An Introduction to the Suffix Tree and its Many Applications

Michelle Gower
Research Readiness
University of Missouri - Rolla *

June 21, 2000

Abstract

One data structure that appears very seldom in data structure books yet is used to solve many string problems is the suffix tree. This paper first explains what a suffix tree is and briefly explains how to build one in linear time. Then a small survey is given describing what problems in string matching, computational biology, and data compression have been solved using a suffix tree. More detail has been provided for an example from each area. This paper finishes with a very brief survey of modifications to the suffix tree.

1 Introduction

One data structure that appears very seldom in data structure books is the suffix tree. While its construction is much harder to describe than the common data structures, the suffix tree has been used in the solutions for many string problems, which makes it an interesting structure to study.

This paper first explains what a suffix tree is and briefly explains how Weiner [43], McCreight [30], and Ukkonen [41] each built one in linear time. Then a short survey

*email: mgower@umr.edu This research was facilitated in part by a National Physical Science Consortium Fellowship and by stipend support from NASA - Ames Research Center, supervisor - Charles Jorgensen.

is given describing what string problems have been solved using a suffix tree. While string searching and matching problems can be interesting by themselves, they are also important parts of solutions to other problems in areas such as computational biology and data compression. So, this paper also surveys other applications in those areas. To illustrate how a suffix tree might be used to solve these types of problems, more detail is given for an example from each of those areas. The last section contains a very brief survey of modifications to the suffix tree.

2 Building a suffix tree

The **length** of a string S will be denoted by $|S|$. Let the string S = $s_1 s_2 ... s_{|S|}$ where s_i is a character from a fixed finite alphabet Σ. Let S[i] = s_i and S[i..j] = $s_i s_{i+1} ... s_j$. Given a string S, a **suffix** of S is any string w such that S = uw for some string u. And a **prefix** of S is any string w such that S = wu. In both cases, the string u may be empty. A **substring** of a string S is any string w such that S = uwv for some, again possibly empty, strings u and v. [41] For example, suffixes for `mississippi` are from longest to shortest: `mississippi`, `ississippi`, `ssissippi`, ... , `pi`, `i`.

Ukkonen [41] and Nelson [32] easily introduce the suffix tree by starting with another data structure called a **suffix trie**. The suffix trie for `mississippi` is shown in Figure 1. Each edge in this tree-like structure is labeled with a single character. No two edges from the same node can be labeled with the same character. The concatenation of the edge labels along a path from the root to a node will be called a `path label`. A string `w` is a substring of a string S if and only if it has a path

label in the suffix trie for S. So a suffix trie has a path label corresponding to each suffix. The path label of each leaf is obviously a suffix, but not all suffixes end at a leaf. For example, the suffix `i` ends at an internal node. This happens because `i` is a prefix of at least one other suffix (e.g., `ississippi`, `issippi`, and `ippi`). The easiest way to build a suffix trie for a string S is to add each suffix to the trie one by one. In the worst case a suffix trie can be built in $\mathcal{O}(|S|^2)$ time using $\mathcal{O}(|S|^2)$ space. This can be illustrated by building a suffix trie for a string in which no character is repeated. This causes each suffix to be an independent path. Not counting the root, the trie will have a path containing $|S|$ nodes, $|S|-1$ nodes ..., and 1 node. The total number of nodes is thus $\mathcal{O}(|S|^2)$ and building the trie will take $\mathcal{O}(|S|^2)$ time.

In the suffix trie there may be nodes with only 1 child, but in a **suffix tree** each internal non-root node must have at least 2 children.[1] To accomplish this, edges are allowed to have strings as labels. The suffix tree for `mississippi` is shown in Figure 1. A suffix trie can be converted into a suffix tree by concatenating edges. Any path between two nodes that only contains nodes which have only 1 child is compressed into one edge. The labels are concatenated and the extra nodes removed.

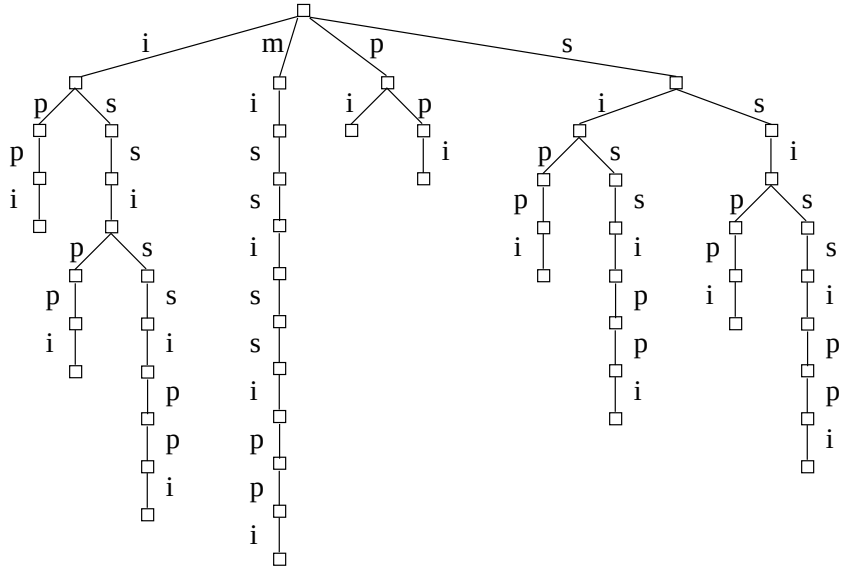
A suffix tree can easily be built without constructing a trie first. Start with a single edge representing the longest suffix (i.e. the string itself). Then start adding the rest of the suffixes in order from longest to shortest. Let `w`, `x`, and `z` be strings, possibly empty, and `a` and `b` be different characters. Assume that a suffix `wax` is already in the suffix tree and the suffix to be inserted is `wbz`. Follow the path label

[1] Giegerich and Kurtz [19] call the suffix trie an **atomic suffix tree** since it has a single character per edge.

string S =

m	i	s	s	i	s	s	i	p	p	i
1	2	3	4	5	6	7	8	9	10	11

(a) Suffix Trie



(b) Suffix Tree

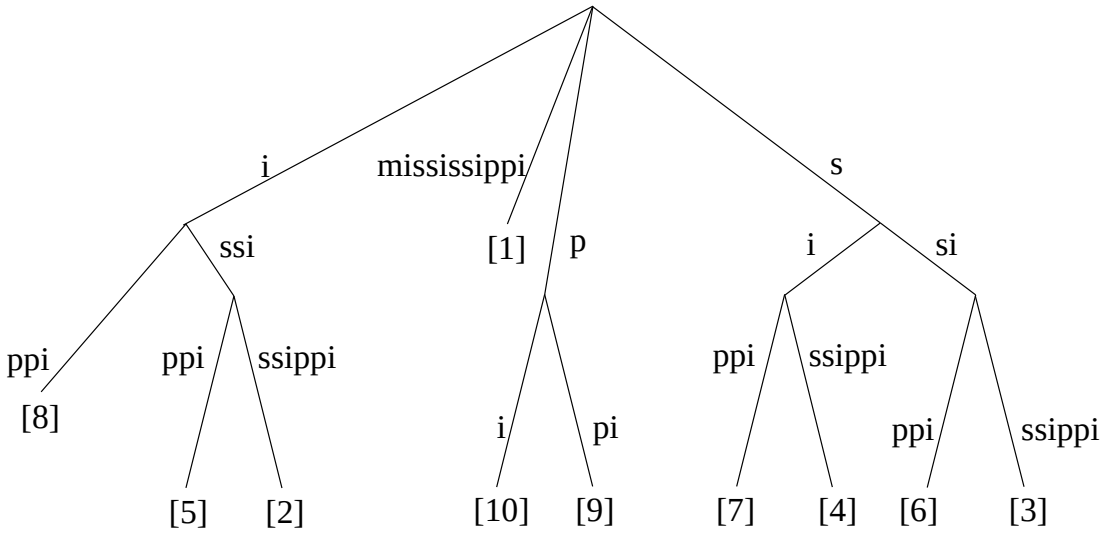


Figure 1: Suffix trie and suffix tree for `mississippi`

for `w` starting from the root. If `w` ends at a node, then from that node create a new edge and node using `bz` as the edge label. If `w` ends inside an edge label, that edge needs to be split so that `w` will end at node. The edge is relabeled with the portion of `w` that was on that edge. The edge is attached to a new node, saving the old node for later. This new node needs two new edges. The first contains the rest of the edge that was split, `ax`. The old node is attached to this edge. The second edge is labeled with the rest of the inserted suffix, `bz`. A new leaf is attached to this edge.

This process takes $\mathcal{O}(|S|^2)$ time. For an example, create a suffix tree for a string with the form $a^{n-1}b$. The algorithm has to make n-1 comparisons to insert the second largest suffix, then n-2 comparisons for the third, ..., 1. The total number of comparisons is $\mathcal{O}(|S|^2)$.

There can be a maximum of $|S|$ leaves because there are at most $|S|$ suffixes. That fact along with requiring more than 1 child per internal node results in a constructive proof that for $|S| > 1$ a suffix tree contains less than $2|S|$ nodes.

Storing the substring on each edge uses quadratic space. Again examine a string S without any repeating characters. It has $|S|$ edges labeled with substrings of length $|S|$, $|S|-1$, ..., 1. The sum of these lengths is $\mathcal{O}(|S|^2)$. There is a simple implementation detail that can change it to linear space. Instead of saving the entire substring on each edge, store the beginning and ending indices of the substring in the whole string (or equivalently the beginning index and the length). Then each edge only needs two numbers even if the string is very long. See Figure 2 for the suffix tree of `mississippi$` using indices as labels.

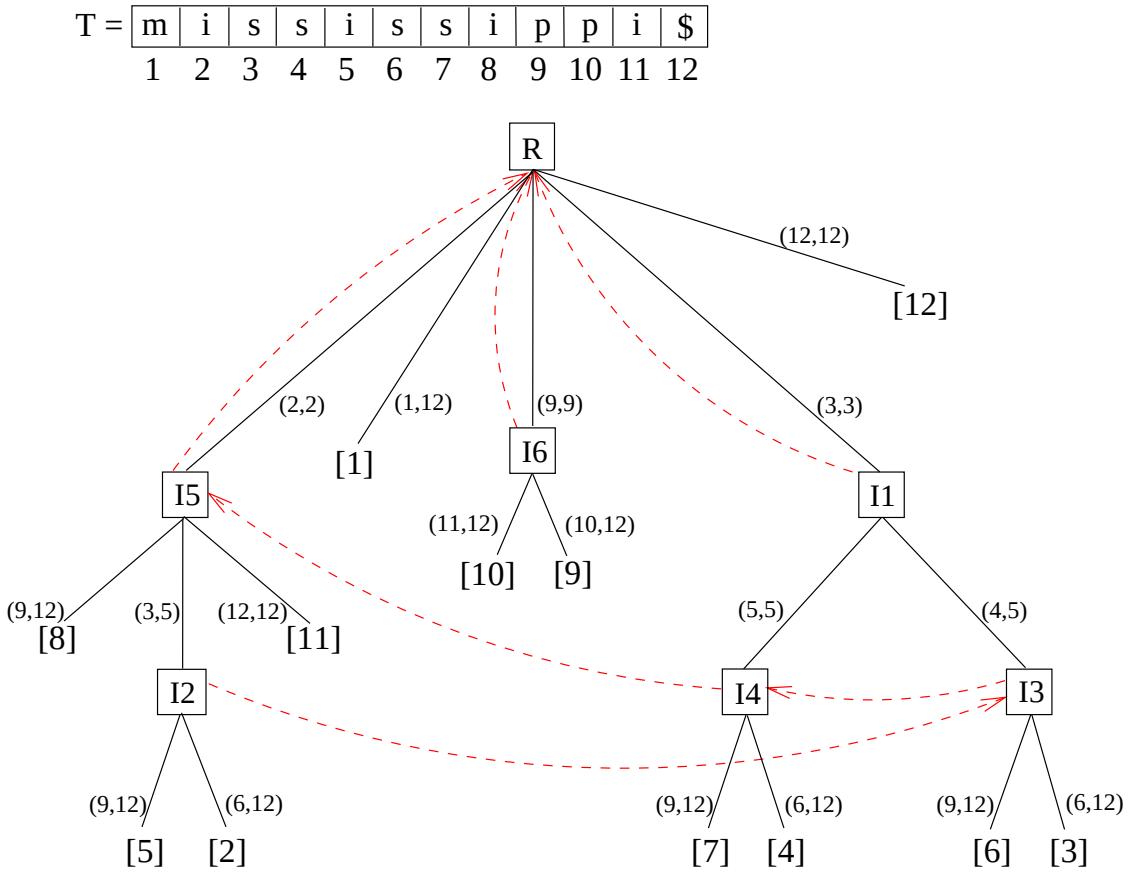


Figure 2: Suffix tree of `mississippi$` showing suffix links as dashed edges and index edge labels. The labels for the interior nodes are not stored and are there for easy reference.

As with the suffix trie, a suffix may not end at a leaf in a suffix tree.[2] For example, the suffix `i` still does not end at a leaf. In this case, it ends at an internal node, but a suffix could have just as easily ended in the middle of an edge label. An example for that case would be the suffix `a` in the suffix tree for `aba`. This tree has an edge labeled `aba` and an edge labeled `ba`. The suffix `a` is already in the tree as the first character of the `aba` edge. Sometimes, algorithms need a one-to-one correspondence between the

[2]Gusfield calls this an **implicit suffix tree** [21]. Giegerich and Kurtz call it a **compact suffix tree** [19].

leaves and the suffixes.[3] The easiest way to ensure that all suffixes end at a leaf is to require that the string ends in a character that is not in the alphabet (most papers use the $ character). Then the leaves can be labeled with the index of where their suffix starts in the string. This does not change the order of magnitude of the space needed to store the suffix tree. Figure 2 has a one-to-one correspondence between the leaves and the suffixes because it used $ as the last character in the string. It also shows leaf labels.

Since the suffix tree only requires linear space, it seems possible for the tree to be constructed in linear time. There have been three main contributors to the linear construction of suffix trees. In 1973 Pete Weiner [43] designed the first linear time algorithm to create suffix trees.[4] He adds the suffixes from smallest to largest thus needing the entire string before beginning. Since his algorithm needs the entire string before starting, it is useless for any application that processes a string from left to right. Besides being difficult to understand, his algorithm uses extra storage to find the point of insertion in $\mathcal{O}(1)$ time [30, 20]. In 1976, Edward McCreight [30] improved on Weiner's algorithm for building suffix trees. His algorithm also needs the entire string since it adds suffixes from the largest to smallest. He explicitly recommends using a hash table to store the edges in the suffix tree, especially when the alphabet becomes large. When using the tree or even while building, algorithms need to find which edge from a node begins with a certain character. By using a hash table one can reduce the average search time for that edge. To get rid of the extra space

[3] Giegerich and Kurtz call this a **position suffix tree** [19].
[4] In this paper, Weiner called them **bi-trees**.

requirements in Weiner's algorithm, McCreight introduced new links, called **suffix links**, to the tree. These allow the algorithm to find the point of insertion in $\mathcal{O}(1)$ time. McCreight makes a suffix link from every nonterminal node, whose path label is $x\delta$ where $x \in \Sigma$ and $\delta \in \Sigma^+$, to another internal node with path label δ. See Figure 2 for the suffix tree of `mississippi$` with suffix links. Giegerich and Kurtz show that McCreight's method is the most efficient of the three (although only slightly better than Ukkonen's method) [20].

In 1995, Esko Ukkonen [41] gave the first on-line algorithm to generate suffix trees. It works from left to right and the intermediate tree is actually a suffix tree. As well as being useful since it does not need the entire string to start building the tree, it is a much more understandable method. The key idea is that once you have a suffix tree T_i for S[1..i], to get the suffix tree T_{i+1} for S[1..i+1] all that has to be done is adding s_{i+1} to the ends of the suffixes in T_i. Of course, it also has to add an empty suffix since the one from the previous tree had s_i appended to it. The problem becomes how to accomplish this in $\mathcal{O}(|S|)$ time. There are three main ways Ukkonen does this [41, 21, 32]. He notices that in each iteration his algorithm increments the end counter in the leaves. So, instead, his algorithm uses a symbol to indicate that the label on the edge is "open to grow." [41, p. 255] Thus, his algorithm does not have to update the end counter on the leaves until the very end which can be taken care of in a $\mathcal{O}(|S|)$ traversal. The next shortcut comes from noticing that if s_i already exists at the end of some suffix then it will also exist at the ends of the other smaller suffixes. So, that iteration can be terminated prematurely. The rest of the problem then becomes how to quickly find the suffixes that do need to have s_i added to them.

Ukkonen uses suffix links to form a **boundary path** along all the states that form the ends of suffixes. So, his algorithm can go from one suffix to the next just by following these links (see Figure 2). Now, it is just a matter of keeping track where along that path each iteration needs to start (to avoid incrementing the leaves).

3 Applications

There have been several applications of suffix trees to solve string problems. In the very first paper with linear time suffix tree generation, Weiner [43] shows how to solve the exact string matching problem and the exact set matching problem. Suffix trees have also successfully been applied to the approximate string matching problem [8, 40, 9, 11]. Knut Risvik applied suffix trees to the approximate word sequence matching problem [33]. Apostolico, Bock, and Lonardi show that "the candidates [for] over- or underrepresented words are restricted to the $\mathcal{O}(n)$ words that end at internal nodes of a compact suffix tree, as opposed to the $\Theta(n^2)$ possible substrings."[2, p. 169]. Suffix trees have also been applied to two-dimensional matching [17, 18, 10] and three-dimensional pattern matching [15].

In the nineties one was inundated with news stories about DNA. It appeared in articles about criminal trials, the Human Genome Project, and the first clones. Part of these problems involves solving manipulation of strings. Dan Gusfield has written a book [21] describing the solution of many of these problems. Suffix trees are used in several of those solutions from searching DNA databases to finding repetitive structures in DNA sequences. Motivated by problems in bio-sequence analysis, the

University of California-Davis is working on a long-term project to build "practical, easily understood software for many different string tasks, based around a single resident data structure, the suffix tree."[38, p. 141] Towards that goal, Stoye and Gusfield [38] create a straightforward application of a suffix tree to find contiguous repeats. In another paper, Marie-France Sagot used a suffix tree to find approximate repeated motifs and approximate common motifs in a sequence [35].

Almost anyone who has used a computer knows the importance of being able to compress and uncompress a file. The suffix tree has been applied to the following compression algorithms: LZ-77 [34, 14, 28, 39], PPM* style compression [28], and Burrows-Wheeler transform in block sorting compression [28, 6].

3.1 Example 1: Exact string matching problem

In the **exact string matching problem** a text and a pattern are given, and all occurrences of the pattern in the text must be found. Let T denote the text string and P denote the pattern string. Let $|P| \leq |T|$ since otherwise there is no possibility for the pattern to exist in the text.

Weiner [43] assumes that the text is fixed and ends in a unique symbol. First he creates the suffix tree for it. This takes $\mathcal{O}(|T|)$ time and space. If the pattern is in the text, then it must be a prefix of some suffix of T. So, there will be a path starting from the root that matches the pattern. Finding whether or not a pattern P is in the text only requires $\mathcal{O}(|P|)$ time. Finding all locations of the pattern in the text requires a subtree traversal starting where the pattern ends in the suffix tree. Each

leaf encountered in this subtree belongs to a suffix that has the pattern as a prefix. And each such leaf stores that starting index of the suffix that will also be the starting index of the pattern. For example, using the suffix tree for `mississippi$` from before (Figure 2), let the pattern equal `ss`. Starting from the root (node `R`), the algorithm follows the link to the internal node `I1`. It matches the remaining letter `s` with the first character on the edge to `I3`. Next the algorithm does a traversal of the subtree rooted at `I3` keeping track of the labels of the leaves it encounters: `6` and `3`. Those two labels do correspond to exactly the starting positions of `ss` in `mississippi`.

The subtree traversal can be done in $\mathcal{O}(k)$ time where k is the number of times the pattern appears in the text. The worst case time for the solution is $\mathcal{O}(|T|+|P|)$, which is as good as the classical methods such as Boyer-Moore and Knuth-Morris-Pratt.

Chang and Lawler [8, 9, 21] give a sublinear solution to the exact string matching problem by building the suffix tree for the pattern. Immediately they save space in the suffix tree. They use the suffix tree to calculate two vectors they call **matching statistics** for each position in T. The first vector, M(T,P), stores at each position i the length of the longest substring of the pattern that matches a prefix of T[i..$|T|$]. The algorithm also stores a pointer into the suffix tree to the ceiling of the matched substring in the second vector, M'(T,P). The **ceiling** is the node on the end of the edge where the substring ends in the suffix tree. If a character in the text does not appear in the pattern, the node ceiling is just the root. Given a suffix tree, these vectors can be calculated in $\mathcal{O}(|T|)$ time and space. Instead of starting from the root for each position i, they use the suffix links to help find the next longest match.

While the same example as given in the previous algorithm can be used again, it

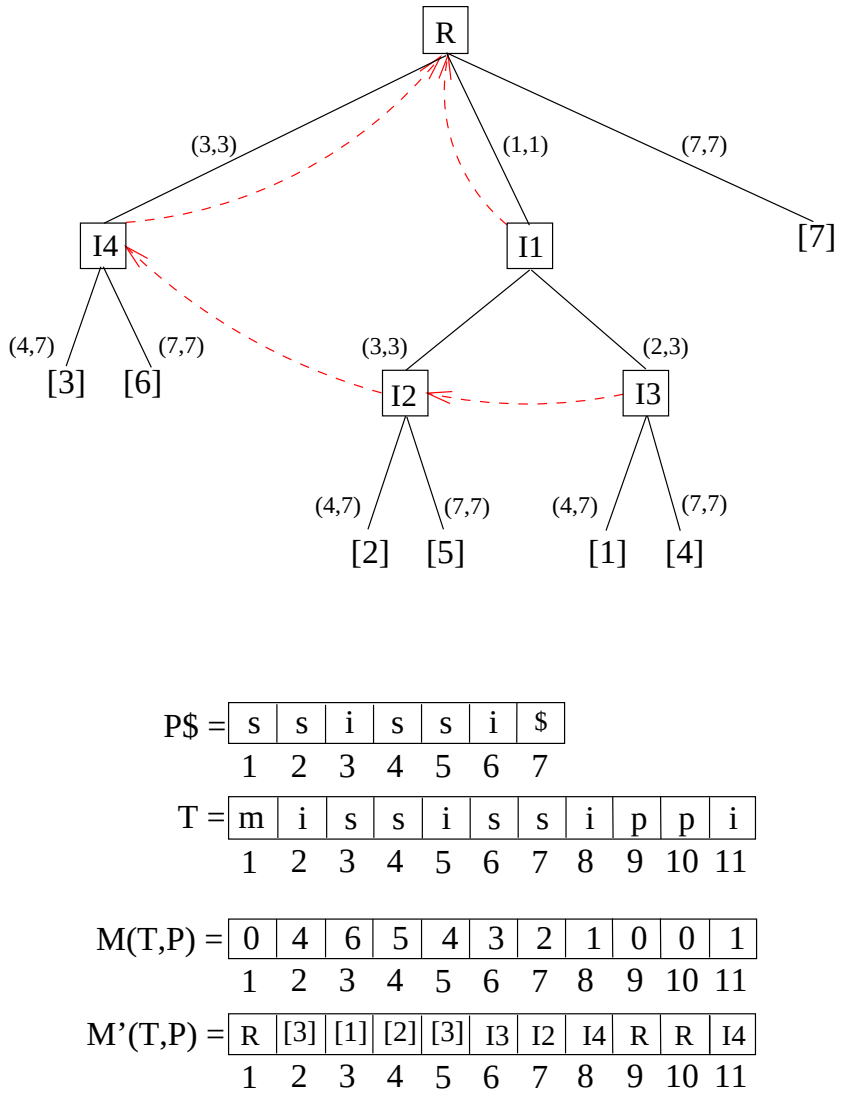


Figure 3: Suffix tree of `P$ = ssissi$`. As in the previous figure, the labels for the interior nodes are there for easy reference.

does not create a suffix tree that will illustrate how they used the suffix links. So, this time, let the pattern equal `ssissi`. The first step is to create the suffix tree for the pattern (see Figure 3). M(T,P)[1] and M'(T,P)[1] are found by starting at the root and descending into the tree while there is a match. After finding a mismatch, it records the length of the match as well as a pointer to the ceiling node. It uses

the suffix link from the node at the other end of that edge to start searching for the match to position 2. The algorithm will match `issi` ending on the edge to leaf `3`. So, 4 is stored in M(T,P) and a pointer to leaf `3` is stored in M'(T,P). After following the suffix link from internal node `I4` to the root node, it knows it can match the characters `ssi`. So instead of comparing those characters, it jumps 3 characters making sure to follow the correct edges. It ends up at internal node `I3`. Then it can continue matching from there. So after matching the next 3 characters, it stores 6 in M(T,P) and a pointer to leaf `1` in M'(T,P). This time when it follows the suffix link, it doesn't end back up at the root. Instead it starts its jumps from internal node `I2`. The complete matching statistics for this example can be found in Figure 3.

Any position i in M(T,P) with length equal to the length of the pattern obviously means that i is the starting position in text of an exact match of the pattern. All the positions of exact matches can be found in $\mathcal{O}(|P| + |T|)$ time using less space than the previous method. Furthermore, by using some probabilities for whether certain substrings of random texts are substrings of the pattern, Chang and Lawler [8, 9] achieve a sublinear average time of $\mathcal{O}(\frac{log|P|}{|P|}|T|)$. They state that Knuth, Morris, and Pratt also have achieved this sublinear time, but that their algorithm must use the classical KMP algorithm to achieve the worst case. So, the suffix tree solution is as good as the non-suffix tree solution.

These matching statistics can be used to save space in some other algorithms. For example, finding the longest common substring just boils down to finding the largest length in the matching statistics. In addition, Gusfield's book also points out more applications where using matching statistics can save space [21].

The **exact set matching problem** is similar to the exact string matching problem except there is a set of pattern strings to search for in the given text. Weiner [43] noticed that by building the suffix tree once and following the same process for searching as in the exact string matching algorithm he could solve the exact set matching problem in $\mathcal{O}(|T| + |P_1| + |P_2| + \ldots + P_k|)$ time. Trying to apply either Boyer-Moore or Knuth-Morris-Pratt to each pattern results in worse time, but Weiner [43] notes that a linear time solution was created by Karp, and Gusfield's book [21] presents Aho and Corsack's linear solution.

3.2 Example 2: Tandem Repeats

Jens Stoye and Dan Gusfield use a suffix tree to easily find tandem repeats [38]. "VNTRs [variable number of tandem repeats] occur frequently and regularly in many genomes, including the human genome, and provide many of the markers needed for large-scale genetic mapping." [21, p. 141]

A string S is a **tandem repeat** if it can be written as S = $\alpha\alpha$ for some substring α. For example, if S = `mississippi`, both `ss` and `ssissi` are tandem repeats. A **branching tandem repeat** is a tandem repeat such that the character immediately after the repeat does not match the first character of the repeat. The first character of the repeat is also the first character of α, and the character at the end of the first α will be the first character of the second α. Thus the character after the first α will be different than the character after the second α. Again in the `mississippi` example, `ississ` is not a branching repeat, but `ssissi` is a branching repeat.

Given a string T, Stoye and Gusfield first find all the branching tandem repeats and then use those to find the non-branching repeats. A **leaf-list** of an internal node is a list containing the labels of the leaves in its subtree. Creating each list can easily be done with a traversal of the subtree. They do not actually store these lists since it would take $\mathcal{O}(|T|^2)$ space and they want to match other algorithms that only use $\mathcal{O}(|\mathcal{T}|)$ space. Instead they will create the lists when needed.

In order to be able to test if a node is in a leaf-list in constant time, they do store values they call **dfs numbers**. To get these numbers, they preprocess the suffix tree with a depth-first traversal. Each leaf gets assigned a successive value based upon when it is encountered in the traversal. The dfs number for each leaf is also stored in an extra array, DFS, in the spot corresponding to its leaf label. Also, during this depth-first traversal, each internal node stores two of these dfs numbers. On the way down, it stores the dfs number that the first leaf in its subtree will use. On the way up, it stores the dfs number of the last leaf in its subtree. See Figure 4 for an example of leaf-lists and dfs numbers. Given that the path label to a node v has length l, a leaf j = i + l is in the leaf-list for v if DFS[j] is between the two dfs numbers stored in v. See Table 1 for a version of their algorithm to find all branching tandem repeats. For the `mississippi$` example, the algorithm finds tandem repeats of length 2 starting at positions 3, 6, and 9 and of length 6 starting at position 3.

Now that they have all the branching tandem repeats, they test left-rotations of those repeats to see if the rotations are also tandem repeats. Figure 5 illustrates how this left-rotation works and Table 1 contains the actual algorithm. This rotation finds the last tandem repeat in `mississippi$` that starts at position 2 and has a length

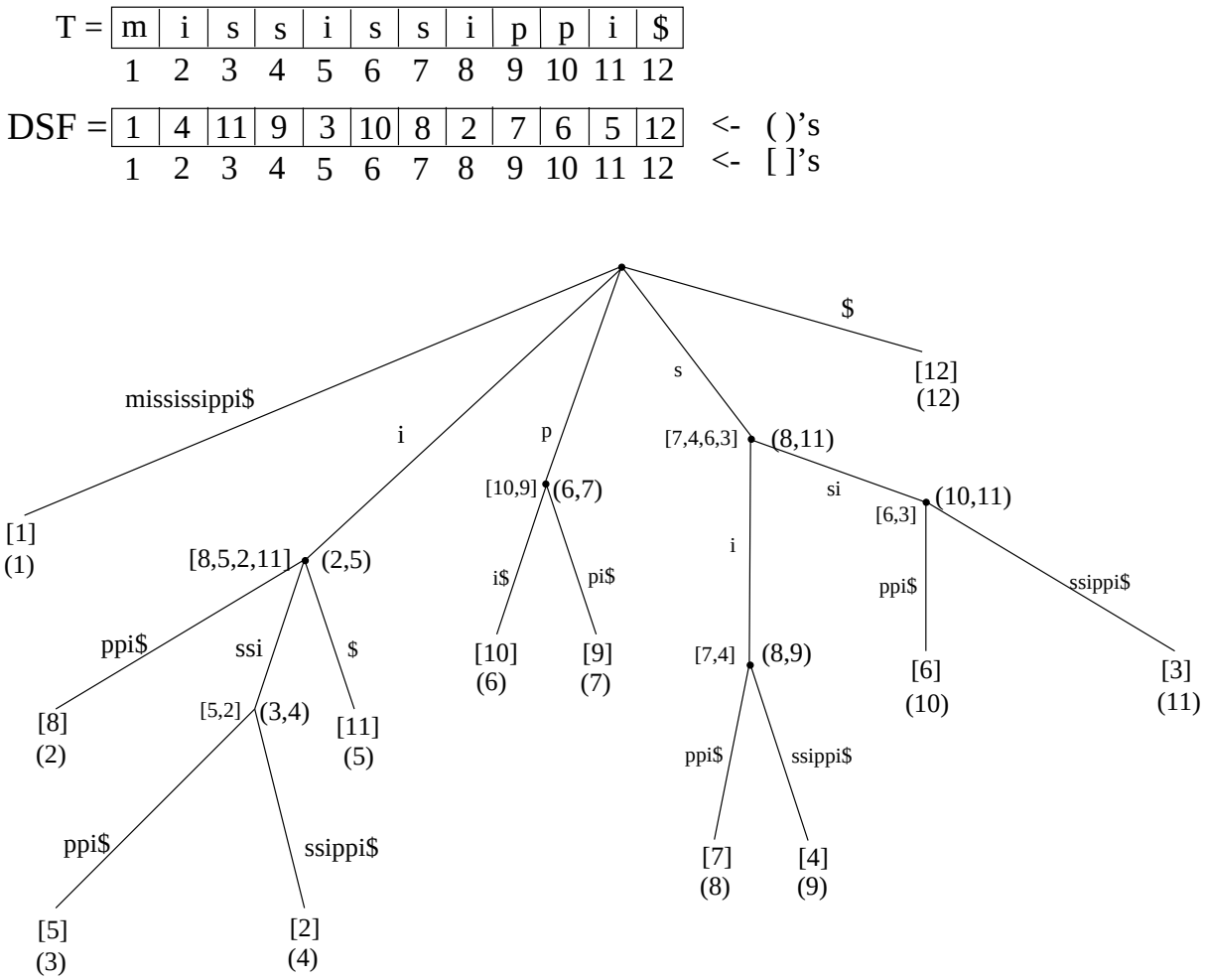
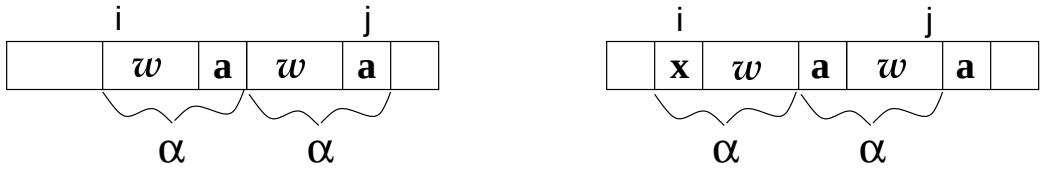


Figure 4: Suffix tree of `mississippi$` showing leaf labels and leaf-lists in []'s and dfs numbers in ()'s. (combination of figures from [38]) Note: substrings and leaf-lists are not actually stored in tree, but are shown for clarity of the algorithm.



(a) Before rotation

(b) After rotation, x must equal a for it to be a repeat

w = some substring (possibly empty), a = some character, x = some character

Figure 5: Illustration of left rotation to find non-branching tandem repeats

(a) Optimized basic algorithm to find all branching tandem repeats in a string T

```
Create suffix tree for T and create the DFS array
For each internal node v:
   let D(v) be the length of the path label to v
   Find the child v' with the most leaves in its subtree by
      using the dfs numbers stored in each child.
   Create a leaf-list for v without using the subtree of v'.
   For each leaf i in that leaf-list:
      let j = i + D(v)
      if DFS[j] is between v's dfs numbers (inclusive)
         if T[i] != T[j + D(v)]
            there is a branching tandem repeat starting at
            position i with length 2D(v)
      let j = i - D(v)
      if DFS[j] is between v's dfs numbers (inclusive)
         if T[j] != T[j + 2D(v)]
            there is a branching tandem repeat starting at
            position i with length 2D(v)
```

(b) Algorithm to find non-branching tandem repeats

```
For each branching tandem repeat:
   let i = starting position of tandem repeat - 1
   let len = length of the tandem repeat
   let j = i + len - 1
   repeat while i >= 1 and T[i] = T[j]
      there is a non-branching tandem repeat starting at
         i with the same length
      decrement i and j by 1
```

Table 1: Stoye and Gusfield's algorithms to find tandem repeats

of 6. Their algorithm uses $\mathcal{O}(|T|)$ space and takes $\mathcal{O}(|T|log|T| + z)$ time. So they met their goal of matching in time and space the other known algorithms.

3.3 Example 3: Sliding window for data compression

Jacob Zempler and Abraham Liv [37, 28] created a data compression algorithm, LZ77, that looks at previously processed text for patterns instead of looking in a dictionary. This classical compression method uses a sliding window to access the text sections at a time. The left part of the window is a search buffer and the right part is called a look-ahead buffer. The main work in this algorithm is finding the longest match between the search buffer and the beginning of the look-ahead buffer.

Finding the longest match is easy given a suffix tree of the search buffer. In a manner similar to the exact string matching problem, the algorithm just walks down the suffix tree as long as it can match characters in the look-ahead buffer. Not counting construction time for the suffix tree, this search is linear to the length of the match. The problem is maintaining the suffix tree as the window moves. A new character enters the right end and a character leaves the left end.

Fiala and Greene [14] based their algorithm upon McCreight's suffix tree construction algorithm. However, as N. Jesper Larsson [28] points out, McCreight's algorithm requires the rightmost character when starting in order for it to be a linear algorithm. The natural construction algorithm for strings that grow on the right is Ukkonen's algorithm, which is on what Larsson bases his sliding window. Now, the problem is removing the suffixes that contain the character that leaves on the left end.

Rodeh, Pratt, and Even [34] handle deleting suffixes in an unexpected manner. Instead of deleting the suffix, they divide the text into overlapping sections whose size is less than half the window size. At any given time, they need a maximum of 3 of these sections. Each of these sections is stored as a suffix tree. When the left-most section is no longer needed, its suffix tree is deleted and the middle section becomes the left section, and the right section becomes the middle section. Thus the deletion is no longer removing one suffix but is just the destruction of an entire tree. While this method does take $\mathcal{O}(|T|)$ time and $\mathcal{O}$(size of window) space, it uses extra suffix trees.

Instead of using the extra suffix trees, an algorithm could actually just delete the suffix containing the old character [14, 28]. Since it is the left-most character in the window, that character will only appear in the longest suffix. So, the algorithm would need to quickly find the leaf associated with that longest suffix and remove it. This is straightforward to accomplish by using a data structure such as a circular buffer [14] for the leaves. But there are other things to worry about. The algorithm must handle the case when a parent node now only has one child due to the deletion of a child. All it must do is combine the child edge with its parent's edge. Because they do not have the \$ in their tree, some suffixes may not end at a leaf. This does not apply to the one being removed, but when removing the leaf it may remove other suffixes that happen to end on that leaf's edge. For example, go back to the suffix tree for the pattern `ssissi$` (Figure 3). If \$ had not been added to the tree, there would be no internal node `I3` and the edge from the node `I1` to leaf `1` would be labeled `(2,6)`. But suffix `ssi` would end in the middle of that edge and would be

lost if leaf1 and its edge were just removed. But Larsson notes that if this is the case, then the end of that suffix is actually the point where the next insertion starts. So, a simple comparison can tell whether or not a suffix might be accidentally lost. If so, the algorithm needs to be replaced by the correct leaf. Larsson's algorithm also takes care of the situation where the insertion point may be invalidated. To solve the last problem with deleting a suffix, Larsson used a technique created by Fiala and Greene. They use a binary credit system to make sure that edge labels stay within the window size in linear time.

Let the size of the window = M and the size of the text = n. Using this suffix tree sliding window algorithm, Larsson derived an algorithm that takes $\mathcal{O}(M)$ space and $\mathcal{O}(n)$ time to compress the text.

4 Other related topics

This paper does not cover all topics related to suffix trees. The purpose of this section is point out some of these topics and give interested readers a place to start.

As stated at the beginning of this paper, the alphabet is assumed to be a fixed finite alphabet. Martin Farach [13] looks at a unique linear construction algorithm for an integer alphabet in the range [1,n].

As with other areas, people have tried to parallelize the construction of a suffix tree. There are a handful of sources describing parallel construction [27, 5, 36, 22, 23, 3]. There are also parallel algorithms for on-line searching [3], and three-dimensional pattern matching [15]. S. Rao Kosaraju [26] constructs only part of a suffix tree in

real-time. But using this portion of the suffix tree, Kosaraju's algorithm can perform pattern matching in real-time.

Along with his online method to construct a suffix tree, Esko Ukkonen shows how to build the suffix automaton (or DAWG). The suffix automaton is the smallest deterministic finite automata that can recognize all suffixes of a given string. It is a compressed version of the suffix tree. Along with Derick Wood [42], he shows how to solve the approximate string matching problem using a suffix automaton.

Another data structure that was created to save space is the suffix array. It is an array containing indices to the suffixes of the string. These indices are sorted so that they point to the suffixes in lexicographical order. Its construction time and space are independent of the alphabet. Udi Manber and Gene Myers [29] introduce this data structure in order to work on huge genetic sequences where the size of the suffix tree became a problem. They claim that suffix trees require "roughly 4 integers of overhead per text character." [29, p. 945], but their suffix array only requires "1.25 integers of overhead per text character."[29, p. 945] Suffix arrays take a little more time to build, but can be used for just as fast searches. Example uses of a suffix array are on-line string searching [29], and for inexact matching in shotgun sequencing[5] [12]. Distributed algorithms have also been written for the construction of suffix arrays [31, 25]. Juha Kärkkäinen introduced another data structure that combined aspects of the suffix tree and the suffix array [24]. This structure has not become as popular as the suffix tree or array.

[5]Briefly, in **shotgun sequencing** DNA is blown into pieces (i.e. as if with a shotgun) and the pieces must be put back together in the correct order.

Sometimes a solution needs to search through multiple strings. That is why the generalized suffix tree was created. A leaf may exist in more than one string, so it must store multiple labels that now need another value to associate the label with a particular string. The edges will also need this extra value in order to associate the indices with a particular string. It can be built in time and space linear to the sum of the strings. Gusfield shows how a generalized suffix tree can easily be used to find the longest common substring of a set of strings and finding the longest suffix-prefix match in linear time [21]. It has also been used in a parallel sequence alignment algorithm [7].

5 Conclusion

A suffix tree is a versatile data structure that can be used to solve many string matching problems as well as problems in hot areas such as computational biology and data compression. The suffix tree can be built in $\mathcal{O}(|S|)$ time. After the suffix tree is built, straightforward tree traversals solve many of these problems.

References

[1] *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing* (Montréal, Québec, Canada, 23–25 May 1994).

[2] Apostolico, Bock, and Lonardi. Linear global detectors of redundant and rare substrings. In *DCC: Data Compression Conference* (1999), IEEE Computer Society TCC, pp. 168–177.

[3] Apostolico, A. On-line string searching. In Apostolico and Galil [4], ch. 3, pp. 89–119.

[4] Apostolico, A., and Galil, Z., Eds. *Pattern Matching Algorithms.* Oxford University Press, Inc., 1997.

[5] Apostolico, A., Iliopoulos, C. S., Landau, G. M., Schieber, B., and Vishkin, U. Parallel construction of a suffix tree with applications. *Algorithmica 3* (1988), 347–365.

[6] Balkenhol, Kurtz, and Shtarkov. Modifications of the burrows and wheeler data compression algorithm. In *DCC: Data Compression Conference* (1999), IEEE Computer Society TCC.

[7] Bieganski, P., Riedl, J., Carlis, J. V., and Retzel, E. F. Generalized suffix trees for biological sequence data: Applications and implementation. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences. Volume 5 : Biotechnology Computing* (Los Alamitos, CA, USA, Jan. 1994), L. Hunter, Ed., IEEE Computer Society Press, pp. 35–44.

[8] Chang, W. I., and Lawler, E. L. Approximate string matching in sublinear expected time. In *31st Annual Symposium on Foundations of Computer Science* (St. Louis, Missouri, 22–24 Oct. 1990), vol. I, IEEE, pp. 116–124.

[9] Chang, W. I., and Lawler, E. L. Sublinear approximate string matching and biological applications. *Algorithmica 12*, 4/5 (Oct./Nov. 1994), 327–344.

[10] Choi, Y., and Lam, T. W. Dynamic suffix tree and two-dimensional texts management. *Inf. Process. Lett. 61*, 4 (28 Feb. 1997), 213–220.

[11] Cobbs, A. L. Fast approximate matching using suffix trees. In Galil and Ukkonen [16], pp. 41–54.

[12] Cull, P., and Holloway, J. L. Optimistically building a consensus sequence using $\mathcal{F}$–inexact matches. In *Proceedings of the Hawaii International Converence on System Sciences* (1992), vol. 1, pp. 643–652.

[13] Farach, M. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science* (Miami Beach, Florida, 20–22 Oct. 1997), IEEE, pp. 137–143.

[14] Fiala, E. R., and Greene, D. H. Data compression with finite windows. *Communications of the ACM, CACM 32*, 4 (Apr. 1989), 490–505.

[15] Galil, Z., Park, J. G., and Park, K. Three-dimensional pattern matching. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures* (Newport, Rhode Island, June 22–25, 1997), SIGACT/SIGARCH and EATCS, pp. 53–62. Extended abstract.

[16] Galil, Z., and Ukkonen, E., Eds. *Combinatorial Pattern Matching, 6th Annual Symposium* (Espoo, Finland, 5-7 July 1995), vol. 937 of *Lecture Notes in Computer Science*, Springer.

[17] Giancarlo, R., and Grossi, R. On the construction of classes of suffix trees for square matrices: Algorithms and applications. In *Automata, Languages and Programming, 22nd International Colloquium* (Szeged, Hungary, 10–14 July 1995), Z. Fülöp and F. Gécseg, Eds., vol. 944 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 111–122.

[18] Giancarlo, R., and Grossi, R. Suffix tree data structures for matrices. In Apostolico and Galil [4], ch. 10, pp. 293–337.

[19] Giegerich, R., and Kurtz, S. A comparison of imperative and purely functional suffix tree constructions. *Science of Computer Programming 25*, 2–3 (Dec. 1995), 187–218.

[20] Giegerich, R., and Kurtz, S. From Ukkonen to McCreight and Weiner: a unifying view of linear-time suffix tree construction. *Algorithmica 19*, 3 (1997), 331–353.

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

[22] Hariharan, R. Optimal parallel suffix tree construction (extended abstract). In ACM [1], pp. 290–299.

[23] Hariharan, R. Optimal parallel suffix tree construction. *J. Comput. Syst. Sci. 55*, 1 (Aug. 1997), 44–69.

[24] Kärkkäinen, J. Suffix cactus: A cross between suffix tree and suffix array. In Galil and Ukkonen [16], pp. 191–204.

[25] Kitajima, J. P., and Navarro, G. A fast distributed suffix array generation algorithm. In *Proceedings of the String Processing and Information Retrieval Symposium* (Sept. 1999), IEEE, pp. 97–104.

[26] Kosaraju, S. R. Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In ACM [1], pp. 310–316.

[27] Landau, G. M., Schieber, B., and Vishkin, U. Parallel construction of a suffix tree (extended abstract). In *Automata, Languages and Programming, 14th International Colloquium* (Karlsruhe, Germany, 13–17 July 1987), T. Ottmann, Ed., vol. 267 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 314–325.

[28] Larsson, N. J. *Structures of String Matching and Data Compression.* PhD thesis, Department of Computer Science, Lund University, Sweden, Sept. 1999.

[29] Manber, U., and Myers, G. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput. 22*, 5 (Oct. 1993), 935–948.

[30] McCreight, E. M. A space-economical suffix tree construction algorithm. *Journal of the ACM 23*, 2 (Apr. 1976), 262–272.

[31] Navarro, G., Kitajima, J. P., Ribeiro-Neto, B. A., and Ziviani, N. Distributed generation of suffix arrays. In *Combinatorial Pattern Matching, 8th Annual Symposium* (Aarhus, Denmark, 30 June–2 July 1997), A. Apostolico and J. Hein, Eds., vol. 1264 of *Lecture Notes in Computer Science*, Springer, pp. 102–115.

[32] Nelson, M. R. Algorithm alley: Fast string searches with suffix trees. *Dr. Dobb's Journal of Software Tools 21*, 8 (Aug. 1996). (Available online at http://dogma.net/markn/articles/suffixt/suffixt.htm).

[33] Risvik, K. Approximate word sequence mathing over spase suffix trees. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching* (Piscataway, NJ, 1998), M. Farach-Colton, Ed., no. 1448 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 65–79.

[34] Rodeh, M., Pratt, V. R., and Even, S. Linear algorithm for data compression via string matching. *Journal of the ACM 28*, 1 (Jan. 1981), 16–24.

[35] Sagot, M.-F. Spelling approximate repeated or common motifs using a suffix tree. In *Proceedings of the 3rd Latin American Symposium* (Campinas, Brazil, 1998), C. L. Lucchesi and A. V. Moura, Eds., no. 1380 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 374–390.

[36] Ṣahinalp, S. C., and Vishkin, U. Symmetry breaking for suffix tree construction (extended abstract). In ACM [1], pp. 300–309.

[37] Sayood, K. *Introduction to Data Compression.* Morgan Kaufmann Publishers, Inc., 1996, ch. 5.4, pp. 100–105.

[38] Stoye, J., and Gusfield, D. Simple and flexible detection of contiguous repates using a suffix tree. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching* (Piscataway, NJ, 1998), M. Farach-Colton, Ed., no. 1448 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 140–152.

[39] Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEETIT: IEEE Transactions on Information Theory 39* (1993), 1647–1659.

[40] Ukkonen, E. Approximate string-matching over suffix trees. In *Combinatorial Pattern Matching, 4th Annual Symposium* (Padova, Italy, 2-4 June 1993), A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, Eds., vol. 684 of *Lecture Notes in Computer Science*, Springer, pp. 228–242.

[41] Ukkonen, E. On-line construction of suffix trees. *Algorithmica 14*, 3 (Sept. 1995), 249–260.

[42] Ukkonen, E., and Wood, D. Approximate string matching with suffix automata. *Algorithmica 10* (1993), 353–364.

[43] Weiner, P. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory* (The University of Iowa, 15–17 Oct. 1973), IEEE, pp. 1–11.