

ARTICLE INFO

Article history:
Received 27 September 2021
Received in revised form 8 February 2022
Accepted 24 April 2022
Available online xxxx
Communicated by Leah Epstein

Keywords:
Algorithms
Data structures
String algorithms
Aho-Corasick machine
Failure transition tree

ABSTRACT

The all-pairs suffix/prefix (APSP) problem is a classic problem in computer science with many applications in bioinformatics. Given a set $\{S_1, \ldots, S_k\}$ of k strings of total length n, we are asked to find, for each string S_i, $i \in [1, k]$, its longest suffix that is a prefix of string S_j, for all $j \neq i$, $j \in [1, k]$. Several algorithms running in the optimal $\mathcal{O}(n + k^2)$ time for solving APSP are known. All of these algorithms are based on suffix sorting and thus require space $\Omega(n)$ in any case. We consider the parameterized version of the APSP problem, denoted by ℓ-APSP, in which we are asked to output only the pairs whose suffix/prefix overlap is of length at least ℓ. We give an algorithm for solving ℓ-APSP that runs in the optimal $\mathcal{O}(n + |\mathsf{OUTPUT}_\ell|)$ time using $\mathcal{O}(n)$ space, where OUTPUT_ℓ is the set of output pairs. Our algorithm is thus optimal for the APSP problem as well by setting $\ell = 0$. Notably, our algorithm is fundamentally different from all optimal algorithms solving the APSP problem: it does not rely on sorting the suffixes of all input strings but on a novel traversal of the Aho-Corasick machine, and it thus requires space linear in the size of the machine.

1. Introduction

The all-pairs suffix/prefix (APSP) problem is a classic problem in computer science. It has many applications in bioinformatics because it is the first step in genome assembly [6]. Given a set $R = \{S_1, \ldots, S_k\}$ of k strings of total length n, the APSP problem asks us to find, for each string S_i, $i \in [1, k]$, its longest suffix that is a prefix of string S_j, for all $j \neq i$, $j \in [1, k]$. Gusfield et al. presented an algorithm running in the optimal $\mathcal{O}(n + k^2)$ time for solving APSP [7]. The algorithm is based on the generalized suffix tree [17] of R. Ohlebusch and Gog [12] gave another optimal algorithm which is based on the generalized suffix array [11] of R. Tustumi et al. [15] gave yet another optimal algorithm based on the generalized suffix array of R. Thus the common denominator of all existing optimal algorithms for APSP is that they rely on sorting the suffixes of all strings in R, and they thus require space $\Omega(n)$ in any case and for any alphabet.

There also exists a large body of works devoted to implementing algorithms for APSP that are suboptimal but practically fast on real-world datasets; see [5,14,9] and references therein for some of the state-of-the-art implementations. For a parallel implementation of the algorithm by Tustumi et al. see [10].

In this paper, we formalize the parameterized version of the APSP problem, denoted by ℓ-APSP, in which we are asked to output only the pairs in R whose suffix/prefix overlap is of length at least ℓ. ℓ-APSP is more attractive to study both from a theory as well as from a practical

* Corresponding author.
E-mail addresses: grigorios.loukides@kcl.ac.uk (G. Loukides), solon.pissis@cwi.nl (S.P. Pissis).

https://doi.org/10.1016/j.ipl.2022.106275

perspective. From a theory perspective, it is more interesting to have an algorithm that is optimal with respect to the actual size of the output. Specifically, even when $\ell = 0$, many pairs of strings may have no suffix/prefix overlap, and so the size of the meaningful output could be asymptotically smaller than $\Theta(k^2)$. From a practical perspective, we stress that most papers studying the APSP problem in fact considered the ℓ-APSP problem in their experiments.

The aforementioned algorithms solving APSP in optimal time do not explicitly consider ℓ-APSP. Gusfield et al. provide an extension of their main algorithm that solves 1-APSP in the optimal time [7]. We observe that an extra, trivial modification of this extended algorithm solves ℓ-APSP in the optimal $\mathcal{O}(n + |\mathsf{OUTPUT}_\ell|)$ time, where OUTPUT_ℓ is the set of output pairs: instead of considering every internal node of the generalized suffix tree, we must only consider internal nodes of string depth at least ℓ. It is perhaps less clear how one could modify the suffix-array-based algorithms presented in [12,15] to solve ℓ-APSP in the optimal time.

Here we give an algorithm running in the optimal $\mathcal{O}(n + |\mathsf{OUTPUT}_\ell|)$ time using $\mathcal{O}(n)$ space for solving ℓ-APSP. Our algorithm is thus optimal for the APSP problem as well by setting $\ell = 0$. Notably our algorithm is fundamentally different from all optimal algorithms for solving APSP. Specifically, our algorithm does not resort to suffix sorting. It relies on a novel traversal of the Aho-Corasick (AC) machine [1]; a finite-state machine that directly encodes all pairwise suffix/prefix overlaps (not only the longest one per string pair). It is thus somewhat surprising that the AC machine has not been used to obtain an optimal algorithm for the APSP problem—as we detail next, the AC machine has been employed for solving other APSP versions. In particular, our algorithm uses a tree induced from the AC machine, which we term the *failure transition tree*. We annotate, decompose, and carefully traverse this tree to infer only the longest suffix/prefix overlap per pair. Our algorithm thus requires space linear in the size of the AC machine, which may be *asymptotically smaller* than $\mathcal{O}(n)$ if prefix redundancy in R is non-negligible, or space $\Theta(n)$ in the worst case.

Other related work. In addition to ℓ-APSP that is formulated in this paper, there are two other versions of APSP that have been studied in the literature. The first version consists in enumerating all pairwise suffix/prefix overlaps (not necessarily the longest ones) in decreasing order of their lengths. This version of the problem was solved by Ukkonen [16], who used this solution as the crux of his classic linear-time implementation of the greedy algorithm for constructing approximate shortest common superstrings. Ukkonen's solution is based on a reversed BFS traversal of the AC machine. Note, however, that by enumerating all such suffix/prefix overlaps in decreasing order of their length, we cannot guarantee that we will enumerate the longest ones *per pair* within the optimal $\mathcal{O}(n + |\mathsf{OUTPUT}_\ell|)$ time. Thus, in some sense, this version of APSP is computationally harder than ℓ-APSP. The second APSP version studied consists in enumerating the *set* of longest suffix/prefix overlaps (not however their association with the corresponding pairs of strings) [2]. Since any suffix/prefix overlap in this set is a prefix of some input string, the size of this set is in $\mathcal{O}(n)$. This version of the problem was solved in the optimal $\mathcal{O}(n)$ time, independently, by Park et al. [13] and by Khan [8]. Note, however, that by enumerating all such longest suffix/prefix overlaps, we cannot trivially infer their association with the corresponding pairs of strings in optimal time. Thus, in some sense, this version of APSP is computationally easier than ℓ-APSP. Park et al.'s solution is based on the efficient assessment of sets sizes along *failure transition* paths in the AC machine. Our work employs ideas that are similar to the ones of Ukkonen [16] and to the ones of Park et al. [13]. Khan's solution is based on a careful simulation and adaptation of Gusfield et al.'s algorithm for APSP [7] on the AC machine.

Paper organization. Section 2 presents some preliminaries and our main result. Section 3 presents the algorithm we develop. Section 4 concludes the paper.

2. Preliminaries and main result

An *alphabet* Σ is a finite nonempty set whose elements are called *letters*. A *string* $S = S[1..m]$ is a sequence of *length* $|S| = m$ over Σ. The *empty string* ε is the string of length 0. The *concatenation* of two strings S and T is the string composed of the letters of S followed by the letters of T; it is denoted by $S \cdot T$ or simply by ST. For $1 \leq i \leq j \leq m$, $S[i]$ denotes the ith letter of S, and the fragment $S[i..j]$ denotes an *occurrence* of the underlying *substring* $P = S[i] \cdots S[j]$. We say that P *occurs* at (starting) *position* i in S. A fragment $S[i..j]$ is a *suffix* of S if $j = m$, and it is a *prefix* of S if $i = 1$. A substring of S is called *proper* if it is not equal to S. Given two strings S and T, a *suffix/prefix overlap* of S and T is a suffix U of S that is a prefix of T; when U is the longest such suffix, then U is called the *maximal suffix/prefix overlap* of S and T.

We now define the main problem investigated in this paper.

ALL-PAIRS SUFFIX/PREFIX OF LENGTH AT LEAST ℓ (ℓ-APSP)
Input: A set $R = \{S_1, \ldots, S_k\}$ of k strings of total length n over an alphabet $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$, and an integer $\ell \geq 0$.
Output: For each string S_i, $i \in [1, k]$, return a set E_i, such that $j_d \in E_i$ if and only if S_i and S_j, $j \in [1, k]$, have a maximal suffix/prefix overlap of length $d \geq \ell$.

We denote the output of ℓ-APSP for all $i \in [1, k]$ by OUTPUT_ℓ. Our main result is the following.

Theorem 1. *The ℓ-APSP problem can be solved in the optimal $\mathcal{O}(n + |\mathsf{OUTPUT}_\ell|)$ time using $\mathcal{O}(n)$ space.*

The AC machine is a finite-state machine which was introduced in [1] to find all occurrences of a set $R = \{S_1, \ldots, S_k\}$ of $k > 1$ strings within an input text. For conceptual convenience, we assume that no string in R is a prefix of another string in R (i.e., that R is

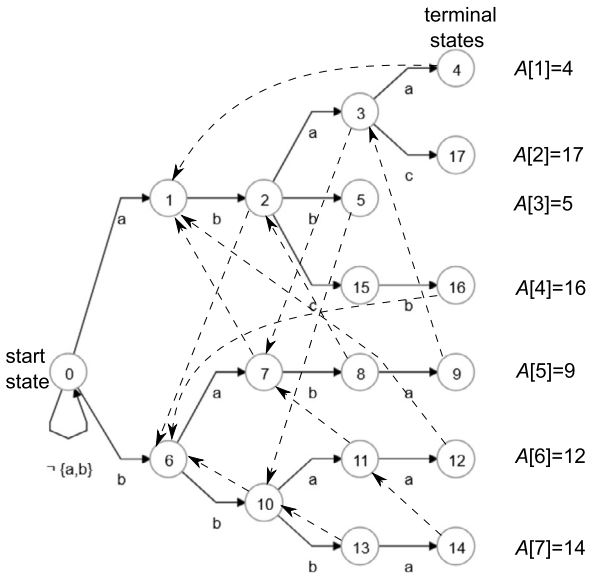

Fig. 1. The AC machine over $R = \{\texttt{abaa}, \texttt{abac}, \texttt{abb}, \texttt{abcb}, \texttt{baba}, \texttt{bbaa}, \texttt{bbba}\}$. Solid arrows correspond to goto transitions and dashed arrows to failure transitions. The lexicographic ranks of the strings in R are the indices of array A.

prefix-free). If this is not the case, we simply append a terminal letter $\# \notin \Sigma$ to every string in R. Let us assume that we need to find all occurrences of strings in $R = \{\texttt{abaa}, \texttt{abac}, \texttt{abb}, \texttt{abcb}, \texttt{baba}, \texttt{bbaa}, \texttt{bbba}\}$ in some other string (the text). To perform this by a single scan of the text, we can employ the AC machine in Fig. 1. The AC machine has two types of transitions: *goto* transitions and *failure transitions*. The goto transitions are shown as solid arrows in Fig. 1. The AC machine goes from a state s to another state t when a letter a of the alphabet is consumed. This is represented by a goto transition, denoted by $g(s, a) = t$. An AC machine can have a goto transition $g(s, a) = t$ for each state s of the machine and each letter a of the alphabet. If the AC machine is at a state s and there is no goto transition from s, for some letter of the alphabet, then a failure transition is made. The *start state* of the AC machine has one goto transition for each letter of the alphabet and thus no failure transition. On the contrary, every other state has exactly one failure transition to another state (including to the start state). In Fig. 1, a dashed arrow shows a failure transition to a state other than the start state 0, while all failure transitions that go to the start state are omitted. If the AC machine is in state s and makes a failure transition to state t, we write $f(s) = t$. We say that state s *corresponds* to string U if and only if U is the string spelled out by following the shortest path of goto transitions from the start state to s. Aho and Corasick [1] proved the following key lemma on suffix/prefix overlaps.

Lemma 1 *(Aho-Corasick lemma [1]). Let state s correspond to string U and state t correspond to string V in the AC machine of a set R of strings. Then, we have that $f(s) = t$ if and only if V is the longest proper suffix of U that is also a prefix of some string in R.*

For example, in the AC machine of Fig. 1, let $s = 16$ and $t = 6$. Then, $f(16) = 6$. Note, $V = \texttt{b}$ is the longest proper suffix of $U = \texttt{abcb}$ that is also a prefix of some string in R.

The Aho-Corasick lemma implies that the depth of the states on a failure transition path is *monotonically decreasing*. Let state s correspond to string U. Further let the failure transition $f(s) = t$ and $S_j \in R$ be any string corresponding to a goto path containing t. Then U has a suffix/prefix overlap V with S_j, where V is the string corresponding to t. Thus, all suffix/prefix overlaps between U and the strings in R can be found by following the failure transition from s to another state s', then following again the failure transition from s' to another state s'', and so on. Let U be some string from R. The state corresponding to U is termed *terminal state*. By repeatedly following the failure transitions from terminal states of the AC machine, we can find all pairwise suffix/prefix overlaps of all strings in R. This observation was used by Ukkonen in [16].

The AC machine for a set $R = \{S_1, \ldots, S_k\}$ of k strings of total length n over an ordered alphabet Σ can be constructed in $\mathcal{O}(n \log |\Sigma|)$ time without suffix sorting [3]. Dori and Landau showed that, when $\Sigma = \{1, \ldots, n^{\mathcal{O}(1)}\}$, the AC machine can be constructed in $\mathcal{O}(n)$ time using suffix sorting [4]. For each state s, let $L(s)$ be the set of indices i such that the goto path that spells out string S_i contains state s. Let $d(s)$ denote the *string depth* of state s, that is, the length of the string corresponding to s. By performing a DFS traversal on the AC machine using the goto transitions, we can construct in $\mathcal{O}(n)$ time an array $A = A[1..k]$, which stores $A[r] = s$, $r \in [1, k]$, if and only if for state s and string S_i, $L(s) = \{i\}$, $d(s) = |S_i|$, and S_i has lexicographic rank r in R. This array is $A = [4, 17, 5, 16, 9, 12, 14]$ in Fig. 1; e.g., $\texttt{abac}$ has lexicographic rank 2 in R.

3. The algorithm

Our new optimal algorithm has two main stages. In the first one, we show that any instance of ℓ-APSP can be reduced to some instance of an abstract tree problem in linear time. In the second stage, we solve the tree problem in optimal time and linear space. A full running example is provided along the way.

3.1. Reducing the ℓ-APSP problem to the ULIT problem

By $[i, j]_d$ we denote an integer interval labeled by an integer d. We write that $[i_1, j_1]_{d_1}$ and $[i_2, j_2]_{d_2}$ *intersect* if they share a common element, and that $[i_1, j_1]_{d_1}$ *contains* $[i_2, j_2]_{d_2}$ if $i_1 \leq i_2$ and $j_1 \geq j_2$. Note that the labels of the intervals play no role in the intersection or containment relationship. Given a collection $I = \{[i_1, j_1]_{d_1}, \ldots, [i_r, j_r]_{d_r}\}$ of r labeled intervals, we define the union of I as the set

$$U(I) = \{e_d : e \in [i, j]_d \in I \textbf{ and } \nexists e \in [i', j']_{d'} \in I : d' > d\}.$$

Note that $U(I)$ can be represented more compactly as a set of labeled intervals; e.g., we can represent $U(I) = \{1_3, 2_1, 3_1, 4_1, 6_1, 7_1, 8_2\}$ by the set $\{[1, 1]_3, [2, 4]_1, [6, 7]_1, [8, 8]_2\}$. We call this the *compact representation* of $U(I)$. Let us now define the following auxiliary abstract problem.

UNION OF LABELED INTERVALS ON A TREE (ULIT)
Input: A rooted tree T of size N with K leaf nodes. Every non-root node u of T is labeled with an interval $I_u = [i, j]_{d_u}$, where $i, j \in [K]$ and $d_u \in [N]$. For any two labeled intervals I_u and I_v, such that u is an ancestor of v, we have that $d_v > d_u$ and either I_u contains I_v or I_u and I_v do not intersect.
Output: For each leaf node w of T, return the union U_w of labeled intervals from w to the root of T.

We denote the output of ULIT for all K leaf nodes by OUTPUT_K.

The first key observation is that every state of the AC machine (except the start state) has a single failure transition. This, together with the fact that the depth of the states on a failure transition path is monotonically decreasing, lead naturally to the following definition, which forms the basis of our reduction.

Definition 1 *(Failure transition tree (FTtree)).* Given a set R of strings, the *failure transition tree* (FTtree, for short) of R is the rooted tree induced by the set of (reversed) failure transitions of the AC machine of R.

We prove the following lemma.

Lemma 2. *Any instance of the ℓ-APSP problem can be reduced to some instance of the ULIT problem in $\mathcal{O}(n)$ time.*

Proof. We first construct the AC machine of $R = \{S_1, \ldots, S_k\}$. We then obtain the FTtree of R, which we prune by excluding nodes of string depth smaller than ℓ. For conceptual convenience, we also prune leaf-to-root branchless paths, for leaf nodes that do not correspond to terminal states. Finally, using a DFS traversal, we decorate each non-root node of string depth d of the FTtree with the interval $[i, j]_d$, if and only if $A[i], A[i+1], \ldots, A[j]$ are all strings that share a prefix of length d. We denote the residual tree by T_ℓ.

We now prove that T_ℓ is a valid input tree to ULIT. The fact that $d_u < d_v$, when u is an ancestor of v, follows by the Aho-Corasick lemma. Consider the case when node u at string depth d_u is an ancestor of v at string depth d_v and the intervals of u and v share some element s. By construction, all elements in the interval I_u of u share a common prefix of length d_u and all elements of the interval I_v of v share a common prefix of length $d_v > d_u$. Since the intersection of the two intervals is non-empty, and all elements in I_v have a common prefix of length d_v with s, by transitivity, the union of elements of I_u and I_v must have a common prefix of length $d_u < d_v$. Thus, when u is an ancestor of v, I_u must contain I_v.

By the Aho-Corasick lemma we know that the failure transition path from a terminal state to the root encodes all suffix/prefix overlaps between the string corresponding to the terminal state and all other strings in R. Thus, in order to maintain the maximal suffix/prefix overlaps for a terminal state, we need to have the union of the collection of labeled intervals decorating every node that lies on such a path. This is precisely the output of ULIT: every element

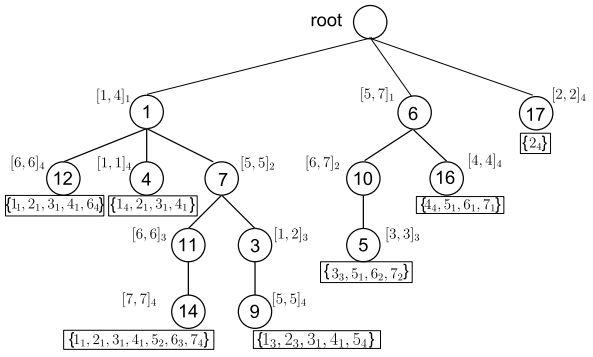

Fig. 2. FTtree T_1 for $\ell = 1$. The output set per leaf node is in a squared box.

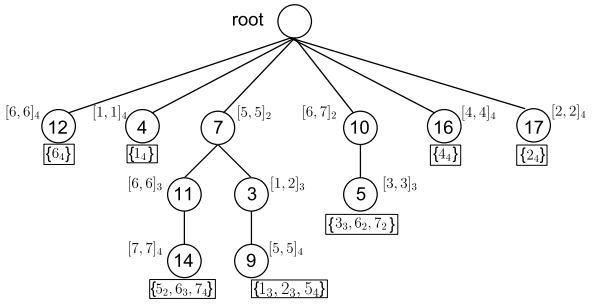

Fig. 3. FTtree T_2 for $\ell = 2$. The output set per leaf node is in a squared box.

$i_d \in U_w$ output by ULIT with T_ℓ as input tree is in one-to-one correspondence with the maximal suffix/prefix overlap of length d between the string represented by w and the string represented by $A[i]$.

The AC machine can be constructed in $\mathcal{O}(n)$ time. The FTtree can be constructed from the AC machine in $\mathcal{O}(n)$ time and decorated with the labeled intervals in $\mathcal{O}(n)$ time. This completes the proof. □

Remark 1. Let us remark that the reduction arguments we provide in Lemma 2 resemble the arguments of correctness of Park et al. in [13]. However, our encoding (labeled intervals) and the ULIT problem are novel as they serve solving a problem which is computationally harder than the problem studied in [13].

Figs. 2 and 3 demonstrate the reduction for $\ell = 1$ and $\ell = 2$, respectively, using the AC machine from Fig. 1. Consider the set $U_9(\{[5, 5]_2, [1, 2]_3, [5, 5]_4\}) = \{1_3, 2_3, 5_4\}$ output at leaf node 9 in Fig. 3. It denotes that string $A[5]$ (`baba`) has a maximal suffix/prefix overlap of length at least $\ell = 2$ with string: $A[1]$ (`abaa`) of length 3; $A[2]$ (`abac`) of length 3; and $A[5]$ (`baba`) of length 4.

3.2. Solving the ULIT problem

A *branching node* of T_ℓ is a non-root node with at least two children. A *branchless subpath* is an upward maximal path of nodes starting from a branching node or a leaf node and ending at the node right before a branching node

Table 1
Steps 1 and 2 using the FTtree in Fig. 2.

Branchless subpaths	Sorted tuples	Compact representation of the union
1	(1,1,4,1)	$\{[1,4]_1\}$
4	(4,1,1,4)	$\{[1,1]_4\}$
5→10	(5,3,3,3) (5,6,7,2)	$\{[3,3]_3,[6,7]_2\}$
6	(6,5,7,1)	$\{[5,7]_1\}$
7	(7,5,5,2)	$\{[5,5]_2\}$
9→3	(9,1,2,3) (9,5,5,4)	$\{[1,2]_3,[5,5]_4\}$
12	(12,6,6,4)	$\{[6,6]_4\}$
14→11	(14,6,6,3) (14,7,7,4)	$\{[6,6]_3,[7,7]_4\}$
16	(16,4,4,4)	$\{[4,4]_4\}$
17	(17,2,2,4)	$\{[2,2]_4\}$

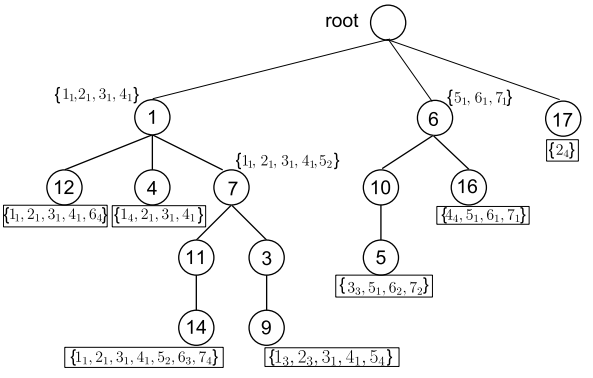

Fig. 4. Step 3 using the FTtree in Fig. 2. At each branching node, we store the resulting union. At each leaf node, we store the output in a squared box.

(or the root node if no branching node exists). For example, the branchless subpaths in Fig. 2 are shown in Table 1. The algorithm for solving the ULIT problem works as follows:

1. Decompose T_ℓ into a set of branchless subpaths. Using a DFS traversal on T_ℓ, for each node decorated with $[i,j]_d$ of every branchless subpath starting with node u, construct a tuple (u,i,j,d).
2. Sort all tuples constructed in Step 1 together. For each branchless subpath, take the compact representation of the union of its set of labeled intervals, which are now sorted. Inspect Table 1.
3. Visit the *branching* and *leaf nodes* using a BFS traversal on T_ℓ. From node u to node v, such that v is the child of u, take the union of the two sets of labeled intervals (one from u and one from v), and update the union associated with v. At the leaf nodes we have the output sets. Inspect Fig. 4.

The following lemma together with Lemma 2 implies Theorem 1.

Lemma 3. *The ULIT problem can be solved in* $\mathcal{O}(N + |OUTPUT_K|)$ *time using* $\mathcal{O}(N)$ *space.*

Proof. That the presented algorithm is correct is trivial. We next analyze the time complexity. Step 1 can be done in $\mathcal{O}(N)$ time because T_ℓ is of size N and thus we have no more than N tuples to create. Step 2 can be done in $\mathcal{O}(N)$ time. Sorting can be done in $\mathcal{O}(N)$ time using radix sort. By definition, for any two labeled intervals I_u and I_v, such that u is an ancestor of v, we have that $d_v > d_u$ and either I_u contains I_v or I_u and I_v do not intersect. Thus taking the compact representation of the union of the set of labeled intervals can be done in $\mathcal{O}(N)$ time because the labeled intervals are sorted and their total number is in $\mathcal{O}(N)$, since we have one interval per node of the branchless subpath. A stack is maintained to obtain the largest label per element (or interval). Note, the compact representation of the union of any two labeled intervals is of size at most three. Step 3 can be done in $\mathcal{O}(N + |\mathsf{OUTPUT}_K|)$ time: taking the union of the two sets of intervals can be done in time linear in their total size as the sets are sorted. The unions of labeled intervals, which are stored at branching nodes, correspond to leaf nodes and so their total size is in $\mathcal{O}(|\mathsf{OUTPUT}_K|)$. The space complexity of the algorithm described above is upper bounded by the time complexity.

Let us now explain how we can improve the space complexity of this algorithm to $\mathcal{O}(N)$. Instead of BFS in Step 3, we perform DFS (Euler tour) on T_ℓ maintaining the union of labeled intervals associated with the currently *active* node u. This represents the union of labeled intervals from the root to u. Since we perform a DFS traversal, we make a union operation when going downward and a set difference operation when going upward. A vector of K stacks is maintained to obtain the currently largest label per element (or interval) during this process. As an example, consider visiting node 7 from active node $u = 1$ (downward) during DFS. When $u = 1$ is active, its associated union is $\{[1,4]_1\}$ (see Fig. 4). The union stored for branchless subpath 7 is $\{[5,5]_2\}$ (see Table 1). The resulting union of the two is thus $\{[1,4]_1,[5,5]_2\}$. Then, consider visiting node 7 from active node $u = 14$ (upward) during DFS. When $u = 14$ is active, its associated union is $\{[1,4]_1,[5,5]_2,[6,6]_3,[7,7]_4\}$. The union stored for branchless subpath 14 (14→11 in Table 1) is $\{[6,6]_3,[7,7]_4\}$. The resulting difference is thus $\{[1,4]_1,[5,5]_2\}$. The currently active node is always associated with a set of $\mathcal{O}(K)$ elements. Thus the total space used at any moment is bounded by $N + K = \mathcal{O}(N)$. The time complexity increases only by a constant factor since we also perform set difference operations. □

4. Final remarks

All existing optimal algorithms for APSP rely on sorting the suffixes of all strings in R. Constructing the generalized suffix tree [17] or the generalized suffix array [11] of R requires $\Theta(n)$ space by definition, in *any case* and for *any alphabet* Σ, for storing the tree or the array, respectively. The AC machine of R can be constructed in $\mathcal{O}(n)$

time if $|\Sigma| = \mathcal{O}(1)$ or in $\mathcal{O}(n \log |\Sigma|)$ time if $|\Sigma|$ is larger, both *without* resorting to suffix sorting; and it may require space that is *asymptotically smaller* than $\mathcal{O}(n)$ if prefix redundancy is non-negligible or space $\Theta(n)$ in the worst case. As we have shown in Lemma 3, we can solve ULIT in space that is *linear* in the size of the AC machine, and so ℓ-APSP can be solved within the same space complexity. Our work may thus inspire further space improvements on solving ℓ-APSP both in theory and in practice.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors would like to thank Anne Luesink (Vrije Universiteit) for implementing the algorithm underlying Theorem 1. The source code is freely available at https://github.com/ALuesink/APSP_algorithm. Grigorios Loukides is partially supported by the Leverhulme Trust RPG-2019-399 project. Solon P. Pissis is partially supported by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively.

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (1975) 333–340, https://doi.org/10.1145/360825.360855.

[2] B. Cazaux, E. Rivals, Hierarchical overlap graph, Inf. Process. Lett. 155 (2020), https://doi.org/10.1016/j.ipl.2019.105862.

[3] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, 2007.

[4] S. Dori, G.M. Landau, Construction of Aho Corasick automaton in linear time for integer alphabets, Inf. Process. Lett. 98 (2006) 66–72, https://doi.org/10.1016/j.ipl.2005.11.019.

[5] G. Gonnella, S. Kurtz, Readjoiner: a fast and memory efficient string graph-based sequence assembler, BMC Bioinform. 13 (2012) 82, https://doi.org/10.1186/1471-2105-13-82.

[6] D. Gusfield, Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology, Cambridge University Press, 1997.

[7] D. Gusfield, G.M. Landau, B. Schieber, An efficient algorithm for the all pairs suffix-prefix problem, Inf. Process. Lett. 41 (1992) 181–185, https://doi.org/10.1016/0020-0190(92)90176-V.

[8] S. Khan, Optimal construction of hierarchical overlap graphs, in: P. Gawrychowski, T. Starikovskaya (Eds.), 32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5–7, 2021, Wrocław, Poland, in: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 17.

[9] J. Lim, K. Park, A fast algorithm for the all-pairs suffix-prefix problem, Theor. Comput. Sci. 698 (2017) 14–24, https://doi.org/10.1016/j.tcs.2017.07.013.

[10] F.A. Louza, S. Gog, L. Zanotto, G. Araujo, G.P. Telles, Parallel computation for the all-pairs suffix-prefix problem, in: S. Inenaga, K. Sadakane, T. Sakai (Eds.), String Processing and Information Retrieval - Proceedings of the 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18–20, 2016, 2016, pp. 122–132.

[11] U. Manber, E.W. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (1993) 935–948, https://doi.org/10.1137/0222058.

[12] E. Ohlebusch, S. Gog, Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem, Inf. Process. Lett. 110 (2010) 123–128, https://doi.org/10.1016/j.ipl.2009.10.015.

[13] S. Park, S.G. Park, B. Cazaux, K. Park, E. Rivals, A linear time algorithm for constructing hierarchical overlap graphs, in: P. Gawrychowski, T. Starikovskaya (Eds.), 32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5–7, 2021, Wrocław, Poland, in: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 22.

[14] M.H. Rachid, Q. Malluhi, A practical and scalable tool to find overlaps between sequences, BioMed Res. Int. 2015 (2015), https://doi.org/10.1155/2015/905261.

[15] W.H.A. Tustumi, S. Gog, G.P. Telles, F.A. Louza, An improved algorithm for the all-pairs suffix-prefix problem, J. Discret. Algorithms 37 (2016) 34–43, https://doi.org/10.1016/j.jda.2016.04.002.

[16] E. Ukkonen, A linear-time algorithm for finding approximate shortest common superstrings, Algorithmica 5 (1990) 313–323, https://doi.org/10.1007/BF01840391.

[17] P. Weiner, Linear pattern matching algorithms, in: 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15–17, 1973, IEEE Computer Society, 1973, pp. 1–11.