Journal Title Here, 2019, pp. 1–8

doi: DOI HERE
Advance Access Publication Date: Day Month Year
Paper

PAPER

Linear indexing for all strings under all internal nodes in suffix trees

Anas Al-okaily[1,*] and Abdelghani Tbakhi[2]

[1]Department of Cell Therapy and Applied Genomics, King Hussein Cancer Center, Amman, Jordan and [2]Department of Pathology and Molecular Medicine, McMaster University, Ontario, Canada
[*]Corresponding author. AA.12682@khcc.jo
FOR PUBLISHER ONLY Received on Date Month Year; revised on Date Month Year; accepted on Date Month Year

Abstract

Suffix trees are fundamental data structure in stringology. In this work, we introduce two algorithms that index all strings/suffixes under all internal nodes in suffix tree in linear time and space. These indexes can contribute in resolving several strings problems such as DNA sequence analysis and approximate pattern matching problems.

Key words: Reads-to-Genome Alignment, Motif search, Suffix Tree

Introduction

Numerous number of string problems are presented in several scientific fields including biological and medical fields. These problems include exact and approximate pattern matching, motif search, lowest common ancestor, and finding tandem repeats. The inputs for these problems could be small documents, databases, DNA data, or corporational big data. In order to resolve string problems more efficiently, several data structures were designed and are used such as mainly suffix trees [1, 2, 3], suffix arrays [4], and FM-index [5].

Building suffix trees, suffix arrays, and FM indexes needs linear time and space. Given the fact that building suffix trees needs more constant factor compared to suffix arrays and FM-indexes, the design of suffix tree structure is more flexible and dynamic. This is attributed to the fact that the structure of suffix trees allows us to notice many systematic redundancies among the strings in the input data when compared to the structure of suffix arrays and FM indexes. One can easily spot that the set of suffixes under an internal node in the suffix tree for some input data can be equal set, subset, or superset of the set of suffixes under many other internal nodes. This observation is not possible using the structure of suffix arrays and FM indexes. As a result and in order to track and simplify the redundancies in the input data, suffix trees would be exclusively the optimal and most efficient choice. Once these redundancies got tracked and simplified in suffix trees, resolving complex string problems will be easier and more efficient than resoling them using suffix arrays, FM indexes, or even suffix trees in their pure form).

In this work, we introduce two algorithms that index all strings/suffixes under all internal nodes in suffix trees in linear time and space.

Methods

Let T be a text of length n derived from alphabet of size Σ and T_i is the suffix at position i in T. Let ST is the suffix tree of T, $Depth(n)$ is the depth of node n i.e. sum of lengths of all edges between root node of ST and node n, and $Suffix_index(l)$ is the suffix index recorded at leaf node l.

Definition 1: Let S be the set of leaf nodes *under* an internal node x in ST, then suffixes under node x, denoted as $SU(x)$, are the suffixes labeled between node x and each leaf node in S i.e. SU(x) = $\{T_{Depth(x)+Suffix_Index(l)}\ \forall\ l \in S.\}$.

As an example, suffixes *under* node 20, SU(node 20), in Figure 1 are: "CTAAG$", "G$", and "TTTAACTAAG$" (which are suffixes with indexes 13, 17, and 7 respectively). Now, note that when node a has a suffix-link to node b, then all the suffixes under node ($SU(a)$) must be a subset or equal to $SU(b)$. This indicates that if we assign index values to the suffixes in $SU(a)$, these index values can implicitly be used/applied to index the same suffixes in $SU(b)$. Therefore, when indexing the suffixes under node b, only the set of $SU(b) - SU(a)$ suffixes will be needed to be indexed. This process can be applied recursively so that if node b has a

© The Author 2019. Published by Oxford University Press. All rights reserved. For permissions, please e-mail: journals.permissions@oup.com

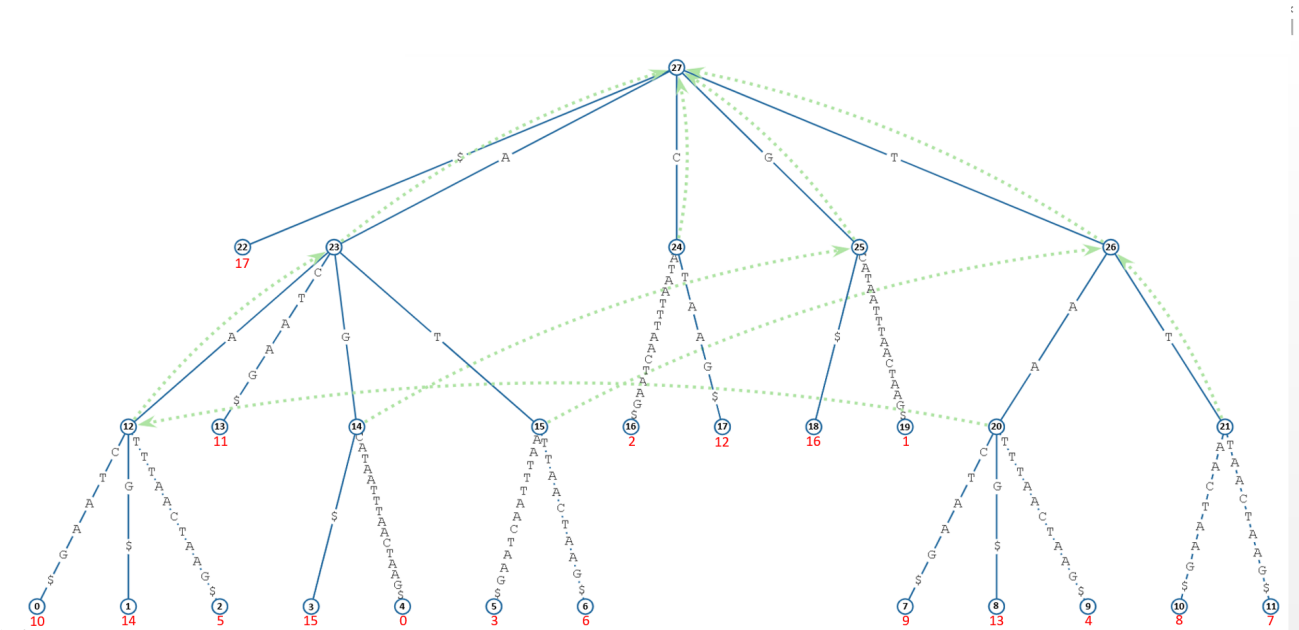

Fig. 1. Suffix tree of string AGCCTAATTTAACTAAG$ (drawn at https://hwv.dk/st/?AGCATAATTTAACTAAG$). Suffix indexes start by 1 (not 0)

suffix-link to node c, then we will need to index only suffixes $SU(b) - SU(b)$ and nothing will be needed for the set of suffixes $SU(c) - SU(a)$ as they have already been indexed. Moreover, as each internal node in the structure of ST, say node x, may have up to $O(\Sigma)$ nodes with suffix-link to node x and in order to build the above indexing schema, we should start by indexing all suffixes under each node with suffix-link to node x before indexing node x itself (which indicates a postorder traversal process). The need to compute this postorder traversal process urged designing and building the following tree structure name as $OSHR$ tree.

OSHR tree

Given ST, the structure of $OSHR$ tree is as the following:

- Root node is the root of ST.
- Leaf nodes are all internal nodes in ST with no incoming suffix-links. Node 14 from Figure 1 as an example.
- Internal nodes are all the internal nodes in ST with at least one incoming suffix-link. Node 25 from Figure 1 as an example.
- Internal nodes are all internal nodes in ST with at least one incoming suffix-link (node 25 from Figure 1 as an example).
- Each internal node, let's say node x, must has a list denoted hereafter as $nodes_suffix_linked_to_me$ to store the nodes suffix-linked to node x (the upper bound of length of this list at any internal node is $O(\Sigma)$). As an example, the list $nodes_suffix_linked_to_me$ of node 26 in Figure 1 must contain nodes 15 and 21.
- There is a directed edge from node a to node b if node b has a suffix-link to node a.
- Edges have no labels.
- Leaf nodes in ST are not included in the structure of $OSHR$ tree.

The directed edges in $OSHR$ tree which are the opposite of suffix-links are actually a simple form of Weiner-links in ST (Weiner-link as defined at [6, 7, 8]). Due to the construction properties of ST and suffix-links, $OSHR$ tree will be a directed acyclic graph. Clearly, the space and time costs for building $OSHR$ tree are linear $O(\Sigma n)$ and can be constructed implicitly (inside ST tree) or explicitly (outside ST). The structure of $OSHR$ tree is different than suffix-tour-graph [9] and suffix-link-tree [9, 7, 8]. Unlike suffix-tour-graph, $OSHR$ structure is acyclic graph. Compared to suffix-links tree structure, the edges in $OSHR$ tree have no labels, does not include the leaf nodes of ST, and the leaf nodes in $OSHR$ tree are internal nodes in ST with no incoming suffix links. What the acronym "$OSHR$" stand for is provided in the Acknowledgment section.

OT indexing

In order to track all similarities and redundancies among the strings/suffixes under different internal nodes in ST, the following traversal must be performed.

Definition 2: Indexing (processing) suffixes/strings under nodes (based on ST structure) that have suffix-links to node (x) and not re-indexing (re-processing) the same suffixes/strings under node x throughout a postorder traversal of $OSHR$ tree is referred hereafter as OT indexing (processing, the reason for the OT acronym is provided in the Acknowledgment section).

A simple example, let node 26 in Figure 1 is the current visited node in postorder traversal of $OSHR$ tree, then node 15 and 21 must have been visited already where the following have been already processed at each of them. Suffixes under node 15 which are suffix index 5 ("AATTTAACTAAG$") and suffix index 8 ("TTAACTAAG$") must have already been indexed/processed. Under node 21, suffixes with indexes 10 ("AACTAAG$") and 9 ("TAACTAAG$") must have been indexed. Now (at node 26), only suffix 16 ("AAG$") will need to be indexed. This way, all suffixes under node 26 could be indexed without an explicit indexing (processing) all of them. Once postorder traversal of $OSHR$ tree reaches the root node, all suffixes under each internal node (including root node) must have been indexed.

Algorithm 1 Non-Trivial algorithm for Finding Base Suffixes

1: Build *OSHR* tree.
2: Create and initialize a hash table, H, that will be used to store the newly found base suffix (by hashing the integer values of base suffixes).
3: **loop** through a postorder traversal of *OSHR* tree and at each visited internal node v
4: Create and initialize an attribute with name $List_of_base_suffixes$ of type list at node v where base suffixes of node v will be stored in.
5: **for** $suffix_index_of_leaf_node$ in $S[v.index_of_leftmost_leaf_in_S T : v.index_in_S_of_suffix_index_of_rightmost_leaf_node{+}1]$ **do**
6: **if** $suffix_index_of_leaf_node + Depth(v)$ not in H **then**
7: Add $suffix_index_of_leaf_node + Depth(v)$ into $v.List_of_base_suffixes$
8: Add $suffix_index_of_leaf_node + Depth(v)$ into H
9:
10: ▷ *Line 2, an array of length n where each element in the array is initialized to -1 can be used accordingly instead of the hash table.* ◁
11: ▷ *Line 5, the for loop will go through suffix indexes of all leaf nodes under node v using list S and values of index_in_S_of_suffix_index_of_leftmost_leaf_node and index_in_S_of_suffix_index_of_rightmost_leaf_node of node v.* ◁

Algorithm 2 Non-Trivial algorithm for Finding Base Suffixes

1: Build *OSHR* tree.
2: Create a list M of size n that maps suffix indexes of leaf nodes to the memory locations of leaf nodes so that $M[x]$ is the memory location of leaf node with suffix index x.
3: **loop** through a postorder traversal of *OSHR* tree or ST, and at each visited internal node v
4: Create and initialize an attribute with name $List_of_base_suffixes$ of type list at node v where base suffixes of node v will be stored in.
5: **for** $suffix_index_of_leaf_node$ in $S[v.index_of_leftmost_leaf_in_S T : v.index_in_S_of_suffix_index_of_rightmost_leaf_node{+}1]$ **do**
6: $p \leftarrow M[suffix_index_of_leaf_node - 1]$ ▷ *The leaf node of previous suffix index.*
7: $f \leftarrow True$
8: **for** node n in $v.nodes_suffix_linked_to_me$ **do**
9: **if** $n.index_in_S_of_suffix_index_of_leftmost_leaf_node \leq p.index_in_S \leq n.index_in_S_of_suffix_index_of_rightmost_leaf_node$ **then**
10: $f \leftarrow False$
11: break
12: **if** f **then**
13: add $suffix_index_of_leaf_node + Depth(v)$ to $v.List_of_base_suffixes$.
14:
15: ▷ *Line 5, the for loop in will go through suffix indexes of all leaf nodes under node v using list S and values of index_in_S_of_suffix_index_of_leftmost_leaf_node and index_in_S_of_suffix_index_of_rightmost_leaf_node of node v.* ◁
16: ▷ *Lines 8-9, the cost of these lines is $O(\Sigma)$. However, if list nodes_suffix_linked_to_me at each internal node is sorted based on index_in_S_of_suffix_index_of_leftmost_leaf_node and index_in_S_of_suffix_index_of_rightmost_leaf_node values, then the check in these lines can be computed in $O(log_2\Sigma)$ time by applying binary search (which is what is coded in the implementation of the algorithm).* ◁
17: ▷ *Line 9, the condition of the if statement checks if the leaf node p is a descendant node for the internal node n by comparing the values created using list S (index_in_S of leaf nodes and index_in_S_of_suffix_index_of_leftmost_leaf_node and index_in_S_of_suffix_index_of_rightmost_leaf_node of internal nodes).* ◁

Two types of OT indexing will be introduced in this paper, indexing all suffixes under all internal nodes in ST in linear costs (OT indexing of base suffixes) and indexing all paths between each internal node in ST and each of its descendant internal nodes in linear costs (OT indexing of base paths).

Base suffixes

Initially, base suffixes will be defined and later we will describe how to find base suffixes under each internal nodes in ST in linear time and space.

Definition 3: base suffixes are suffixes that are presented under a node in ST, let's say node x, and never have been presented under any of the nodes suffix-linked to node x. In case the node is an *OSHR* leaf node, then all suffixes under this node are base suffixes.

The following are examples from Figure 1. The base suffix under node 26 is suffix "AAC$", the base suffixes for node 23 are "TAATTTAACTAAG$", "GCATAATTTAACTAAG$", "ATTTAACTAAG$", "AG$", and "ACTAAG$". As node 20 has no incoming suffix-links, then all suffixes under it are base suffixes. Base suffixes for node 12 are none.

The total number of base suffixes under all internal nodes in ST is exactly equal to n (as demonstrated and proved in the implementations). The term "base" is selected to indicate that this is the first appearance of this suffix under any internal node.

Definition 4: If s is a base suffix under node x, then *extent suffixes* of base suffix s are each suffix under each node that is ancestor to node x based on *OSHR* tree structure (throughout suffix-links) where each of these suffixes is equal to suffix s.

For example, the suffix "TTAACTAAG$" which ends at leaf node 6 is a base suffix under node 15, the extant suffixes for this base suffix are: suffix "TTAACTAAG$" under node 26 which ends at leaf node 11, and the one under root node which ends at leaf node 9.

The upper bound for the number of extent suffixes for any base suffix is $O(h)$ where h is the height of the ST. The last extent suffix of any base suffix is the one under the root node.

Algorithm 3 Linear algorithm for Finding Base Suffixes

```
Ensure: Each internal node in ST has an attribute of type list named as List_of_base_suffixes. This list will be used to store the base
        suffixes under the internal node.
Create list M as described in algorithm 2.
loop through ST, and each visited node v
    if node v is a leaf node then
        ▷ The following code computes base suffixes derived from reference leaf nodes ◁
        if Suffix_index(v) + 1 < number_leaf_nodes then
            leaf_node_of_next_suffix_index ← M[Suffix_index(v) + 1]
            if leaf_node_of_next_suffix_index.parent ≠ v.parent.suffix_link then
                temp ← leaf_node_of_next_suffix_index.parent
                end_node ← v.parent.suffix_link
                while True do
                    if temp = end_node then
                        if v.parent = ST.root then
                            temp.List_of_base_suffixes.append(Suffix_index(v) + 1 + Depth(temp)
                        break
                    else
                        temp.List_of_base_suffixes.append(Suffix_index(v) + 1 + Depth(temp))
                        temp ← temp.parent
    else
        ▷ Then node v is an internal node and the following code computes base suffixes derived from reference internal nodes ◁
        if v.suffix_link.parent ≠ v.parent.suffix_link then
            top_node ← v.parent.suffix_link
            bottom_node ← v.suffix_link
            temp = bottom_node.parent
            while temp ≠ top_node do
                for suffix_index_of_leaf_node in S[v.index_in_S_of_suffix_index_of_leftmost_leaf_node :
                v.index_in_S_of_suffix_index_of_rightmost_leaf_node + 1] do
                    temp.List_of_base_suffixes.append(suffix_index_of_leaf_node + 1 + Depth(temp)
                    temp ← temp.parent

▷ This procedure for root node only ◁
for each child node c of root node do
    if node c is a leaf node then
        if Suffix_index(c) + 1 < number_leaf_nodes then
            if (M[suffix_index(c) + 1]).parent = ST.root then
                ST.root.List_of_base_suffixes.append(suffix_index(c) + 1)
    else
        if c.suffix_link ≠ ST.root then
            for suffix_index_of_leaf_node in [c.index_in_S_of_suffix_index_of_leftmost_leaf_node :
            c.index_in_S_of_suffix_index_of_rightmost_leaf_node + 1] do
                ST.root.List_of_base_suffixes.append(suffix_index_of_leaf_node + 1)

▷ This procedure for the nodes in the path of T0 suffix ◁
temp ← M[0]
while temp ≠ ST.root do
    temp ← temp.parent
    temp.List_of_base_suffixes.append(Depth(temp))

▷ Node c.suffix_link is the node suffix-linked from node c. Node c.parent is the parent node of node c. ◁
```

This way, once a base suffix is processed/indexed, this processing/indexing can be applied *implicitly* to all $O(h)$ extent suffixes. Hence, OT indexing/processing all suffixes under all internal nodes in ST can be computed with a factor of n.

Finding base suffixes

Base suffixes under each internal nodes can be computed throughout a postorder traversal of $OSHR$ tree or ST tree where at each visited internal node, v, check whether each suffix under node v presented or not under any of the Σ nodes suffix-linked to node v. If not, then this suffix is a base suffix for node v. The cost of this process will be $O(n)$ space (for storing the base suffixes at each internal node) and $O(nhc)$ time where nh is the sum of looping through all suffixes under all internal nodes in ST and c is the time cost for the checking procedure. Trivially, the checking can be performed by looping through each suffix under each of the Σ nodes suffix-linked to node v, however this will be very costly ($O(n^2h)$). We will now describe non-trivial algorithms with $O(nh)$, non-trivial algorithms with $O(nhlog_2\Sigma)$ time costs, and then describe a linear algorithm.

Before of all, ST must be already built. In addition, the following list named as S must be built which will be used in the three algorithms. Traverse ST and append the suffix indexes of leaf nodes from left to right into list S (the first element of S is the suffix index of leftmost leaf node and the last element is the suffix index of rightmost leaf node). Meanwhile, create an attribute named as $index_in_S$ at leaf node where its value is the index of leaf node in list S, and create recursively two attributes at each internal node: $index_in_S_of_suffix_index_of_leftmost_leaf_node$ and

Algorithm 4 Non-Trivial algorithm for Finding Base Paths

```
1: Build OSHR tree.
2: loop through postorder traversal of OSHR tree and at each visited internal node v (node v is considered here the top base node of the
   base path):
3:     Add an attribute List_of_bottom_base_nodes of type list to node v where bottom base nodes of the found base paths will be stored.
4:     if node v is an OSHR leaf node then
5:         ▷ All paths between node v and its descendants internal node are base paths, so, ◁
6:         for each internal node t under node v do
7:             v.List_of_bottom_base_nodes.append(t)
8:     else
9:         for each internal node t under node v do:
10:            f ← True
11:            for node k in v.nodes_suffix_linked_to_me do:
12:                for node p in t.nodes_suffix_linked_to_me do
13:                    if p.index_in_S_of_suffix_index_of_leftmost_leaf_node ≤ k.index_in_S_of_suffix_index_of_leftmost_leaf_node ≤
                       k.index_in_S_of_suffix_index_of_rightmost_leaf_node ≤ p.index_in_S_of_suffix_index_of_leftmost_leaf_node
                       then.
14:                        f ← False
15:                        Break
16:            if f then
17:                ▷ The path between node v and node t (bottom base node) is a base path, so, ◁
18:                v.List_of_bottom_base_nodes.append(t)
19:
20: ▷ Line 4, a node v is an OSHR leaf node if node v is an internal node in ST with no incoming suffix_links (list v.nodes_suffix_linked_to_me
    is empty), while it's an OSHR internal node if node v is an internal node in ST with at least one incoming suffix_link (list
    v.nodes_suffix_linked_to_me is unempty). ◁
21: ▷ Line 11-13, the cost of these lines is O(Σ^2). However, if list nodes_suffix_linked_to_me at each internal node is sorted based
    on index_in_S_of_suffix_index_of_leftmost_leaf_node and index_in_S_of_suffix_index_of_rightmost_leaf_node values, then the
    check in these lines can be computed in O(Σlog_2Σ) time by applying binary search (which is what is coded in the implementation of the
    algorithm). ◁
22: ▷ Line 13, the condition of the if statement checks if the internal node k is a descendant node for the internal
    node p by comparing the values created using list S (index_in_S_of_suffix_index_of_leftmost_leaf_node and
    index_in_S_of_suffix_index_of_rightmost_leaf_node of internal nodes) of both nodes. ◁
```

$index_in_S_of_suffix_index_of_leftmost_leaf_node$ where the value of these attributes are the index in S of leftmost and rightmost leaf nodes under the internal node, respectively.

The first non-trivial algorithm will cost $O(nh)$ with auxiliary $O(n)$ space (hash table) as shown in Algorithm 1. The second non-trivial algorithm will cost $O(nhlog_2\Sigma)$ with auxiliary $O(n)$ space as shown in Algorithm 2.

The linear algorithm motivated from the fact that the total number of base suffixes for all internal nodes in ST is n, then finding each base suffix in constant time will cost $O(n)$ time. Before stating this algorithm, we need to define the followings.

Definition 5: Let A be a leaf node in ST with suffix index x, B is the parent node of node A, C is the leaf node with suffix index $x+1$, and D is the parent node of node C. If the node suffix-linked from node B is not node D, then denote node A as a *reference leaf node* and denote each node between node C and node D as an *inbetween* node.

As an example from Figure 1, node 6 is reference leaf node for node 21 and node 21 is an inbetween node for node 6. Note that a reference leaf node can have more than one inbetween node and an inbetween node can have up to $O(\Sigma)$ reference leaf nodes. In addition, the total number of reference leaf nodes for all internal nodes in ST cannot be more than n.

Definition 6: Let A be an internal node in ST, B is the parent node of node A, C is the suffix-link node of node A, and D is the parent node of node C. If the node suffix-linked from node B is not node D, then denote node A as a *reference internal node* and denote each node between node C and node D as an *inbetween* node.

As an example from Figure 1, node 20 is reference internal node and node 23 is an inbetween node for node 20. Note that a reference internal node can have more than one inbetween node and an inbetween node can have up to $O(\Sigma)$ reference internal nodes. In addition, the total number of reference internal nodes for all internal nodes in ST is much less than n. Moreover, an inbetween node can be derived from reference leaf node (or more) and reference internal node (or more).

Now, the linear algorithm can be computed with or without explicit finding and recording of inbetween nodes and their reference internal nodes and/or reference leaf nodes. The main form of this algorithm is without explicit finding and recording of inbetween nodes. Algorithm 3 states this algorithm. The algorithm that explicitly create and record inbetween nodes and their reference leaf nodes and reference internal nodes is provided in the code provided in this work.

As the upper bound of the number of reference leaf nodes and the upper bound of the number of reference internal nodes is Σ for any internal node (given that most of internal nodes will not be inbetween nodes in the first place), the cost for finding inbetween nodes will be $O(\Sigma n)$. For the computing base suffixes, the cost will be $O(n)$ as the total number of base suffixes under all internal nodes in ST is equal to n.

Theorem 1: Finding all base suffixes under all internal nodes in ST can be computed in linear time and space ($O(n)$).

Consequently, once base suffixes are computed and found under each internal nodes in ST and throughout OT indexing/processing, any index/process P with cost p for a base suffix s under an internal node will implicitly be applied to all $O(h)$ extent suffixes of s. Hence, the total cost of applying process P for all suffixes under all internal nodes in ST will be to the factor of n and will cost $O(pn)$ instead of (phn).

Base paths

The motivations behind this index are the following observations. Firstly, the main complexity in a tree structure is due to the branching caused by internal nodes. Secondly, the tails of suffixes (label between leaf node and its parent node) are usually of very long length, hence processing them are costly. Thirdly, assuming a process reaches an internal node where all its children are leaf nodes, then the cost for performing some computations on these leaf nodes will be bounded to Σ factor. Therefore, there is no need to always perform any indexing/processing for the tails of the suffixes and it will be sufficiently efficient for many applications to perform indexing/processing only to the labels/paths between each internal node and its descendant *internal* nodes.

Definition 7: A path between two internal nodes, let's say nodes a and b, where this path was never appeared between any other two internal nodes throughout suffix-links to node a and to node b are referred hereafter as *base path*. This is induced by having no incoming suffix-link to node a, b, both, or there are incoming suffix-links to node a and to node b but the label between any node with suffix-link to node a and any node with suffix-link to node b is not equal to the label between node a and node b. Hereafter, node a is referred as *top base node* and node b as *bottom base node* for the base path.

For example, the path in Figure 1 (given that the input text for the ST in Figure 1 is very small) between node 23 and node 14 is a base path as it was never appeared between any other two internal nodes. Paths between nodes 26 and 20 and path between nodes 23 and 12 are also base paths.

Definition 8: If p is base path between top base node a and bottom base node b, then the path between node a' (node suffix-linked from node a) and node b' (node suffix-linked from node b) is an *extent path* for the base path p. This applies between all ancestor nodes (under $OSHR$ tree structure and throughout suffix-links) of node a and node b. The last extent path for base path p, is the one where the top base node is the root node of ST.

As an example, the path between the root node and node 25 is an extent path for the base path between nodes 23 and 14.

Note that the set of unique labels of all base paths *must* eventually equal exactly to the set of labels of paths between the root node and all internal nodes in ST. This must be true, as proved also by implementation, since a path between the root node and a descendant internal nodes is the last appearance (extent) path of some base path/s throughout suffix-links in ST. As any path between the root node and internal node can have up to $O(\Sigma)$ paths from other pair of nodes, the total number of base paths can't be more than $O(\Sigma n)$.

Finding base paths

All base paths in ST can be found using a trivial algorithm with time cost of $O(hn^2)$ as follows. Perform postorder traversal of OSHR tree and at each visited node v: check if the label of each path between node v and a descendant internal node of v is presented between any node (with suffix-link to node v) and one of its descendant internal node; if not, then this path is a base path. However, in this work we will introduce a non-trivial algorithm that costs $O(nh\Sigma log_2\Sigma)$ time and a linear algorithm ($O(\Sigma n)$ time and space).

The first non-trivial algorithm for finding base suffixes which uses hash table can't be applied here to find base paths as base paths are not totally unique like base suffixes (total number of base suffixes are exactly n where for base paths they can be up to Σn). For instance, two sibling nodes under $OSHR$ tree structure (both nodes must suffix-link to the same node), can have base path under each of them with the same label. So if the base path under the first node is found and hashed, then the base path under the second node will not be detected and found as it was found/hashed already. However, Algorithm 4 states non-trivial algorithm which is similar to the second non-trivial algorithm for finding base suffixes can compute base paths with $O(nh\Sigma log_2\Sigma)$ time and $O(\Sigma n)$ space.

As the total number of base paths is no more than Σn, if the time cost for finding each base path is constant, then finding all base paths will cost $O(\Sigma n)$ time and space. Certainly this is achievable using Algorithm 5.

Theorem 2: Finding all base paths under each internal nodes in ST can be computed in linear time and space ($O(\Sigma n)$).

Hereby, once base paths are computed and found under each internal node in ST, throughout OT processing any processing P with cost p for a base path t under an internal node will implicitly be applied to the $O(h)$ extent paths of t under the other internal nodes. Hence, the total cost of applying process P for all paths under all internal nodes in ST will be to the factor of n and will cost $O(np)$ instead of (phn).

Results

In order to test the proposed algorithms, we used the genomes: WS1 bacterium JGI 0000059-K21 (Bacteria), Astrammina rara (Protist), Nosema ceranae (Fungus), Cryptosporidium parvumIowa II (Protist), Spironucleus salmonicida (Protist), Tieghemostelium lacteum (Protist), Fusarium graminearumPH-1 (Fungus), Salpingoeca rosetta (Protist), and Chondrus crispus (Algae).

As a prepossessing step, header lines and newlines were removed from the fasta file and any small letter nucleotide were capitalized. This generates a one line genome file for each fasta file where all nucleotide are capital. The python script used for this preprocessing step is provided at `https://github.com/aalokaily/Finding_base_suffixes_and_base_paths`.

All five algorithms stated in this work were implemented using Python language and are available at `https://github.com/aalokaily/Finding_base_suffixes_and_base_paths`. For base suffixes finding, the outcomes of the non-trivial algorithm (algorithm 1) and linear algorithm (algorithm 3) were reported in order to validate the finding of base suffixes and compare their time costs. As the non-trivial algorithm (algorithm 2) is theoretically and as was found practically (data not shown) slower than algorithm 1, we ignore its comparisons. For base paths finding, the outcomes of non-trivial algorithm (algorithm 4) and linear algorithm (algorithm 5) were reported in order to validate the finding of base paths and compare their time costs. All results are shown in Table 1.

First of all, all base suffixes and base paths found under each internal nodes in ST using the linear algorithms (algorithms 3 and 5) were the same as the ones found using their correspondent non-trivial algorithms (algorithm 1 and 4). Clearly as shown in Table 1, both linear algorithms are faster than the correspondent non-trivial algorithms. In addition, the cost for linear algorithms from small size to large size genomes grows in constant factor as expected while their respected non-trivial algorithms grows larger as they are dependent on a variable factor (the h value). Lastly, the total number of base paths for any genome (small to large) is no more than $O(\Sigma n)$.

Algorithm 5 Linear algorithm for Finding Base Paths

Ensure: Each internal node in ST has an attribute of type list named as $List_of_bottom_base_nodes$. This list will be used to store the bottom base nodes of base paths where the internal node itself is the top base node of the base paths.

1: Build $OSHR$ tree.
2: **loop** through postorder traversal of ST
3: Find and record inbetween nodes and their reference internal nodes.
4: Create list denoted as $List_of_OSHR_leaf_nodes_from_left_to_right$ which stores $OSHR$ leaf nodes from left to right with recording at each internal node two values $index_of_leftmost_OSHR_leaf$ and $index_of_rightmost_OSHR_leaf$ (these values are the correspondent indexes accordingly in $OSHR_leaf_nodes_left_to_right_list$ list). ▷
5: **loop** through postorder traversal of ST or $OSHR$ tree, and at each visited node v (node v is considered here as the top base node of base path)
6: **if** node v is an $OSHR$ leaf node **then**
7: ▷ *All paths between node v and its descendants internal node are base paths, so,* ◁
8: **for** each internal node t under node v **do**
9: $v.List_of_bottom_base_nodes.append(t)$
10: **else**
11: ▷ *The path between node v and each descendent internal node under node v that is an OSHR leaf node is a base path. So, add each of them into $v.List_of_bottom_base_nodes$ list. The set of OSHR leaf nodes under node v can be retrieved using $List_of_OSHR_leaf_nodes_from_left_to_right$ and $index_of_leftmost_OSHR_leaf$ and $index_of_rightmost_OSHR_leaf$ values of node v without the need to traverse all internal nodes under node v, so,* ◁
12: **for** OSHR_leaf_node in $OSHR_leaf_nodes_left_to_right_list[v.index_of_leftmost_OSHR_leaf : v.index_of_rightmost_OSHR_leaf + 1]$ **do**
13: $v.List_of_bottom_base_nodes.append(OSHR_leaf_node)$
14: **if** node v is an inbetween node and has reference internal node/s **then**
15: Create and initialize a hash table R.
16: **for** each reference-internal-node r of node v, if any **do**
17: **for** each internal node t under node r **do**
18: $R.append(t.suffix_link)$
19: ▷ *As some node can be suffix-linked by more than one reference internal nodes (or their descendant internal nodes), list R may contain duplicates so remove them, so* ◁
20: Remove duplicates by sorting or hashing R
21: ▷ *Now, as the path between node v and a node n in R may be covered already by some base path (hence not consider this path as a base path), we need to check that as follows* ◁
22: **for** each node n in R **do**
23: $f \leftarrow True$
24: **for** node p in $v.nodes_suffix_linked_to_me$ **do**
25: **if** $p.index_in_S_of_suffix_index_of_leftmost_leaf_node \leq n.index_in_S_of_suffix_index_of_leftmost_leaf_node \leq n.index_in_S_of_suffix_index_of_rightmost_leaf_node \leq p.index_in_S_of_suffix_index_of_leftmost_leaf_node$ **then**
26: $f \leftarrow False$
27: Break
28: **if** f **then**
29: ▷ *The path between node v and node n (bottom base node) is a base path, so,* ◁
30: $v.List_of_bottom_base_nodes.append(n)$
31:
32: ▷ *Line 4 & 6, a node v is an OSHR leaf node if node v is an internal node in ST with no incoming suffix_links (list $v.nodes_suffix_linked_to_me$ is empty), while it's an OSHR internal node if node v is an internal node in ST with at least one incoming suffix_link (list $v.nodes_suffix_linked_to_me$ is unempty).* ◁
33: ▷ *Line 18, node $t.suffix_link$ is the node suffix-linked from node t.* ◁
34: ▷ *Line 22-30, the cost of these lines is $O(|R|\Sigma)$. However, if list $nodes_suffix_linked_to_me$ at each internal node is sorted based on $index_in_S_of_suffix_index_of_leftmost_leaf_node$ and $index_in_S_of_suffix_index_of_rightmost_leaf_node$ values, then the check in these lines can be computed in $O(l|R|og_2\Sigma)$ time by applying binary search (which is what is coded in the implementation of the algorithm).* ◁
35: ▷ *Line 25, the condition of the if statement checks if the internal node n is a descendant node for the internal node p by comparing the values created using list S ($index_in_S_of_suffix_index_of_leftmost_leaf_node$ and $index_in_S_of_suffix_index_of_rightmost_leaf_node$ of internal nodes) of both nodes.* ◁
36: ▷ *Clearly, there is no need to compute base paths for the root node.* ◁

Acknowledgments

The name "OSHR" tree stands for Okaily-Sheehy-Huang-Rajasekaran. These letters stand for the last name of the first author and last names of his PhD committee in the University of Connecticut, Department of Computer Science, as the initial version of the error tree structure was a chapter in the PhD dissertation of the first author. The committee members were: Chun-Hsi Huang (Major Advisor), Sanguthevar Rajasekaran, and Don Sheehy. This naming meant to tribute and appreciate their kind, influential, and professional teaching and supervision. The name OT stands for the last names of the authors of this work.

Declaration of interests

There is NO Competing Interest.

Table 1. Results for testing, validating, and comparing algorithms 1 and 3 for finding base suffixes and algorithms 4 and 5 for finding base paths.

Genome	WS1 bacterium	Astrammina	Nosema	Cryptosporidium	Spironucleus	Tieghemostelium	Fusarium	Salpingoeca	Chondrus
GenBank Accession	GCA000398605.1	GCA000211355.2	GCA000988165.1	GCA000165345.1	GCA000497125.1	GCA001606155.1	GCF000240135.3	GCA000188695.1	GCA000350225.2
No. of alphabets	5	5	4	15	5	4	5	5	5
No. of nuc/leaf nodes	509,552	1,450,096	5,690,749	9,102,325	12,954,589	23,375,663	36,458,047	55,440,310	104,980,421
No. of Internal nodes	328,917	926,087	3,849,880	5,931,181	8,684,514	15,606,221	22,972,065	37,480,799	81,909,252
No. of OSHR leaf nodes	132,774	380,518	1,353,289	2,328,480	3,127,031	5,740,807	9,800,385	11,545,032	15,307,982
No. of OSHR internal nodes	196,143	545,569	2,496,591	3,602,701	5,557,483	9,865,414	13,171,680	25,935,767	66,601,270
Alg 1 (time complexity/sec)	2,409,867/3	6,838,684/9	26,549,891/37	42,989,315/61	60,212,775/92	109,641,901/172	172,751,172/278	250,969,523/446	457,348,511/983
Alg 3 (time complexity/sec)	6,117,025/3	18,036,912/8	83,860,995/41	186,083,328/113	215,252,439/110	364,737,747/209	851,871,768/561	19,667,387,766/8,763	2,787,481,705/1,508
No. of base paths	1,229,296	3,685,339	16,277,962	27,440,361	37,673,155	73,606,066	116,915,929	161,201,632	234,284,261
Alg 4 (time complexity/sec)	3,940,965/6	11,336,761/19	47,652,774/87	76,241,139/140	108,625,373/214	201,220,732/396	307,913,746/647	479,928,226/1,070	882,404,297/3,222
Alg 5 (time complexity/sec)	6,314,996/21	18,734,271/63	99,497,879/328	210,245,283/829	238,310,327/877	430,212,888/1,542	804,854,850/3,766	15,946,058,214/93,295	3,670,449,444/14,452

Data and materials availability

The algorithms proposed in this paper are implemented using python programming language and are available at `https://github.com/aalokaily/Finding_base_suffixes_and_base_paths`.

References

1. Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.
2. Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.
3. Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
4. Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.
5. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.
6. Philip Wellnitz. *Counting patterns in strings and graphs*. PhD thesis, Saarländische Universitäts-und Landesbibliothek, 2021.
7. Alberto Apostolico and Fabio Cunial. Suffix trees and arrays. *Journal: Encyclopedia of Algorithms*, pages 1–10, 2014.
8. Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Transactions on Algorithms (TALG)*, 16(2):1–54, 2020.
9. Tatiana Starikovskaya and Hjalte Wedel Vildhøj. A suffix tree or not a suffix tree? *Journal of Discrete Algorithms*, 32:14–23, 2015.