Linear Approximate Pattern Matching Algorithm

Anas Al-okaily (AA.12682@khcc.jo)
King Hussein Cancer Center
Abdelghani Tbakhi
King Hussein Cancer Center

Research Article

Keywords: pattern matching, approximate pattern matching, reads alignment

Posted Date: October 29th, 2021

DOI: https://doi.org/10.21203/rs.3.rs-1021063/v1

Linear Approximate Pattern Matching Algorithm

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

[1]Department of Cell Therapy & Applied Genomics, King Hussein Cancer Center
Amman, 11941, Jordan.

[*]To whom correspondence should be addressed; E-mail: AA.12682@khcc.jo

Pattern matching is a fundamental process in almost every scientific domain. The problem involves finding the positions of a given pattern (usually of short length) in a reference stream of data (usually of large length). The matching can be as an exact or as an approximate (inexact) matching. Exact matching is to search for the pattern without allowing for mismatches (or insertions and deletions) of one or more characters in the pattern), while approximate matching is the opposite. For exact matching, several data structures that can be built in linear time and space are used and in practice nowadays. For approximate matching, the solutions proposed to solve this matching are non-linear and currently impractical. In this paper, we designed and implemented a structure that can be built in linear time and space and solve the approximate matching problem in ($O(m + \frac{log_{\Sigma}^{k}n}{k!} + occ)$ search costs, where m is the length of the pattern, n is the length of the reference, and k is the number of tolerated mismatches (and insertion and deletions).

Introduction

The applications of pattern matching are tremendous and in practice in minutely basis over the globe. Almost every aspect of our lives involves us to search for data in a reference that is short data (small document or small database) or big data (DNA data, internet webpages, banking data, etc). The input is a text (S) of length n over an alphabet of size Σ where the length of pattern (P) is m and number of allowed errors (mismatch, insertion, or deletion) is k. The output are the starting positions in S of the sub-sequences that are at k Hamming (or edit distance) with P.

Exact matching, in which the k value is zero, is the simple form of pattern matching problem. Many structures solved the exact matching problem in optimal time and space (linear) (*1*). This includes suffix trees (*2–4*), suffix arrays (*5*), and FM-index (*6*). While, approximate pattern matching, where the value of k is one or more, has not been solved in deterministic linear time and space. Due to this and in order to obtain as much as practical solution as possible, solutions for approximate matching problem is solved by using structures that solve the exact matching (the above structures as examples) followed by heuristic techniques. Tools that are solving reads-to-genome alignment problem, which is the main motivation of the work in this paper, are examples of this approach. More reviews and surveys for reads-to-genome alignment solutions and approximate pattern matching in general can be found at (*7–10*).

Given the larger constant factor of suffix trees when compared to other linear structures such as suffix arrays or FM-index, the design of suffix tree structure is more flexible and dynamic to tackle string problems where approximate pattern matching is one of them.

The structures proposed in this paper is continuation and improvement on the error tree structure proposed as part of the PhD dissertation (*11*) of the first author and was published in this article also (*12*). The name of this structure will be also error tree ET.

Methods

The proposed structure in this paper involves the following four phases where further explanations, elaborations, and motivations for each phase are provided in the Supplementary Material. The first stage is to build suffix tree of the input data and preprocess the tree. Secondly, building a novel index that index all suffixes under all internal nodes in the suffix tree in linear time and with maintaining the inter-connectivity among the suffixes under different internal nodes. The third stage is to collect keys of nodes at depth m in the path of each suffix in the input data and map these outcomes to the index in stage 2. Lastly, the algorithm based on error tree to compute the approximate pattern matching is described.

Stage 1: Building suffix tree

Build a suffix tree (ST) for S ensuring the following attributes are constructed with the ST: $depth$ attribute at each node which is equal to the lengths of all edges from root to the node, $Suffix\_index$ attribute at each node which is equal to the index in S of the starting character of string extracted from root to that node, $parent$ attribute at each node that stores the memory location of parent node, and $Suffix\_links$ (which must be originally constructed in ST) should be preserved. The cost of this stage is linear in space and time.

Stage 2: Building a novel index

In a general analysis of suffix trees, there are main challenges that are easily faced and needed to be carefully addressed in order to resolve the approximate matching problem more efficiently or optimally. Firstly, under different internal nodes there are similar suffixes; how can we track these same suffixes across different nodes so that if a process is performed on a single suffix then we can apply or just link the outcomes of this process to the same suffix in different node/s and save the costs of performing the same process again and again. Secondly, the structure under an internal node is symmetric partially or fully to the structure of other internal nodes (we mean by ”partially symmetric” a subtree under an internal node is symmetric with the subtree under another internal nodes); how can we trace and embed these interconnectivity across different internal nodes.

The possible answer to the first challenge is to build a global index for each $O(n)$ suffixes, then record the index value of each suffix under each internal node. Away from the $O(nh)$ cost (where h is the height of ST) of this indexing schema, the index values that will be recorded in some internal node will be distributed randomly. This will lead to a costly computation when the index values of two different nodes need to be compared or intersected for some purposes. In addition, given that this indexing schema will be useful when two internal nodes are fully symmetric, but it will not be highly useful in case two internal nodes are partially symmetric or asymmetric (which are the common cases). So, the indexing schema can not be arbitrary and has to follow some structure and take into account (and take advantage) of the interconnectivity among different internal nodes, but how to find or create this structure and from which suffix/node should I start the indexing schema and in which order. The answers for these questions are in the following structure.

As we need to refer to this index and to distinguish it from suffix indexes in ST, let's denote this index as ***OT*** (the reason for this naming is in the Acknowledgment section) and before showing how to build the index, the following observations must be introduced.

First observations, which is the key building block in construction the aforementioned index, is the following. If node a has a suffix link to node b, then all the set of suffixes *under* node a, denoted as subset A, must be a subset or equal to the set of the suffixes under node b denoted as subset B. This indicates that if we assign index values to the suffixes in subset A, then these suffixes will be implicitly indexed in subset B and we just need to assign new OT index values to the indexes $B - A$. Note that this process will work recursively, in other words, if node b has a $suffix\_link$ to node c, then we will just need to assign OT index values to the set $C - B$ where C is the set of suffixes under node c and there will be no computation or indexing process associated with the set of $C - A$ as they are already covered in OT index.

Secondly, the structure of ST includes the fact that an internal node, let's say node x, may have up to $O(\Sigma)$ nodes with $suffix\_links$ linking to it. Now, in order to build OT index correctly, we should start indexing all suffixes under each node with $suffix\_link$ linking to node x before indexing node x. This indicates a postorder traversal process, hence, we must construct a tree structure in order to perform this postorder traversal.

Building $OSHR$ tree

As we need to refer to this tree structure and to distinguish it from ST tree and other tree structures, the name of this tree is $OSHR$ tree (the reason for this naming is in the Acknowledgment section). $OSHR$ tree is constructed by reversing the suffix links in ST as the following:

- The root of $OSHR$ tree is the root of ST.
- Internal nodes are all internal nodes in ST with at least one incoming $suffix\_link$.
- Leaf nodes are all internal nodes in ST with only outgoing $suffix\_link$ (no incoming $suffix\_link$).
- There is a directed edge from node a to node b if b has $suffix\_link$ to node a.

.

Leaf nodes in ST are not included in $OSHR$ tree as there is no $suffix\_link$ outgoing from or incoming to them. Note that by the construction properties of ST and suffix links, $OSHR$ tree will be a directed acyclic graph. Clearly, the space and time cost for building $OSHR$ tree is linear ($O(n)$). The tree structure can be built implicitly (inside ST tree) or explicitly (outside

ST). For graphical representation of ST, this webtool `https://hwv.dk/st/?$` can draw ST tree nicely, place the string between ? and $.

Building OT index

Building OT index requires both ST and $OSHR$ trees. The following steps describe how to build the index:

- Initialize a counter, c, to zero and a list, OT, with size equal to the number of leaves in ST with each element in the list is initialized to -1 (for instance).
- Traverse $OSHR$ tree using postorder method. At each traversed node, let's say node x, scan each leaf node, let's say leaf node l which is within the construction of ST tree not $OSHR$ tree, under x and compute the length of the suffix from node x to leaf l. This length can be computed as: $s = suffix\_index$ of l + $depth$ of node x. Now, check if $OT[s]$ equals to -1. If yes, let $OT[s]$ equal to c and then increment c. If not, then do nothing (which means this suffix length is already covered in OT index).
- Perform the second step until postorder traversal is completed.
- For each node, let's say node x, in $OSHR$ tree, set two attributes: $OT\_index\_of\_leftmost\_suffix$ and $OT\_index\_of\_rightmost\_suffix$. Assign $OT\_index\_of\_leftmost\_suffix$ the value of OT index of the length of suffix from node x to $leftmost\_leaf$ under node x to the attribute; similarly for the attribute $OT\_index\_of\_rightmost\_suffix$. This process can be performed within the second step.

Clearly, the space cost of OT is linear but the time cost will be $O(nh)$ as the leaf nodes under each internal node in $OSHR$ should be processed. However, the following algorithm shows how to build OT index in linear time.

Linear time construction of OT index

The non-linear (nh) time cost is due to the trivial process of checking whether each suffix under each internal node is already indexed in OT index. This process is needed to find the set $B - A$ under node b and index them in OT index (as we know already that the set $B \cap A$ was already indexed when we visited node a and we know that set A is a subset of set B.

Let's denote the set of suffixes that need to be indexed in OT index under node x to be $uncovered\_suffixes\_list(x)$. Note that the $uncovered\_suffixes\_list(.)$ at all internal nodes will be bounded to $O(n)$, then the algorithm that could find directly all $uncovered\_suffixes\_list(.)$ of all internal nodes (without processing all suffixes under each internal node) will cost linear time. The following paragraphs describes the linear algorithm to achieve so.

After long and deep thoughts and analysis, the $uncovered\_suffixes\_list()$ of node x could be found using both structures of ST and $OSHR$ trees and processing child (direct child) leaf nodes of node x, $uncovered\_suffixes\_list()$ of child (direct child) internal nodes of node x, nodes that link (back and forth) to node x and its child internal nodes. So, using both ST and $OSHR$ trees, a postorder traversal of ST tree, and several rules/tricks (that can be derived from python code snippet in Listing 1, $uncovered\_suffixes\_list()$ of all internal nodes can be computed in linear time and space. Few minor rules/tricks, that until now cost overhead on true negative cases more then saving costs on true positive cases, are currently under investigations and implementation for inclusion or exclusion.

Next, we traverse $OSHR$ in postorder method and build OT index using the $uncovered\_suffixes\_list()$ of all internal nodes only (again without processing all suffices under all internal nodes). Moreover, as a validation metric and as shown in the implementation, the above traversal does not involve checking whether any suffix in any list is already existed in the OT (it involves processing the lists directly without any checking which indicates their correctness),

As the eventual upper bound of OT index is linear $O(n)$ (in which the $uncovered\_suffixes\_list()$

and the *already* covered suffixes are indexed) and as the upper bound of all $uncovered\_suffixes\_list()$ of all internal nodes is also linear $O(n)$ (in fact they count almost half of the n suffixes), the cost of the algorithm is linear (the linear bound was also verified by implementation).

Another expectedly linear algorithm in which suffixes indexes of each leaf node are processed in a bottom-up approach. Briefly, starting from each leaf node, we walk to parent node and check whether the previous suffix index ($suffix\_index\_at\_leaf\_node - 1$) is existed (covered) under any node from the nodes that link to the parent node. If no, then append $suffix\_index\_at\_leaf\_node$ to the $uncovered\_suffixes\_list()$ of parent node and walk up to the upper parent node. If yes, then stop the walk. The process proceeds until root node is reached if needed. This algorithm is under implementation and testing. The initial intuition of the cost of this algorithm is $O(nh)$, but there are some rules/tricks that can be applied which may render the algorithm to be linear. With or without these rules/tricks, we will check, proof, and show its linearity (or non-linearity) and compare it with the trivial algorithm and the linear one.

Stage 3: Building Error Tree

Note that OT index can be useful for different string processing problems not only for approximate pattern matching, so its description and presentation is independent. Now, for approximate pattern matching problem, we are using some processes performed earlier in error tree structure with major modifications made due to the presence of OT index.

- Set a key attribute for each leaf node where the key of leftmost leaf node must be 0, the next leaf node is 1, and proceed until the rightmost leaf node which must have a key equal to the number of leaf nodes minus 1. Likewise, set attribute named as key to each internal node in ST. The assignment of unique keys for internal nodes should not intersects with leaf-nodes' keys. The reason for separating the keys assignment between leaf nodes and internal nodes is to make the keys of leaf nodes serialized (from left to right) so they can be retrieved faster. The cost of this process is linear.

As the pattern length is given, one may trim the ST tree then records the indexes of suffixes of the trimmed nodes in the new leaf nodes of the trimmed tree. This reduces the space cost especially when the pattern length is short and in the case the final structure may be serialized and de-serialized for later usage. This trimming process is omitted in this paper.

- Initialize a dictionary (or list), denoted as ETD, where keys are integers and values are lists. Now, collect for each suffix the key of the first node with depth equal to or more than m, let this key be y, then append to the list of $ETD[y]$ the value of $[SI - 1]$ where SI is the index of suffix in the input data (the reason for subtracting 1 will be shown in the next section). As we will need to store a single key for each $O(n)$ suffixes, the cost of this process is linear in space and time.

- Once ST traversal has finished, then for each key in ETD initialize an empty list, L. Now, for each suffix index, SI in $ETD[key]$ find $OT[SI]$ and append it to L. Once all elements in $ETD[key]$ appended to L, sort L and assign it back to $ETD[key]$. As finding OT index value will cost a constant time and the list of $ETD[y]$ contains integer values (hence sorting will cost linear time) and as all ETD contain in total n suffixes, the cost of this process is linear.

Searching for approximate pattern matching

The following algorithm shows how to search for pattern of length m with up to k Hamming distance (edit distance will be described later).

Let's assume $k = 1$, find the key of the node that reached by each suffix of the pattern. If a suffix ends in the middle of an edge, then return the key of the sink node of that edge. This process will cost linear time and space using the suffix links of ST. Note that the keys of some suffixes will be the same, so remove duplicated keys. Moreover, record for each suffixes the number of mismatches occurred on the edges.

Now, walk the pattern in ST. If the walk is on an edge and a mismatch occurred at position x, then proceed the walking as exact matching until the end of pattern reached. If reached, report approximate matching at position x which occurred at positions equal to the suffix indexes under the reached point. If not, report no approximate matching with $k = 1$ value.

If the walk reach no mismatches on an edge and reach an internal node, let's say node e, then find the positions of $OT\_index\_of\_rightmost\_suffix$ and $OT\_index\_of\_rightmost\_suffix$ in the list of $ETD[x]$ where x is the key of pattern's suffix $depthofnodee + 1$. Now, map back the OT index values found in between $OT\_index\_of\_leftmost\_suffix$ and $OT\_index\_of\_rightmost\_suffix$, if any, to their suffix indexes and report these indexes as the approximate matching of the pattern at position equal to the $depth$ of e. Continue the searching using the above manner until the end of pattern is reached.

For the case of $k \geq 2$, after inserting the $OT\_index\_of\_leftmost\_suffix$ and $OT\_index\_of\_rightmost\_suf$ get the first index value to the right of the $OT\_index\_of\_leftmost\_suffix$ and the first left index value to the left of the $OT\_index\_of\_leftmost\_suffix$ and proceed with these values in the next insertion for the upcoming ETD. Proceed in similar fashion.

The time cost for search for a pattern of length m will be $O(\frac{m^k}{k!} + occ)$ in the worst case where ST tree is unbalanced and $O(m + \frac{log^k_\Sigma n}{k!} + occ)$ in the average case where ST is balanced. Note that the approximate matching outcomes of $k - 1$ can be used for approximate matching process of k. Lastly, given that upper bound of the number of patterns that are at k Hamming distance (combinations) with a pattern of length m is $O(\frac{m^k}{k!})$, however, merging and sorting the values of $OT\_index\_of\_leftmost\_suffix$ and $OT\_index\_of\_rightmost\_suffix$ of each internal node in ST with all OT index values in ETD (which needs additional $O(n)$ space) then apply few trivial procedures can contribute in linear searching time (this algorithm is under implementation).

Category	**Organism**	**Accession**	**Size (byte)**
Virus	Gordoniaphage GAL1 (*13*)	GCF 001884535.1	50,654
Bacteria	WS1 bacterium JGI 0000059-K21 (*14*)	GCA 000398605.1	521,951
Protist	Astrammina rara (*14*)	GCA 000211355.2	1,712,167
Fungus	Nosema ceranae (*14*)	GCA 000988165.1	5,809,207
Protist	Cryptosporidium parvumIowa II (*14*)	GCA 000165345.1	9,216,802
Protist	Spironucleus salmonicida (*14*)	GCA 000497125.1	13,142,503
Protist	Tieghemostelium lacteum (*14*)	GCA 001606155.1	23,672,980
Fungus	Fusarium graminearumPH-1 (*13*)	GCF 000240135.3	36,915,673
Protist	Salpingoeca rosetta (*14*)	GCA 000188695.1	56,150,373

Table 1: Dataset

Results

The structure proposed in this paper was implemented using python language (python3). First process in the implementation is building ST. For building ST, we used a python package (urlhttps://pypi.org/project/suffix-trees/) without any modification and ensuring that the attributes of $index, depth, parent, andsuffix\_link$ are implemented at each node.

Next, ST was traversed using iterative postorder method where in this traversal the following preprocessing steps were computed: assigning serialized keys to leaf nodes from left to right, assigning keys to internal nodes (where these keys do not intersect with the keys of the leaf nodes), collect the keys of nodes at depth m for each suffix in the input data (ETD dictionary), construct $OSHR$ tree, set attributes of *key_of_leftmost_leaf_node* and *key_of_rightmost_leaf_node* to each internal node where the value of *key_of_leftmost_leaf_node* equals the key of leftmost leaf node under the internal node (*key_of_rightmost_leaf_node* likewise), and create two auxiliary lists. First list is to store the suffixes indexes of leaf nodes from left to right. The second list maps suffix index of leaf node to the key of that leaf node; as an example, if the suffix index stored at leaf node is x, then store at position x of the list the key of the leaf node. The size of each list is linear (equals to the number of leaf nodes).

For building OT index, an iterative postorder traversal of ST is computed to collect the $uncovered\_indexes$ (discussed in the section of Building OT index) at each internal node. Next, an iterative postorder traversal of $OSHR$ tree is computed to build the OT index. Lastly, map/convert the suffixes in each key of ETD dictionary to their OT index values then sort the resultant values.

For testing the implementation of ET, we used ten genomes, listed in Table 1, ranging in size from 50KB to 100MB. The time cost for each step involved in the linear building ET structure is listed in Table 1. For each genome, we built ET structure for m value of 30. Table2 shows the time needed to build error tree for each genome. Note that building of OT index took time (and space) close to building ST. This applies also to the step of preprocessing ST tree.

Category	**Size (byte)**	**Building ST (minute)**	**Preprocessing ST (minute)**	**Building OT index (minute)**	**Processing ETD (minute)**	**Total time (minute)**
Gordonia-	50,654	0.01	0.01	0.01	0.00	0.02
WS1 bacterium	521,951	0.10	0.10	0.11	0.01	0.32
Astrammina	1,712,167	0.30	0.29	0.27	0.02	0.90
Nosema	5,809,207	1.37	1.16	1.38	0.09	4.00
Cryptosporidium	9,216,802	2.26	1.93	2.31	0.15	6.65
Spironucleus	13,142,503	3.42	2.84	2.68	0.20	9.14
Tieghemostelium	23,672,980	6.06	6.41	5.00	0.38	17.85
Fusarium	36,915,673	9.77	8.42	10.07	0.63	28.90
Salpingoeca	56,150,373	15.65	13.17	16.06	0.93	45.82
Chondrus	106,387,446	29.02	27.28	34.04	1.61	91.96

Table 2: Building error tree for each genome in the dataset

The structure of ET allows to handle different values of m without the need to reconstruct or re-process ST tree, OT index, or $OSHR$ tree. All what is needed is to construct ETD dictionary for each m value then perform the last step in stage 3 for each dictionary. Lastly, insertions and deletions can be naively tackled by ET structure.

Lastly, the estimated constant factor for the linear algorithm is a bit large (10-16). To test this, we run the trivial algorithm, that needs $O(nh)$ time for building OT index, on each genome in the dataset (data to be collected and shown). We could notice that the trivial is faster than the linear one on small genomes, as expected, but for large genomes the linear algorithm is much faster. This is due to the constant factor of linear algorithm being larger than h factor and the fact that linear algorithm needs to perform two traversal (traverse ST and then $OSHR$ tree) while the trivial algorithm perform only a single traversal (ST tree).

The run time for each genome is presented in Table 3

Data and materials availability: To be published.

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

References and Notes

1. S. I. Hakak, *et al.*, *IEEE access* **7**, 69614 (2019).

2. P. Weiner, *14th Annual Symposium on Switching and Automata Theory (swat 1973)* (IEEE, 1973), pp. 1–11.

3. E. M. McCreight, *Journal of the ACM (JACM)* **23**, 262 (1976).

4. E. Ukkonen, *Algorithmica* **14**, 249 (1995).

5. M. I. Abouelhoda, S. Kurtz, E. Ohlebusch, *Journal of discrete algorithms* **2**, 53 (2004).

6. P. Ferragina, G. Manzini, *Proceedings 41st Annual Symposium on Foundations of Computer Science* (IEEE, 2000), pp. 390–398.

7. M. Alser, *et al.*, *Genome biology* **22**, 1 (2021).

8. G. Kucherov, *Bioinformatics* **35**, 3547 (2019).

9. S. Canzar, S. L. Salzberg, *Proceedings of the IEEE* **105**, 436 (2015).

10. G. Kucherov, K. Salikhov, D. Tsur, *Theoretical Computer Science* **638**, 145 (2016).

11. A. Al-Okaily (2016).

12. A. Al-Okaily, *Journal of Computational Biology* **22**, 1118 (2015).

13. N. A. O'Leary, *et al.*, *Nucleic acids research* **44**, D733 (2016).

14. K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, E. W. Sayers, *Nucleic acids research* **44**, D67 (2016).

Supplementary materials

```
def find_uncovered_suffix_indexes(tree):
    stack.append(tree.root)
    children_stack.append((list(tree.root.transition_links[x] for x in sorted(tree.root.transition_links.keys(), reverse=True))))
    while stack:
        current_node = stack[-1]
        if len(children_stack[-1]) > 0:
            last_node_under_top_node_in_stack = children_stack[-1][-1]
            stack.append(last_node_under_top_node_in_stack)
            children_stack[-1].pop()
            children_stack.append((list(last_node_under_top_node_in_stack.transition_links[x] for x in sorted(last_node_under_top_node_in_stack.transition_links.keys(), reverse=True))))
        else:
            stack.pop()
            children_stack.pop()

            # alongside processing
            temp_uncovered_suffixes_list = []
            if not current_node.is_leaf():
                if hasattr(current_node, "nodes_link_to_me"):
                # if node has nodes_link_to_me attribute then it’s an internal node in OSHR tree
                # where the nodes that link to it are stored in this attribute
                    # collect data from nodes_link_to_me attribute
                    temp_uncovered_suffixes_list = []
                    nodes_linked_to_me = current_node.nodes_link_to_me
                    s = 0
                    for node_linked_to_me in nodes_linked_to_me:
                        s += node_linked_to_me.key_of_rightmost_leaf - node_linked_to_me.key_of_leftmost_leaf + 1
                    # now the value of s is the sum of all suffix indexes under all nodes that link to current_node in the traversal
                    if s != current_node.key_of_rightmost_leaf - current_node.key_of_leftmost_leaf + 1:
                        # if the above condition is false, then there will be no uncovered suffixes at all to search for, as the sum of suffixes under nodes link to
                        # current_node are equal to the number of suffix indexes under current_node
                        for child_node in current_node.transition_links.values(): # current_node.transition_links.values() contains child nodes of current_node as part of ST structure
                            if child_node.is_leaf():
                                # The following lines check whether the the previous suffix index of the suffix of the child leaf node was covered under any of the nodes that link to current_node.
                                # If not covered then add it to the uncovered_suffixes list of current_node
                                # As the opposite of the if statement below are the uncommon cases, we code it for speeding
                                leaf_node_of_previous_suffix_index = tree.leaf_suffix_index_to_leaf_memory_list[child_node.idx - 1]
                                if leaf_node_of_previous_suffix_index.parent._suffix_link != current_node:
                                    f = True
                                    for node_linked_to_me in nodes_linked_to_me:
                                        if leaf_node_of_previous_suffix_index.key in range(node_linked_to_me.key_of_leftmost_leaf, node_linked_to_me.key_of_rightmost_leaf + 1):
                                            f = False
                                            break
                                    if f:
                                        temp_uncovered_suffixes_list.append(child_node.idx)

                            else:
                                if hasattr(child_node, "nodes_link_to_me"): # This means child_node is an internal node in OSHR tree
                                    a = 0
                                    for node_links_to_child_node in child_node.nodes_link_to_me:
                                        # The following lines code a tricky process which compute that if the condition is true, then all suffix indexes under node_links_to_child_node must be
                                        # in the uncovered suffixes under current node
                                        if node_links_to_child_node.parent._suffix_link != current_node:
                                            for suffix_idx in tree.leaf_node_indexes_list[node_links_to_child_node.key_of_leftmost_leaf:node_links_to_child_node.key_of_rightmost_leaf + 1]:
                                                temp_uncovered_suffixes_list.append(suffix_idx + 1)
                                        else:
                                            a += 1

                                    if hasattr(child_node, "uncovered_suffixes"):
                                        if a == len(nodes_linked_to_me):
                                            temp_uncovered_suffixes_list += child_node.uncovered_suffixes
                                            # the above condition cover a special and common case in order to speed up the processing and avoid the computation in else statement below
                                        else:
                                            # The following lines check whether the the previous suffix index of the suffixes in child_node.uncovered_suffixes list was covered under any of the nodes that
                                            # link to current_node. If not covered then add it to the uncovered_suffixes list of current_node
                                            for suffix_idx in child_node.uncovered_suffixes:
                                                f = True
                                                key_of_prev_idx_node = tree.leaf_suffix_index_to_leaf_memory_list[suffix_idx - 1].key
                                                for node_linked_to_me in nodes_linked_to_me:
                                                    if key_of_prev_idx_node in range(node_linked_to_me.key_of_leftmost_leaf,
        node_linked_to_me.key_of_rightmost_leaf + 1):
                                                        f = False
                                                        break
                                                if f:
                                                    temp_uncovered_suffixes_list.append(suffix_idx)
                                else:
                                    # means child_node is a leaf node in OSHR tree (no suffix_link is linking to it), as so, check whether the the previous suffix index of the suffixes in
                                    # child_node.uncovered_suffixes list was covered under any of the nodes that link to current_node. If not covered then add it to the uncovered_suffixes list of current_node
                                    for suffix_idx in child_node.uncovered_suffixes:
                                        # As the opposite of the if statement below are the uncommon cases, we code it for speeding
                                        leaf_node_of_previous_suffix_index = tree.leaf_suffix_index_to_leaf_memory_list[suffix_idx - 1]
                                        if leaf_node_of_previous_suffix_index.parent._suffix_link != current_node:
                                            f = True
                                            for node_linked_to_me in nodes_linked_to_me:
                                                if leaf_node_of_previous_suffix_index.key in range(node_linked_to_me.key_of_leftmost_leaf, node_linked_to_me.key_of_rightmost_leaf + 1):
                                                    f = False
                                                    break
                                            if f:
                                                temp_uncovered_suffixes_list.append(suffix_idx)
                        setattr(current_node, "uncovered_suffixes", temp_uncovered_suffixes_list)
                else:   # means current_node is a leaf node in OSHR tree (no suffix_link is linking to it), as so, just add all suffix indexes under current_node to the uncovered_suffixes list of the node (itself)
                    temp_uncovered_suffixes_list += tree.leaf_node_indexes_list[current_node.key_of_leftmost_leaf:current_node.key_of_rightmost_leaf+1]
                    setattr(current_node, "uncovered_suffixes", temp_uncovered_suffixes_list)
```

Listing 1: Finding Uncovered Suffix indexes under Internal Nodes