A Comprehensive Study of Suffix Tree Algorithms

Master Thesis Presented to the
Department of Computer Science (Chair V) at the
Rheinische Friedrich-Wilhelms-Universität Bonn

In Partial Fulfillment of the Requirements for the Degree of
Master of Science (M.Sc.)

Supervisor: Prof. Dr. Norbert Blum
Submitted in July 2018 by:
Ludvig Sundström
Matriculation Number: 2952155

Abstract

This thesis is an in-depth study on the construction of suffix trees. Its first part reviews and investigates the theory behind the algorithms of Ukkonen, McCreight and Farach as well as the simple top-down recursive algorithm. Its second part is concerned with implementation strategies and the various uses of suffix trees in practice. Here, two C programs are discussed in detail, one implementing McCreight's algorithm and the other implementing the top-down recursive algorithm. The programs are complete and very usable in practice; they are efficient and work for most modern text encodings such as UTF-8. In the average case, the implementation of McCreight's algorithm requires 20 bytes and and the implementation of the top-down algorithm 12 bytes per input character. Constructing a full suffix tree for a 10 megabyte sample file takes 4.3 seconds with McCreight's algorithm. The top-down algorithm is best for finding only the first occurrence of a pattern in a string. When only the first occurence is desired, the performance of the top-down algorithm is practically linear, and it outperforms McCreight's algorithm in almost every such situation.

Contents

1 Introduction

The suffix tree is, as the name suggests, a tree data structure representing the suffixes of a particular string. The edges of the tree provide this representation: each suffix of the string is spelled out by the concatenation of edge labels on a specific root to leaf path in the tree. If the string has n suffixes, then its suffix tree has precisely n such paths, each ending with a leaf identified with the starting index of the suffix it represents. This elegant exposure of string structure is the principal property of suffix trees, making them useful for solving many string-related problems[19].

The most basic use of suffix trees is matching a string "pattern" on a longer string, often referred to as "the text". Given the suffix tree for a text, one can efficiently match an arbitrary pattern string with text, simply by following the specific root-path in the suffix tree matching the pattern. If the entire pattern is matched by a path, then the indices of all occurrences of the pattern are given by the leaves in the subtree under this matched path. The time to query a pattern for all its starting positions in the text thus only depends on the length of the pattern, not the text. The full text only needs to be processed during the construction of the suffix tree, which is possible to do in both linear time and space[37][29][39][12].

Thus, Suffix tree construction implies preprocessing of a text into an indexing data structure, which can help to boost performance when querying multiple patterns in fixed texts. However, this is not the only use of suffix trees; others include finding longest common substrings, detecting structural patterns like repeats and palindromes as well as data compression[27]. The suffix tree is in some sense a "swiss army knife" for string related problems: according to Apostolico, no other indexing data structure can compete with its elegance and versatility[4].

Starting with Weiner in 1973, a four-decades-long continuous research effort has been dedicated to the suffix tree[5]. Given its theoretical importance, one would probably expect suffix trees to be a part of almost every software implementation dealing with sequential data. This is however not the case. The implementation of suffix trees is actually very limited in practice, restricted to special applications that are heavily focused on performance[1].

The hindrance to implementing suffix trees seems to be due to the handling of excessive greed for space. The algorithms need much more memory than the amount needed by storing the string — vertices, edges, edge labels and other components all need to be stored and retrieved during execution of the algorithm[26]. Additionally, most suffix tree algorithms assume constant time random access to the entire string in memory, making it impossible to consume the string in smaller parts[14]. Even if sufficient memory is available for building the suffix tree; the construction itself is far from trivial. Most algorithms are based on careful local modifications, using tricks and structural exploits, making them hard to understand. This issue has been acknowledged by multiple authors[37][13][14][33], and a desire for simplifying suffix tree construction has been a common theme over the years[5]. Currently, the best resource to learn about suffix trees are academic textbooks, which usually assume at least some background in computer science and which seldom provide sample code or experiments[26]. In short, the effort of understanding and implementing suffix trees may not always amortize[17].

This thesis introduces no new algorithms and makes no materialistic improvements to existing ones. Instead, it focuses on what exactly suffix trees are, what they are useful for and how to construct and implement them efficiently. Section 2 begins the investigation by presenting the motivation and prior work on suffix trees. Section 3 continues with the foundation, giving essential definitions used for describing the suffix tree algorithms and their applications in Section 4, 5 and 6. The second part of the thesis is concerned with the implementation and experimentation of suffix trees. The purpose of Sections 7 is to accurately describe two possible implementations of suffix tree construction algorithms, and with actual, working computer code highlight the most important details in efficient implementations. Section 8 describe a series of tests and experiments carried out with the two implementations and presents the results. Section 9 gives a few conclusions based on the study. The thesis is intended to be read sequentially, as later sections build on earlier ones.

2 History of Suffix Trees

String queries of type "Does the text t contain the pattern p?"[1] are very common and important in computer science, both in theory and practice[29]. The naive approach for answering queries of that kind is iterating through the text left to right, attempting to match the pattern with all possible alignments in the text. This method has a quadratic time in the worst case: consider for example the text *aaaaaaaaaaaaaaaaab* and pattern *aaaab* for which all characters in the pattern has to be compared with every position in the string before finally finding a match on the last alignment. Once a mismatch happens, the naive algorithm "forgets" that all characters that were compared up to the current one were equal. Acknowledging this, Knuth, Morris and Pratt in 1974 designed the *Knuth-Morris-Pratt algorithm*[25] (KMP), based on *not* forgetting the mismatch information, thereby reducing the quadratic time to linear. KMP is less efficient when querying a large number of patterns in the same text as both text and pattern need to be scanned fully for each new pattern. In this type of scenario, it would be better to scan the text once and create an auxiliary index which later could be used for fast pattern lookup.

The Suffix tree (initially called *bi-tree*) introduced by Weiner[39] in 1973, can be used as an index in the pattern matching problem above; as well as solving a myriad of other problems[4]. The *crux* lies in its construction, which is far from trivial — the last 40 years have seen numerous and continuous efforts to new and existing suffix tree algorithms, something hardly true for any other data structure[5].

Weiner's initial motivation for developing suffix trees was to be able to transmit messages in minimal space and time[29]. Soon, the versatility of the new structure was recognized, and many ideas in stringology[2] changed for good. For example, Donald Knuth had previously conjectured that finding the longest substring common to two strings of a total length of n required $O(n \log n)$ time[5]. Weiner showed that this was not only possible in $O(n)$ time but also extremely

[1] Both "the text" and "the pattern" refers to strings. Usually, the length of the text is assumed to be much longer than the length of the pattern.

[2] Stringology is a part of algorithmic research that deals with the processing of text strings.

elegant using suffix trees. Knuth later praised Weiner's algorithm to be "The algorithm of 1973"[19].

The development of suffix trees continued with McCreight in 1976[29], publishing a paper describing a new, more space-efficient algorithm for suffix tree construction. At the CPM conference in 2013, McCreight revealed that he developed his algorithm as an attempt to understand Weiner's paper. When he went to Weiner asking him to confirm that he had understood the paper the answer was "No, but you have come up with an entirely different and elegant construction!"[5].

McCreight was not the only author that considered Weiner's paper hard to understand[37][27], which perhaps was the reason that suffix trees got somewhat less attention in the years that followed. In 1992 however, Ukkonen published a highly influential paper[37] describing a new construction algorithm, having all the advantages of McCreight's algorithm but also allowing for a simpler description[19]. The two algorithms are in many ways similar. Both Ukkonen and McCreight's base their algorithms on repeatedly inserting substrings into the tree, but the former differs from the latter in that it inserts longer and longer prefixes instead of shorter and shorter suffixes of the text. Ukkonen's algorithm is, therefore, an online algorithm, where every intermediate tree during the construction phase is an actual suffix tree for the prefix considered so far. As a result, Ukkonen's algorithm does not require that the length of the string is known beforehand, which can be useful when only the first occurrence of a pattern in the text is desired. When constructing a full suffix tree, McCreight's algorithm is slightly more efficient[16].

Another similarity of McCreight's and Ukkonen's methods is the way they use different mechanics and observations for reducing time complexity. One essential mechanic is the so-called *suffix link*, providing shortcuts during construction. Suffix links together with a few other ideas give these algorithms a linear worst case time bound, assuming the number of distinct characters in the string (also called the alphabet) is constant. This number does, however, influence both the speed and space requirement of these algorithms, since most basic operations depend on finding the correct character in the current branch

of the tree[15]. Giegerich and Kurtz confirmed these observations through experiments[14]. In 1997, Farach introduced a new algorithm[12] based on recursive merging of suffix trees for subsets of suffixes. This algorithm had little in common with the prior suffix tree methods; it abandoned both the "one suffix at a time" approach and suffix links. Importantly, this algorithm is truly linear — it depends only on the text size and not the alphabet. Also, a variant of the algorithm was developed for optimal parallel construction of suffix trees in $O(\log n)$ time[13].

The main problem with implementing suffix trees for actual software is that that they tend to be very greedy for space[5]. Even though their linear characteristics seem attractive, it is not immediately clear how to store and process the tree efficiently[17]. This is specifically relevant for large-scale applications where strings of several gigabytes such as complete genomes need to be processed[1]. The space problem was repeatedly worked on by Kurtz and his colleagues, giving rise to more efficient and usable suffix tree implementations[26][17][1]. Additionally, Giegerich and Kurtz were the first to consider suffix tree construction from the perspective of functional programming[14][15][27], resulting in many new insights and in addition a new simple *lazy* suffix tree algorithm, which can be very useful in certain practical situations[14].

In 1990, Manber and Myers took a different approach to the space problem when they developed a completely new data structure called a *suffix array*[28], eliminating most of the structure of the suffix tree, thus reducing the rquirement for space considerably but at the same time implying disadvantages in versatility, rendering suffix arrays applicable only to string matching problems. Additionally, a smaller term of $\log(|t|)$ was added to the search complexity, where t is the text[26]. Suffix arrays are constructed in linear time given a suffix tree for the same string[23], and thus suffix arrays are just compact representations of suffix trees that imply some loss of information. Despite their limitations, suffix arrays are far more popular in modern software than suffix trees[5]. From 2000 and onwards, there has been many efficient implementations of suffix arrays, most notably by Nong, Zhang and Chan[31].

3 Basic Notation and Definitions

This section lay the foundation for Section 4, 5 and 6. The only assumption is some familiarity with basic set theory.

3.1 Strings

Let Σ be a finite set called *the alphabet* with elements called *characters* ordered by a total order[3]. A *string* is a list of characters from Σ. Σ^* denotes all possible strings from Σ, including the empty string ϵ of zero length. In general, the length of a string t is denoted $|t|$, the number of characters in t. The i'th character of t is denoted $t[i]$ for the integer index i, $1 \leq i \leq |t|$. Indices start at 1, $t[1]$ is the first character of t and $t[|t|]$ the last. A *substring* longer than just a single character of t is denoted using two indices i and j, $1 \leq i \leq j \leq |t|$, by $t[i:j]$, i.e the string $t' = t[i], t[i+1], \ldots, t[j]$. A *prefix* of t is a substring where $i = 1$, i.e $t[1:j]$ for some $1 \leq j \leq |t|$. Similarly, a suffix of t is a substring where $j = |t|$, i.e. $t[i:|t|]$ for some $1 \leq i \leq |t|$. A prefix p of t is written $p \sqsubset t$ and a suffix s of t written $t \int s$. A substring of t is called *proper* if it is different from t. To join two strings end to end is called concatenation; the concatenation of the two strings x and y is written xy.

3.2 Graphs

A *graph* is a pair $G = (V(G), E(G))$ where $V(G)$ is a finite set called the *vertices* of the graph. $E(G)$ is called *the edges* of the graph, a set whose elements represents relations between vertices. The graph is called *directed* (a digraph) if $E(G) \subseteq V(G) \times V(G)$. Here, an edge e is an ordered tuple (v, w), where v is called the *origin* and w is called the *endpoint* of e. The graph is called *undirected* if $E(G) \subseteq \{X \subseteq V(G) : |X| = 2\}$. Here an edge is a two-element set $\{v, w\}$ with no particular order. Only *simple graphs* are considered in this thesis, that is, graphs that do not allow for $E(G)$ to contain identical elements.

[3] In this thesis, the characters will normally be sampled from the ordinary English alphabet, such as $a, b, c \ldots$. These are ordered by standard dictionary order also called lexicographical ordering.

For a directed graph G, its *underlying undirected graph* is the undirected graph on the same vertex set as G which contains an edge $\{v, w\}$ for each edge $(v, w) \in E(G)$. A *subgraph* of G is a graph H where $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. If H is a subgraph of G, G is said to *contain* H.

Given graph G, two vertices v, w are called *neighbors* if $\{v, w\} \in E(G)$ (if G is directed, then this is true for the underlying undirected graph of G). If G is directed, it is distinguished between *incoming* edges of v (the edges $(w, v) \in E(G)$) and *outgoing* edges of v (the edges $(v, w) \in E(G)$). The incoming edges are denoted $\delta^-(v)$ and the outgoing edges are denoted $\delta^+(v)$. For undirected graphs, all adjacent edges (the edges $\{v, w\} \in E(G)$) to v are all denoted by $\delta(v)$.

In a graph G, a *walk* W is a sequence $v_1, e_1, v_2, \ldots, v_k, e_k, v_{k+1}$ such that $e_i = (v_i, v_{i+1}) \in E(G)$ for $i = 1, \ldots, k$ and $e_i \neq e_j$ for all $1 \leq i < j \leq k$. W is closed if $v_1 = v_{k+1}$. A *path* is a graph $P = (\{v_1, \ldots, v_{k+1}\}, \{e_1, \ldots, e_k\})$ such that $v_i \neq v_j$ for $1 \leq i < j \leq k + 1$ and the sequence $v_1, e_1, v_2, \ldots, v_k, e_k, v_{k+1}$ is a walk. P is called a *path* from v_1 to v_{k+1} or a v_1-v_{k+1}-path. A *cycle* is a graph $C = (\{v_1, \ldots, v_{k+1}\{, \{e_1, \ldots, e_k\})$ such that the sequence $v_1, e_1, v_2, \ldots, v_k, e_k, v_1$ is a closed walk and $v_i \neq v_j$ for $1 \leq i < j \leq k$.

3.3 Trees

Let G be an undirected graph. If there is a v-w-path for all $v, w \in V(G)$, then G is a *connected* graph. An undirected graph without a cycle as its subgraph is called a *forest*, and a connected forest is called a *tree*. A digraph is connected if its underlying undirected graph is connected. Moreover, a digraph is called a *directed forest* if the underlying undirected graph is a forest and each vertex v has at most one incoming edge. If in addition the directed forest is connected, it is called a *directed tree*. A directed tree with n vertices has $n-1$ edges[8], and thus exactly one vertex with $\delta^-(r) = \emptyset$. This vertex is called the *root*, and the directed tree is said to be rooted at r. The vertices v for which $\delta^+(v) = \emptyset$ are called *leaves* and the vertices that are neither root nor leaves are called *inner vertices*.

The rooted directed tree impose a hierarchy on its vertices: for some v-w-*path* P in a rooted directed tree T, every vertex $u \in V(P) \setminus \{w\}$ is called an

ancestor of w, and every vertex $u' \in V(P) \setminus \{v\}$ is called a *descendant* of v. A descendant also a neighbor with v is called a *child* of v and the ancestor also a neighbor with w is called the *parent* of w.

3.4 Labeled Trees

A labeled tree is a tree where each edge is associated with a particular string, its label. The relation *label* is defined to retrieve a label, given an edge. If $e = (v, w)$ is an edge and $label(e) = x$, e is denoted $v \xrightarrow{x} w$. If the first character of x is a, then e is called an a-edge. By concatenating every edge label along a particular u-v-path in the tree, one obtains the *label path* between vertices u and v, the string spelled out by the path. Most often, the label path from the root to a vertex w is used, in which case the label path can be obtained by the relation $lp(w)$. For another start vertex than the root, say w', one applies the overloaded version $lp(w', w)$.

3.5 Modifying a Tree

Only three procedures are used for modifying a rooted directed labeled tree T, these are listed below.

- For contracting two edges $v \xrightarrow{x} w$ and $w \xrightarrow{y} u$, the procedure *contract* is used. By invoking $contract(v \xrightarrow{x} w, w \xrightarrow{y} u)$, the two vertices are contracted into $v \xrightarrow{xy} u$. *contract* is well-defined as long as $|\delta^+(w)| = 1$.

- For splitting an edge $v \xrightarrow{x} u$ into two edges, the procedure *split* is used. By invoking $split(v \xrightarrow{x} u, r)$, the vertex is split into the two edges $v \xrightarrow{x[1:r]} w \xrightarrow{x[r+1:|x|]} u$: a new vertex w is inserted that becomes the child of the first and the parent of the second edge. The original edge label is split after index r. *split* is well defined as long as the edge $|x| > 1$ and $1 \leq r \leq |x| - 1$.

- To create a new edge, the procedure *add* is used. By invoking $add(v, x)$, $v \xrightarrow{x} w$ is added to the tree where w is a new leaf vertex, v is an existing vertex in T and $x \in \Sigma^+$.

3.6 Compact Tries

Suppose a set S of n pairwise distinct strings is given. A *compact trie* T on S is a rooted directed labeled tree such that:

1. T contains exactly n leaves numbered 1 to n. The relation *number* is defined for retrieving these numbers, given a leaf.
2. The edges of T are labeled as in Section 3.4 with strings from Σ^+.
3. Any vertex in T, has at most one outgoing a-edge for each $a \in \Sigma$.
4. The $lp(v)$ for some leaf v with $number(v) = i$ in the trie exactly spell out the i'th element of S.
5. Every inner vertex has at least two children. The root is not considered an inner vertex, so it is allowed to have only one child.

3.7 Suffix Trees

Let t be a string of length n. A suffix tree for the string t, denoted $ST(t)$, is just a compact trie for the set of suffixes of t. Thus each leaf directly corresponds to a particular suffix of t: given leaf v in $ST(t)$ with $number(v) = i$, $lp(v) = t[i : n]$. This property grants suffix trees their power and usefulness: any substring of t also exist as a prefix of the $lp(v)$ for at least one vertex $v \in V(ST(t))$, and can thus be matched character by character on such a path, starting from the root.

Since the outgoing edges of each vertex are unique, the full $lp(v)$ is also unique for every $v \in V(ST(t))$. This allows for identifying a vertex v with $\bar{x}$ if and only if $lp(v) = x$, which is convenient since the vertices often are considered in parallel to the string they represent in the suffix tree.

3.7.1 The Sentinel Character

The problem with defining suffix trees as compact tries for the suffixes of a string is that it doesn't guarantee that a suffix tree exists for every string. The reason is that if one suffix is a prefix of another suffix, both can not end on a leaf, so the "suffix tree" for such a string would result in a tree with fewer than n leaves. For example, if $t = cabca$, then $ST(t)$ would contain $cabca$ as a path, so the suffix ca cannot end with a leaf.

The workaround to this problem is to add another character denoted \$ such that $\$ \notin \Sigma$, to the end of the string. Then all suffixes of the string are unique and $ST(t\$)$ thus contains exactly $|t| + 1$ leaves, one for each suffix of t and one extra for the "sentinel character".

3.7.2 Implicit Suffix Trees

An *implicit suffix tree* is obtained from $ST(t\$)$ by removing the symbol \$ from every edge label in the tree that contains it, deleting any edges with empty labels and then calling *contract* on the tree, as long as *contract* is well defined for two edges in the tree. An implicit suffix tree does thus not contain any branches corresponding to sentinel characters, that is, less unnecessary branches if the only motive is to pattern match on the suffix tree. However, removing these branches could invalidate the structure of the tree, so that problems like finding the longest common prefix of two strings can no longer be solved using the suffix tree.

3.7.3 Generalized Suffix Trees

A generalized suffix tree is a suffix tree representing more than one string. If S is a set of strings, then $GST(S)$ is the generalized suffix tree constructed from S to represent the suffixes of each $s \in S$. An easy way of constructing a generalized suffix tree is to append a unique sentinel character to each $s \in S$ and then concatenate those strings into a new string like $s_1\$_1 s_2\$_2 \ldots \$_3 s_n$ where $\$_1, \$_2$ and $\$_3$ are unique sentinel characters, and construct a suffix tree from that string. The resulting suffix tree gets a leaf for each suffix in each $s \in S$ and is built in linear time proportional to the length of t. During construction, the algorithm can easily add a second number i to each leaf, identifying the index of the string in S the suffix originates from. For an illustration, see figure 3.7.4.

One problem with this approach is that the tree will represent substrings that span more than one element in S. However since the end of every string is marked with a unique string marker, a *lp(v)* for some inner node v cannot span two strings. If it did it would imply that there would be at least two different characters after a sentinel character, so that sentinel character would in fact not be unique. Therefore, any sentinel character can only appear on a leaf edge in

the generalized suffix tree. Since none of the sentinel characters occur in the original strings, one obtains an implicit general suffix tree for S by following the same procedure as in Section 3.7.2, but instead of just deleting the sentinel character on the label for a leaf, one deletes the first sentinel character of the leaf edge, and every character to the right of it.

3.7.4 Matching a Pattern in a Suffix Tree

A pattern p can be queried for existence in the text t in the following way. Choose the edge $e = r \xrightarrow{x} v$ originating at the root such that $x[1] = p[1]$. Proceed with comparing $x[i]$ with $p[i]$ for $i = 2, 3, \ldots$ until $i = |x|$. At this point, the full edge e is "matched" and the new a-edge is chosen under v such that $a = p[|x| + 1]$. This procedure is repeated until one of the following cases arises.

(*i*) At any point, $x[i] \neq p[l + i]$, where x is the current label, $1 \leq i \leq |x|$ and l is the combined length of the labels successfully matched so far.

(*ii*) The current edge $e = v \xrightarrow{x} y$ is fully matched but p is not fully matched and w has no $p[|x| + 1]$-edge.

(*iii*) p is fully matched.

In case (*i*) and (*ii*), p does not occur in the suffix tree and therefore also not in t. In case (*iii*), suppose the current edge is $e = (v, w)$. Then every leaf vertex in the subtree rooted at w contains an index i such that $t[i : |p|] = p$.

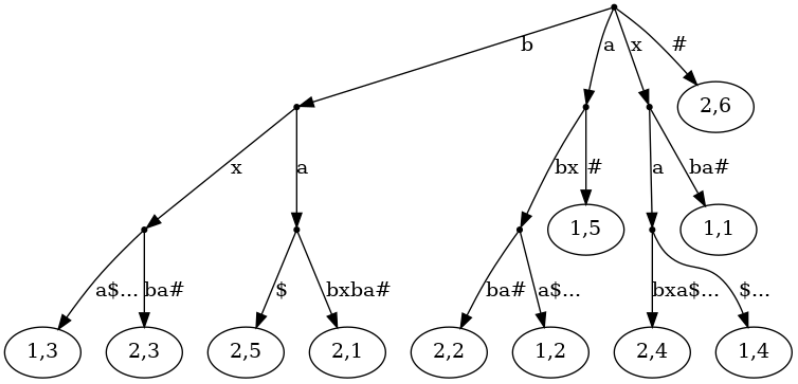

Figure 1: Generalized suffix tree for the strings $x_1\$x_2\#$, where $x_1 = xabxa$ and $x_2 = babxba$, with the two sentinel characters \$ and #. The leaves are marked with integer pairs (i, j) where i corresponds to x_i and j is the start position of the suffix in x_i. The elipsis denotes a remainder of a suffix which spans more than one of the original strings. This can be deleted with no loss of information.

4 Suffix Tree Applications

There are many texts on applications on suffix trees. Gusfield devotes over 70 pages of his book[19] to the applications of suffix trees and other books on the subject include Crochemore[9], Crochemore and Rytter[11] as well as Crochemore, Hancart and Lecroq[10]. Even though it is not the main purpose of this thesis, this section describes a selection of simple and elegant applications of suffix trees to different problems, inspired by[19].

4.1 Exact Substring Matching

THE SUBSTRING SEARCH PROBLEM is the problem of searching a text for the occurrence of shorter patterns. The standard version of the problem is solved using suffix trees, as discussed in Section 3.7.4. Suffix trees are most effective for this problem when matching a large number of patterns on the same text.

Another version of the problem is when given a set of strings S (called the database) and a query string x. The task is to find all strings in the database that contains x. This problem can be solved using suffix trees by building a generalized suffix tree on S as described in Section 3.7.3, using linear space time and space in the combined length of the strings of the database. In a generalized suffix tree, each leaf contains an index for which string it corresponds to in S. If p is the pattern matched, one readily obtains all strings containing p by traversing the subtree under the matched path.

4.2 The Longest Common Substring

THE LONGEST COMMON SUBSTRING PROBLEM for two strings can be solved using suffix trees by constructing a generalized suffix tree for the two strings. Each path in the tree corresponds either to a substring the first, the second or both strings. By marking each inner vertex with an 'A' or 'B' depending on which substring it corresponds to, one can naturally find longest common substring by traversing the tree for the inner vertex with the longest string depth marked with both 'A' and 'B'. This approach can be extended to solve variants of the problem, such as finding all common substrings of length at least l. In that case, the above algorithm needs to report each inner vertices v marked

with 'A' and 'B' such that $|\textit{label-path}(v)| \geq l$.

4.3 The Longest Common Substring of Many Strings

A generalization of the previous problem is called THE K-COMMON SUBSTRING PROBLEM and is defined as follows. Given a set S of distinct strings and an integer k, find the length of the longest substring common to at least k elements in S. Suppose the number of strings $|S| = m$ and their combined length $\sum_{s \in S} |s| = n$. Define $lcs(i)$ to be the longest common substring to at least i strings in S. The problem is solved incrementally, by computing all k values for lcs, which in fact can be done in linear time using suffix trees[19]. Here a simpler, $O(mn)$ algorithm is presented for this problem (also relying on suffix trees). Assume the generalized suffix tree has been constructed for S. For each inner vertex $v \in GST(S)$, define $c(v)$ to be the number of (distinct) sentinel characters for the subtree rooted at v. Once $c(v)$ is known for each $v \in GST(S)$, the $lcs(i)$ values can easily be found with a linear time traversal of the tree, accumulating a list of successive values for lcs. To compute $c(v)$ for each inner vertex v, assign to it a bit vector of length k where bit i is set to one if and only if the subtree of v has a leaf with sentinel character i. Then, $c(v)$ is the sum of ones in this vector. The vectors are computed bottom-up by the bitwise **OR** operation on the vectors of the children. Since the tree has $O(n)$ edges, the time needed to build the entire table is $O(mn)$.

4.4 All Pairs of Suffix-Prefix Matches

A suffix-prefix match of two strings x and y is a suffix of x that matches a prefix of y. The ALL PAIRS SUFFIX-PREFIX MATCH PROBLEM is defined as follows. Let S be a set of strings and s_i be the i'th element of S. Furthermore, let $m = |S|$ and $n = \sum_{s \in S} |s|$. The all-pairs suffix prefix problem is the problem of finding, for each pair x, y from S the longest suffix-prefix match of x, y.

This problem can be solved by building a similar structure as the auxiliary index used by KMP[25]. This structure is an array whose elements represent the length of the longest proper prefix that is also a proper suffix of $p[1 : i]$ for some index i. The longest prefix-suffix pair of two strings x and y is thus found by applying the KMP preprocessing to xy. Using this technique on the

m^2 string pairs separately yields a total bound of $O(mn)$ because each string has length $O(n/m)$.

With suffix trees, one can reduce the bound to $O(n + m^2)$. The algorithm proceeds as follows: Construct $GST(S)$, and at the same time generate a list $l(v)$ for each inner vertex v, containing the index i if and only if v has a child vertex w such that w is a leaf and $s_i \int \mathit{label}(w)$. For example, let $v = ba$ as in figure 3.7.4, then, $l(v) = \{2\}$.

Now consider a fixed leaf $\overline{s_j}$ in the tree. The key observation is the following: If v is a vertex on the root path to $\overline{s_j}$ in the generalized suffix tree, and $i \in l(v)$, then both $s_i \int \mathit{label\text{-}path}(v)$ and $\mathit{label\text{-}path}(v) \sqsubset s_j$ holds. So for each index i, the deepest vertex v on the path to $\overline{s_j}$ such that $i \in l(v)$ identifies the longest match between a suffix of s and a prefix of s'.

By traversing $st(S)$ using depth-first search, the algorithm can collect the suffix-prefix matches by maintaining m stacks, one for each string. When encountering vertex v, the algorithm scans each $i \in l(v)$, and pushes v onto the stack i. When the algorithm reaches a leaf with number j, it inspects the top elements of every stack. The vertex on top of stack i represents the longest suffix-prefix match of the pair (s_i, s_j). When the depth-first traversal backtracks over vertex v, pop the top element of any stack whose index is in $l(v)$.

Since the total number of indices in the lists l are $O(n)$, and the number of edges is $O(n)$, traversing $st(S)$ and updating the stacks takes $O(n)$ time. Recording each of the $O(m^2)$ answers take constant time, so the time required by the algorithm is $O(n + m^2)$.

5 Imperative Algorithms

This section presents Ukkonen's and McCreight's suffix tree construction algorithms in detail. They are imperative by nature since they rely on local mutation, pointers, and random access into the text to achieve efficiency. The description of Ukkonen's algorithm is based on[19] and the description of McCreight's algorithm is based on[9].

First, some new terminology is needed. Let $TR(t)$ be a compact trie on the string t. Matching a string xz along a path in $TR(t)$ is essential for these algorithms. The matching starts on an edge (which does not have to be a root edge) and ends on an edge, say $e = \overline{x} \xrightarrow{y} \overline{xy}$ such that z is a prefix of y with $z[1:l] = y[1:l]$ for some index i. The tuple (e, l) is called a *location* in $TR(t)$. If $l = |z|$, then the entire xz is matched and is thus said to *occur* in $TR(t)$. If not, then there exists no *continuation* for z and the pattern does not occur in the tree. If $l = |y|$ on the last edge, then xz is said to end on a vertex, otherwise on an edge. The location in $TR(t)$ where a string ends is also called *the end* of that string.

5.1 Ukkonen's Suffix Tree Algorithm

Ukkonen's algorithm[37], henceforth referred to as *Ukk*, constructs the suffix tree for the string t of length n. The algorithm is an online algorithm, meaning that it constructs actual suffix trees longer and longer prefixes of the text, one index at a time until finally the full suffix tree is constructed. The gist of the algorithm is quite simple, but how it achieves its goal in linear time requires more effort to see. The approach here is first to describe the algorithm using simple ideas, then gradually introduce mechanisms and eventually show the linear worst case.

5.1.1 High Level Description

The algorithm is divided into n phases, where phase number i creates the suffix tree for the prefix $t[1:i]$, denoted $ST_i(t)$. Furthermore, each phase is divided into i steps, also called extensions. In phase $i+1$ and step j, the objective of

the algorithm is to find the end of the root path labeled $t[j:i]$ in the current tree. If needed, it extends the tree for that path by adding the character $t[i+1]$, thus assuring that the string $t[j:i+1]$ exists in the tree. Then, it continues with the same procedure finding the end of the path $t[j+1:i]$, until $j=i$, after which $ST_{i+1}(t)$ is finished. The first tree $ST_1(t)$ representing the suffix tree for $t[1]$ is trivially constructed, consisting simply of a single edge with label $t[1]$.

5.1.2 Extension Rules

To turn the high-level description into an algorithm, it needs to be specified how to extend a path. Suppose the algorithm is currently in phase $i+1$ and step j, then the string $t[j:i]$ needs to be matched. Let $a = t[i+1]$, the new character that may be used to extend the tree, and $xz = t[j:i]$ such that $z \neq \epsilon$ and $\overline{x}$ is a vertex in $ST_i(t)$ and $|lp(\overline{x})|$ is maximal over all possible choices of $\overline{x}$. There are four cases to consider, each using the procedures from Section 3.5 to extend the tree.

(i) If $\overline{xz}$ is a leaf in $ST_i(t)$, then no new vertices or edges should be inserted. Let e be that leaf edge whose label is z. Extend the tree by setting $label(e) := za$.

(ii) If $\overline{xz}$ is an inner vertex with no outgoing a-edge, then a new outgoing a-edge needs to be added. Use $add(\overline{xz}, a)$.

(iii) If $\overline{xz}$ is not a vertex in $ST(t)$, then xz ends on an edge, and z is a prefix of the label y on $\overline{x} \xrightarrow{y} \overline{xy}$. If in addition $y[|z|+1] \neq a$, then first a new inner vertex under z and then a new leaf edge from that inner vertex needs to be inserted. First use $split(\overline{x} \xrightarrow{y} \overline{xy}, |z|)$ and then $add(\overline{xz}, a)$.

(iv) If none of the previous cases holds, then the string xza is already represented by the tree, and nothing has to be done.

5.1.3 The Agent

Each extension takes constant time to perform once the end of the string xz has been found. The key of the algorithm is to efficiently finding the end of the current string in every phase and step.

The successive ends between the steps can be at completely different locations of the tree, so to apply the extension rules the algorithm has to move between these locations. It is convenient to think about this movement of the algorithm in terms of an entity called *the agent*. The simplest agent always starts at the root when searching for a new end, matching the string character by character, as when searching for a pattern in the finished suffix tree. This process is known as *scanning*, and implemented with the procedure *scan*. $scan(v, x)$ takes starting vertex v and a string x, and attempts to match each character under v with x, taking linear time in $|x|$. If *scan* was the only way of agent traversal in *Ukk*, $ST_i(t)$ could be created in $O(i^2)$ time, resulting in a $O(n^3)$ worst case.

5.1.4 Suffix Links

The first idea for reducing the way traveled by the agent is suffix links, a well-known concept also used in the algorithms of Weiner and McCreight. Suffix links provide shortcuts between vertices, a direct link which the agent can use to get close to the end for the next substring. This vertex does not have to be the nearest vertex, but it will be near enough to reduce the overall time bound.

Suffix links are defined by the function *link* as follows: Let $\overline{ax}$ be an inner vertex in $ST(t)$, where a is a single character. If there is another vertex $\overline{x}$ in $ST(t)$, then $link(\overline{ax}) = \overline{x}$, if not then $link(\overline{ax})$ is undefined. Furthermore if r is the root vertex then $link(\overline{a}) = r$ for any $a \in \Sigma$, but $link(r)$ is undefined.

With the above information only, it is unclear if *link* is well-defined for every inner vertex in a suffix tree. Below it is shown that this, in fact, will be the case. After that, it is shown that the nature of *Ukk* allows for suffix links to be wholly defined and used during construction in constant time.

Lemma 5.1.1. *If a new inner vertex $\overline{ax}$ is added to the current tree in extension j of phase $i+1$ then either $\overline{x}$ is already an inner vertex of the current tree or $\overline{x}$ will be created in extension $j+1$ in the same phase.*

Proof. Assume a new inner vertex v is created in phase $i+1$ and step j. Then, the algorithm applied extension rule (*iii*) for extending the path $t[j : i+1] = cx$, $c = t[j]$. Thus, the path labeled cx continued with some other character than $t[i+1]$, say a, so in the next extension $j+1$, the end of path x certainly has a continuation with a (Since cx continued with a, x has to do that as well). First, consider the case where the path x continues only with the character a. Then the end is on an edge and extension rule (*iii*) will create a new inner node at $\overline{x} = \mathit{link}(\overline{xa})$ in this phase. On the other hand, if $\overline{x}$ continues with multiple characters, then there already exists a inner vertex $\overline{x} = \mathit{link}(\overline{xa})$, because only in vertices can the path continue in more than one character. In both cases, the suffix link for phase j can be set by the algorithm in phase $j+1$. □

Corollary 5.1.2. *For every newly created inner vertex v, $\mathit{link}(v)$ will be correctly defined by the end of the next extension during execution of Ukk.*

Proof. The claim is true for the first tree $ST_1(t)$ since it contains no inner vertices. Suppose the claim is true for the i'th tree $ST_i(t)$, and consider phase $i+1$. By Lemma 5.1.1, when a new vertex v is created in extension j, the correct vertex $\mathit{link}(v)$ ending the suffix link from v will be found or created in extension $j+1$. The last phase corresponds to the single character suffix $t[i+1]$ which only can create a leaf but no inner vertex. Thus all suffix links from inner vertices created in phase $i+1$ are known of by the end of the phase, and can easily be updated accordingly for the previous phase. Thus, any implicit suffix tree $ST_i(t)$ will have access to all suffix links. □

5.1.5 Using Suffix Links

In phase $i+1$ and extension j, the agent has to locate the suffix $t[j..i]$ of the prefix $t[1..i]$. This section describes how to actually use the suffix links that are added by the algorithm. The first extension ($j = 1$) is special. The rest of the extensions are similar, but there is a minor difference between extension $j = 2$ and the rest where $j > 2$. The three cases are distinguished below.

- $j = 1$. Since the full prefix $t[1 : i]$ is the longest string in the current tree, it is represented by a leaf. The end of this path can be found at the

beginning of each phase by maintaining a pointer to this leaf during the algorithm. If $t[1:i]$ in the next phase points to a different leaf, this is easy to detect and update.

- $j = 2$. Consider the edge e entering leaf number 1, i.e. $e = v \xrightarrow{y} u$ with $number(u) = 1$. The agent must now find the end of the string $t[2:i] = x$. If v is the root, then to find the end of x the agent just has to follow the root path along x as in the naive case. If v is an inner vertex, then by Corollary 5.1.2, $link(v) = v'$ is defined, and $lp(v')$ is a proper prefix of x. It is proper, since the end of $t[1:i]$ is on the leaf u and not on v. Thus, the agent first needs to scan "backward" along e to v, and then jump to v'. Since $lp(v')$ is a prefix of x, x must end in the subtree below v'. Consequently, by following the suffix link, the algorithm can start its search from v' instead of the root.

  The backward scan to v passes a prefix of e's label, call this z. Exactly the string z must be scanned downwards again when arriving at v', to get the correct end. This scan may span more than a single edge, as discussed in the next section. Finally, extend the tree with the character $t[i+1]$.

- $j > 2$. The same idea applies as for $j = 2$. The minor difference is that $t[j-1:i]$ may end on a vertex, in which case no backward scan is needed and also no forward scan once the agent arrives at v'.

The use of suffix links allows for the agent to make "shortcuts", but it is not yet clear if it reduces the worst case bound. More ideas are needed: the following section introduces a new procedure that reduces the time bound to $O(n^2)$.

5.1.6 The Skip/Count Procedure

The previous section explained on how to find the end of $t[j..i]$ starting at $t[j-1:i]$ using suffix links. The same notation is used here, v being the closest vertex above the current end of $t[j-1:i]$ in the current tree. The downwards scan along z as discussed in case $j = 2$ takes time proportional to $|z|$. Since the algorithm knows that z occurs under v', a new traversal method *skip-count*(v', z) can be used, computing the same result as *scan*, but takes time proportional in

the number of vertices in the path matching z under v', not in the length of z. The *skip-count* procedure does not compare every single character to follow this path. Instead, it is enough to inspect the first character of each edge on the path matching z, then "skipping" over the edge to the next vertex as long as the total length of the *label path* traversed under v' is shorter than z.

The procedure is quite straightforward. Let v' be the suffix link for v as before. Because $ST(t)$ is a suffix tree, v' has exactly one edge starting with $z[1]$, let it be denoted $e = v' \to u$. Three additional variables are used, namely $q' := |label(e)|$, $q := |z|$ and h. If $q' < q$, the algorithm can skip over e, but it needs to update $q := q - q'$ and $h = q' + 1$. Now, at the vertex u, find the edge whose character matches $p[h]$, and set q' to the length of that edge. Continue these steps, updating q, q' and h accordingly until $q' \geq q$. At this point the procedure skips until character q on the current edge and quits. Then, z has been matched in a forward-scan proportional to the number of vertices in the path matching z in $ST(t)$.

To be able to reason about *skip-count*, it is useful to think about how the number of vertices contained by the root path changes as the agent changes position in the tree. Hence, let the $depth(v)$ be the number of vertices on the root path to v in $ST(t)$. The following lemma shows that the agent depth does not change too much when following suffix links.

Lemma 5.1.3. *When the suffix link $v' = link(v)$ is traversed by the algorithm for some vertex v, the node depth of v is at most one greater than v'.*

Proof. Consider a specific time during the algorithm where $v' = link(v)$ is followed for some v. Let $u = \overline{ax}$ be an ancestor of v (a being a single character and x a string) that is also an inner vertex. u has by Corollary 5.1.2 a suffix link $\overline{x}$. If $x \neq \epsilon$, then $\overline{x}$ is also an inner vertex. Since u is a ancestor to v, ax is a prefix of $path(v)$, implying that a suffix link from any inner ancestor of v goes to an ancestor of v'. The depth of two ancestors of v must be different. This fact together with the previous clause implies that each ancestor of v has

a suffix link to a distinct ancestor of v'. The only extra ancestor that v can have, without a corresponding ancestor of v' is an inner ancestor whose path label has length 1 (the single character a). The conclusion is that v can have node depth at most one more than v'. □

Theorem 5.1.4. *Using the skip/count trick, a single phase of Ukk takes $O(n)$ time.*

Proof. There are never more than n extensions in one phase. In a given extension, the agent could: do a backward-scan at most one edge, follow a suffix link, do a forward-scan, apply an extension rule and add a suffix link. All of these operations are constant, except for maybe the scans. The backward-scan decreases the current depth by at most one and by Lemma 5.1.3, each suffix link traversal decreases the depth by at most another one. Each edge traversed during the skip/count traverse increases the depth by one. Thus over the entire phase, the current depth is decremented at most $2n$ times, and since no node can have greater depth than n, the total possible increment to current depth is bounded by $3n$ over the entire phase. It follows that the total number of edge traversals during the forward-scan is bounded by $3n$, and since the algorithm uses the skip/count trick, the total time for all scanning is $O(n)$. □

Corollary 5.1.5. *Ukk implemented with suffix links and skip/count run in $O(n^2)$ time.*

The corollary is the result of crudely multiplying the time for a single phase with the total number of phases. Even though this is an improvement over the initial bound, it is still far away from the desired result. However, just a few more observations on the inner workings of the algorithm turns out to be enough to prove a linear time bound.

5.1.7 Edge Label Compression

Before making the last observations, a word is given on how the algorithm represents edges and edge labels. Since one of the critical problems suffix trees

is that they often need to be kept as a whole in memory (recall Section 2), it would be highly unpractical to store each label as a string in the current edge. *Ukk* instead stores each label in constant space by using a pair of integers denoting the start and end indices of a particular label in t. The algorithm maintains a reference to the full text t during execution, so assuming constant time random access into the string, the edge labels can in addition be retrieved in constant time. Since the numbers of edges are at most $2n - 1$, a suffix tree with this labeling scheme requires $O(n)$ space in total. This fact makes it more reasonable to believe a suffix tree can be built in linear time (as otherwise, the algorithm would not have time to spell out all labels on the edges).

5.1.8 Two Observations for Linear Runtime

The following two observations provide the last bits for proving a linear worst case bound.

O1. In any phase, if extension rule (iv) applies, it will also apply to the following steps in that phase. This is true, because if rule $(iv$ is applied, some earlier phase has ensured that $t[j : i + 1]$ was in the tree, and thus the same phase also ensured that $t[j + 1 : i + 1]$ was in the tree, and so on for all the suffixes of the current phase. Thus, once (iv) applies, the algorithm can skip directly to the next phase.

O2. If *Ukk* spawns a leaf edge as a result of rule (ii) or (iii), the created leaf will remain a leaf for the rest of the construction process. To see that this is true, note that both rule (ii) and (iii) only can create new leaf edges if the origin is an inner vertex or the root.

Now consider the start of phase $i + 1$, before any operations have been made on $ST_i(t)$. The algorithm needs to make sure that each suffix of $t[1 : i + 1]$ exists in the current tree. All suffixes for $t[1 : i]$ already exist in the tree, and the leaves represent the end of a particular set of suffixes. Taking "once a leaf, always a leaf" into account, these suffixes will now instead end with $t[i+1]$. Thus for every leaf represented by these suffixes, the algorithm applies rule (i) before any other extension rule. Also, if (ii) or (iii) is applied in step j of phase $i + 1$, then (i) will not be applied

for the remainder of that phase: since $t[j : i + 1]$ did not end on a leaf, $t[j + 1 : i + 1]$ cannot end on a leaf.

Altogether this means that, if for any suffix tree $ST_i(t)$, rule (i) was applied k times and then rule (ii) and (iii) l times, then rule (i) will be applied $k + l$ times during the construction of ST_{i+1}.

Both observations suggest a set of extensions that the algorithm does not need to perform explicitly. For O1, it is the following extensions in a given phase after rule (iv) is applied. For O2, it is extending all leaf edges that existed in previous suffix tree with rule (i), Ukkonen implemented this idea with having "open" edges labeled with the integer pair (i, l), l being the (successively growing) length of the current prefix. These extensions will be referred to as *implicit extensions* and can clearly be implemented to be performed automatically by the algorithm without asymptotically influencing the worst case bound.

5.1.9 The Final Result

Theorem 5.1.6. *Using suffix links, skip/count and the two observations the previous section, Ukk can construct the implicit suffix trees $ST_i(t)$ through $ST_n(t)$ in $O(n)$ total time.*

Proof. The time for all implicit extensions in any phase is constant and thus takes $O(n)$ time total. As the algorithm executes explicit extensions, consider a phase i and extension j. Because of observation 2 of the previous section, and implicit extensions, j never needs to decrease during the entire execution of the algorithm. Since there are n phases, and j is bounded by n, the algorithm executes $2n$ explicit extensions. The time for an explicit extension is constant, plus the time proportional to the number of nodes traversed during the *skip-count* processes.

For a bound on the total number of *skip-count* nodes, consider how the current vertex depth changes during successive extensions. Notice that the first explicit extension in any phase begins with extension j, which was the last explicit extension of the previous phase. Then, with an identical argument as in Theorem 5.1.4, it follows that the maximum number of vertex skips during

all forward-scanning is bounded by $O(n)$. Because every $ST_i(t)$ is a suffix tree for the current prefix considered so far, $ST(t)$ is indeed an implicit suffix tree.

5.2 McCreight's Suffix Tree Algorithm

McCreight's algorithm[29], henceforth referred to as *Mcc*, constructs the suffix tree the string t with $t[|t|] = \$$ of length n. The high level description *Mcc* differs quite considerably from *Ukk*, but the construction processes share many ideas. As pointed out by Giegerich and Kurtz, *Ukk* is actually just a disguised version of *Mcc* and vice versa[15].

5.2.1 High Level Description

Mcc is split into n phases, one for each suffix. Starting with $TR_1(t)$, the tree consisting of a single edge labeled by the entire string t, *Mcc* now inserts shorter and shorter suffixes into the tree, yielding the series $TR_1(t), TR_2(t) \dots TR_n(t)$ of trees. To insert a suffix, the prefix of that suffix is matched in the current tree. At the location for which it cannot be matched anymore a (possibly new) vertex is inserted from which a new leaf edge labeled by the rest of the suffix is attached. Because t ends with \$, every phase creates a new leaf, and for the same reason only $TR_n(t) = ST(t)$ is an actual suffix tree in the series.

5.2.2 Head and Tail

In phase i, all suffixes $t[k : n]$ for $1 \leq k \leq i$ occurs in the tree. Now, the prefix of $t[i + 1 : n]$ occurring in $TR_i(t)$ needs to matched and the remainder inserted at the location where $t[i+1 : n]$ has no continuation. The location where this prefix ends is called the *head* for iteration $i + 1$ and denoted $head_{i+1}$. The remainder of $t[i + 1 : n]$ after $head_{i+1}$ is called $tail_{i+1}$ so that $t[i + 1 : n] = head_{i+1}tail_{i+1}$. When $head_{i+1}$ is found the algorithm simply uses $add(\overline{head_{i+1}}, tail_{i+1})$ so that a new leaf edge is added to the tree and $t[i + 1 : n]$ is represented in the tree. If $\overline{head_{i+1}}$ is not already a vertex in the tree, it first has to be created. The algorithm also adds the suffix link (as defined in Section 5.1.4) for the vertex corresponding to the previous head, $head_i$. Suffix links are again defined for all inner vertices successively, since all head-vertices are inner. Moreover, the only

inner vertex in phase i which might not have a defined suffix link is the latest one added, the one corresponding to $head_{i+1}$, but it is defined in the next phase.

5.2.3 Finding the New Head Efficiently

All additions, including suffix links, to the tree in a given phase i depends on the location of $head_{i+1}$, so the key of the algorithm is to find it efficiently. Again, suffix links and the *skip-count* procedure and a couple of additional observations are used for this task. The following observations are based on the lemmas in Section 5.1 but tailored for *Mcc*. Suppose the algorithm currently executes in phase i.

O1. Because $head_i$ is a prefix of $t[i:n]$, $link(\overline{head_i})$ is an ancestor of $\overline{head_{i+1}}$. To see this let $a = t[i]$ and $x = link(\overline{head_i})$ such that $head(i) = ax$ and note that by the definition of *head*, there exist a longer suffix $t[j:n]$ such that both $ax \sqsubset t[i:n]$ and $ax \sqsubset t[j:n]$. But then both $x \sqsubset t[i+1:n]$ and $x \sqsubset t[j+1:n]$, so $link(\overline{head_i})$ must be a prefix of $\overline{head_{i+1}}$. Note that $\overline{head_{i+1}}$ may not yet exist in the current tree, if it does not, this observation "pretends" that it does as $\overline{head_{i+1}}$ will be added at the end of the phase.

O2. Let $v = parent(u)$ for some inner vertex u. Then either v is the root or $link(v)$ is an ancestor of $link(u)$ (Lemma 5.1.3).

O3. In TR_{i+1} for nonempty $head_i$, the suffix link for $link(parent(\overline{head_i}))$ has already been computed, since $link(\overline{head_i})$ is the only inner vertex in the current tree which may not have a suffix link.

Based on these observations, the scan to $head_{i+1}$ can be started at $v' = link(parent(\overline{head_i}))$, and not the root. It may of course be the case that $\overline{head_i}$ is the root, in which case the scan is started from the root.

Assume $u = \overline{head_i}$ is not the root and let $v = parent(u)$, $v' = link(v)$ and x be the label on the edge $v \xrightarrow{x} u$. By observation 1, the string $w = lp(v')x$ must be a prefix of $head_{i+1}$. This means that, even though $\overline{head_{i+1}}$ might not exist in $TR_i(t)$ yet, the string x can be matched by some downward path from v'

using *skip-count*(v', x) and w can be found in time proportional to the number of vertices along the path x. Once w is found, two cases can arise:

(i) $\overline{w}$ is a vertex in $TR_i(t)$. Then to find $head_{i+1}$, the algorithm has to match the remainder of $t[i+1:n]$ (i.e $tail_i$) after w as far as possible, which has to be done using *scan*. The location where no continuation exists for $t[i+1:n]$ is exactly $head_{i+1}$. Assume the last edge matched using scan is $e = \overline{w} \xrightarrow{y} \overline{wy}$ and z was the prefix matched on that edge. If q is the remainder of $tail_i$ after z, then call *split*$(e, |z|)$ and then *add*$(\overline{wz}, q)$ to add the new leaf vertex. Naturally it can happen that $head_{i+1} = w$ and in which case no further scan is needed under $\overline{w}$, *add*$(\overline{w}, tail_{i+1})$ is simply used.

(ii) $\overline{w}$ is not a vertex in $TR_i(t)$. Then, suppose that $head_{i+1} \neq w$, i.e the continuation after w spells out a non-empty prefix of $tail_i$. Since $\overline{head_i}$ is an inner vertex, it has at least two different continuations, $tail_i$ and one other. But since $\overline{w} = link(\overline{head_i})$ it follows that $\overline{w}$ must also have two continuations, a contradiction. Thus, $head_{i+1} = w$ and a new vertex needs to be inserted at the location where w ends. If $w = xz$ ends on the edge $e = \overline{x} \xrightarrow{zy} \overline{wy}$, then *split*$(e, |z|)$ is used to create the inner vertex $\overline{w}$ after which *add*$(\overline{w}, tail_{i+1})$ is used.

5.2.4 Summary of the Algorithm

An exact description of *Mcc*, based on Section 5.2.2 and 5.2.3, is given below.

1. Initialize the algorithm by create $TR_1(t)$, the trie representing the entire suffix $t[1:n]$ and nothing else. Let $head_1 = \epsilon$
2. Then, for $1 \leq i \leq n$ do the following:
   (a) Let e be the edge $v \xrightarrow{x} \overline{head_i}$. Jump to $v' = link(v)$ and then match the string $w = label(v')x$ using *skip-count*.
   (b) Insert a new leaf edge, and possibly a new inner vertex as described in Section 5.2.3.
   (c) Define $link(\overline{head_i}) = \overline{w}$.

5.2.5 The Final Result

Theorem 5.2.1. *McCreight's Suffix tree computes $ST(t)$ and defines all suffix links correctly.*

Proof. By definition of tree extension, there exists no inner vertex in $TR_n(t) = st(t)$ with less than two children. Because $head_i$ is always the longest match in the current tree, the downward edges from each inner vertex have no common prefix. The suffix links for $TR_1(t)$ are correct because it contains no inner vertices, and for each successive iteration, the algorithm correctly adds the suffix link for the previous vertex head (except if it is the root in which case nothing has to be done.) Because $head_i tail_i$ exists in the current tree at the end of every iteration $i-1$, there exists a leaf vertex i for every suffix with number i. □

Theorem 5.2.2. *McCreight's Suffix tree algorithm implemented using suffix links and skip/count computes $ST(t)$ in $O(n)$ time.*

Except for possibly *skip-count* or *scan*, all operations in a given phase are constant, so disregarding these operations, the total time required by the algorithm is linear. Now consider the separate contribution *skip-count* to the total runtime, which is executed at most $O(n)$ times. By Theorem 5.1.4, the depth decreases at most 1 when traversing a suffix link, and the depth decreases another one when moving to a parent node. Moreover, no path in $ST(t)$ is longer than n, so the total contribution of *skip-count* is bounded by $O(n)$.

Now consider the separate contribution by *scan*. Observe that since $link(\overline{head_i})$ is a proper prefix of $head_{i+1}$ it follows that $|head_{i+1}| - |head_i| + 1 \geq 0$. Summing over n cancel out every *head* term except for the first and the last, so $\sum_{i=1}^{n} |head_{i+1}| - |head_i| + 1 = |head_{n+1}| - |head_1| + n$. This clearly linear, so the total contribution is $O(n)$ here as well. □

5.3 The Intersection of Ukkonen's and McCreight's Algorithm.

It should now be clear that *Ukk* and *Mcc* are quite similar and share many ideas. *Ukk* reads t from left to right, character by character and incrementally constructs suffix trees for the prefixes of t seen so far. During the algorithm, the labels of the leaf edges grow implicitly as t is read, while some edges may be split to introduce new leaf edges. McCreight's algorithm reads t from left to right, suffix by suffix and incrementally inserts suffixes into an initially empty tree. As it inserts a new suffix, it splits one specific edge and introduces a new leaf. Disregarding whether or not the algorithm introduces "open" edges as in Ukkonen's case, both algorithms only rely on the same abstract operations, namely *split* and *add*. The difference is the order and location of the operations applied to the intermediate trees. Giegerich and Kurtz[16] showed how *Ukk* could be transformed into McCreight's by a modification of its control structure, principally by removing the steps where *Ukk* do not add any new edge to the current tree. This transformation removes the online property of the algorithm but makes it slightly faster.

6 Recursive Algorithms

Even though suffix trees algorithms traditionally have been implemented and analyzed in the imperative paradigm[27], it is not the only way. In this section, the paradigm is shifted to a more declarative style, introducing two new algorithms, inspired by elements of functional programming.[4]. Studying computation from a functional view can lead to to completely new insights, a good example being Giegerich and Kurtz[15]. Their work resulted in many interesting observations on *Ukk* and *Mcc*, also discovering that the recursive top-down approach for building suffix trees can be very fast and usable in practice.

6.1 Giegerich and Kurtz's Suffix Tree Algorithm

Giegerich and Kurtz was not the first to study declarative top-down suffix tree construction[3][18], but due to its disappointing $O(n^2)$ worst case bound it was never seriously considered useful in practice[17]. Giegerich and Kurtz added value to this approach of suffix tree construction by introducing ideas from programming, giving the algorithm a much more dynamic characteristic. Functional programming is based on fundamentally different ideas on computation compared to conventional, procedural languages like C or FORTRAN, eliminating for example loops and side effects in most computations[36]. The central idea from functional programming on which Giegerich and Kurtz based the new top-down algorithm was "laziness", an evaluation strategy common among functional languages that only evaluates expressions when explicitly forced to do so. Their algorithm, henceforth referred to as *GK*, recursively generates a suffix tree from the root and downwards, successively evaluating vertices only for the part of the tree needed by, for example, matching a pattern.

6.1.1 Lazy Evaluation

The concept of lazy evaluation was introduced for lambda calculus by Wadsworth in 1971[38] and got its breakthrough in programming when lazy, statically typed, pure languages like MIRANDA[35] and HASKELL[22] gained popularity during the

[4]For an extensive discussion on the functional programming paradigm, see[21].

1980's and 1990's[36]. Lazy evaluation is useful in the context of suffix tree construction since it implies automatic pruning of unused parts of the tree, reducing the total computation effort. In a lazy execution environment, no work needs to be done before any pattern is to be matched — it is the pattern matching itself that drives the evaluation of the parts of the tree needed. All vertices and edges that do not lie on the path that represents the pattern are left unevaluated until maybe a new pattern is presented for which they need to be evaluated. This property is obtained "for free" in lazy-by-nature languages, but it is also fairly easy to simulate lazy evaluation in a strict language like C, by an explicit representation of the unevaluated data.

GK is the only suffix tree algorithm discussed in this thesis that can take advantage of lazy evaluation[17] as all other algorithms rely on order of execution. Lazy evaluation obscures execution order, focusing on the data itself to drive the execution. Thus, *GK* lets the structure of the suffix tree drive the evaluation rules, not the other way around.

6.1.2 Description of the Algorithm

The algorithm is very simple and can be fully described in a half-dozen lines of text: A suffix tree for the text t of length n is to be constructed. First group the non-nested[5] suffixes of t according to their first characters, so that $S_a = \{s \mid as \in S'(t)\}$ for all $a \in A$ and $S'(t)$ being the non-nested suffixes of t. Then recursively construct a subtree for each S_a, attached to the root by the edge labeled with the longest common prefix of the labels in S_a. The method naturally proceeds in a top-down manner, evaluating each branch until $|S_a| = 1$, at which point a single leaf edge is inserted marked by the only element of S_a.

6.1.3 Performance in Theory and Practice

A tight time bound for *GK* is given by considering the number of characters the algorithm needs to read. All operations not related to reading suffixes, such as extending the current node with a new edge, is constant. The sum of suffix lengths is $n(n+1)/2$. All suffixes could be read to the last character, implying a

[5] A suffix of t is called *non-nested*, if it is not a prefix of another suffix in t.

worst case of $O(n^2)$. However, since the expected length of the longest repeated substring is $O(\log n)$[7], and because no suffix is read further when it becomes unique, the expected case is $O(n \log n)$.

Why bother with *GK*, when linear time algorithms exist? Other than being exceptionally simple to understand and implement, the answer lies in the laziness and the "natural" way of computation. *GK* can be implemented lazily because it only ever needs to consider an edge once during construction, and then never again. Consequently, it can be implemented without random access into memory, cache hits increase and page faults decrease[1]. This can lead to great performance in practice[15], as confirmed the experiments in Section 8.

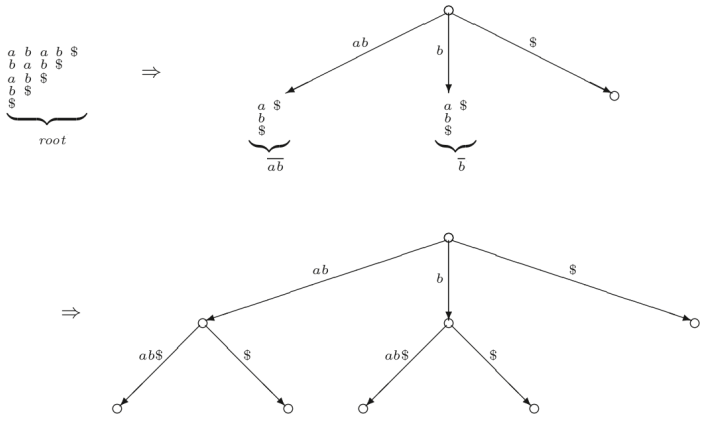

Figure 2: The evaluation of a suffix tree for *abab*\$ using *GK*. Each vertex v corresponds to a set of vertices, grouped according to their first characters. The longest common prefix for each group is computed, these become the labels of the new edges under v.

6.2 Farach's Suffix Tree Algorithm

The asymptotic time analysis of the imperative algorithms *Ukk* and *Mcc* can be misleading: it assumes that the alphabetical factor is a small constant and thus disregards it completely in the analysis. However, empirical evidence indicates that the alphabet size often plays a significant role in practice [15]. The *scan* and *skip-count* procedures has to for every vertex they encounter, find the correct continuation edge under that vertex. This implies the need for a lookup data structure, for example: a static array, a linked list, a balanced tree or a hash table. Whatever structure used in the implementation implies a certain space and time penalty, which when applied frequently can be drastic, even if the alphabet size is in the order of hundreds compared to a text size of thousands or even millions[19].

Farach's algorithm[12] is henceforth referred to as *Fch*. It is recursive just like *GK*, but instead of constructing the entire tree top-down it is based on recursive merging of suffix trees for smaller sets of suffixes. Instead of iterating over the characters in the alphabet, it simply compares a linear number of characters during merging, and thus circumventing all problems related to alphabet size. The only assumption is that the alphabet is of integer type, i.e $\Sigma = \{1, 2, \ldots k\}$ such that $k \in O(|t|)$ for the text t. As pointed out by Smyth, this is usually the case in practice[34]. The following description of *Fch* is based on [34] and[12].

6.2.1 High Level Description

Fch constructs $ST(t)$ for the string t with $t[|t|] = \$$ of length n in linear time. The first step of is to compute the *odd tree* $ST_o(t)$, the suffix tree of all suffixes beginning at odd positions in t. $ST_o(T)$ provides enough information to derive the *even tree* $ST_e(t)$, the suffix tree of all suffixes beginning at even positions in t. The odd and even trees together clearly represent all suffixes of t, so in order to obtain $ST(t)$, $ST_o(t)$ and $ST_e(t)$ are merged.

6.2.2 Constructing the Odd Tree

The odd suffixes of t are $t[2i-1:n]$ for $1 \leq i \leq \lceil n/2 \rceil$, the suffixes starting at odd indices in t. To construct $ST_o(t)$, consider the suffix tree for a completely new string t', half the size of t, obtained as follows: Map out the list of pairs $l(t)$ given by $(t[2i-1], t[2i])$ for $1 \leq i \leq \lceil n/2 \rceil$, the pairs of every even/odd character pair in t. If t is even, then \$ is excluded, otherwise the last pair in the list becomes $(t[n], \$)$. For example, if $t = 1211122122 21\$$ for $\Sigma = \{1, 2\}$, then $l(t) = [(1,2), (1,1), (1,2), (2,1), (2,2), (2,1)]$. To map a pair into a single character, radix sort $l(t)$, remove duplicates and compute the rank[6] of each pair in $l(t)$. The string t' emerges from this mapping, so $t' = 212343\$$ in the example. Assume now that the suffix tree $ST(t')$ is given and consider the following observations.

O1. Due to the nature of the procedure above, each odd suffix $t[2i-1:n]$ corresponds to the suffix $t'[i:n]$ in the new string, thus each leaf with *number*(i) in $ST(t')$ corresponds to the leaf with *number*$(2i-1)$ in $ST_o(t)$.

O2. An inner vertex in $ST(t')$ of length k corresponds to an inner vertex in $ST_o(t)$ of length $2k$.

A new labeled tree $T'(t)$ can be formed by replacing each character in $ST(t')$ by the two characters it was generated from in t. Thus, $T'(t)$ is the tree with the same structure as $ST(t')$, but with different labels. $T'(t)$ and $ST_o(t)$ are similar to the degree that the latter can be derived from the former in linear time. $T'(t)$ contains all odd suffixes of t, but is not necessarily a compact trie for those. First off, each character in $ST(t')$ was replaced by two characters, so all edge labels of $T'(t)$ have even length labels but $ST_o(t)$ might very well have odd label lengths as well. Moreover, the pairs in $l(t)$ can have different ranks but still have the same first component, which after the transformation would result in two a-edges originating at the same vertex for some $a \in \Sigma$. To transform $T'(t)$ into $ST_o(t)$, both inconsistencies are repaired by the following procedure: For every inner vertex u, and for all children that start with the same character a, a new vertex u' is inserted below u, such that *label*$((u, u')) = a$. If all children

[6] The index of the pair in the sorted, unique list. $(1,1) \implies 1$, $(1,2) \implies 2$, $(2,1) \implies 3$ and $(2,2) \implies 4$ in the example.

of u starts with the same character, then u' would be the only resulting child of u, in which case (u, u') can be contracted with the edge $(parent(u), u)$. $T'(t)$ has now been transformed into $T_o(t)$; this transformation clearly takes constant time for each inner vertex, so linear time overall.

Lemma 7.2.1. *If $\tau(n)$ is the time it takes to build a suffix tree of size n, and $ST(t')$ is already constructed, then the odd tree can be constructed in $\tau(n/2) + O(n)$.*

The tree $ST(t')$ is recursively constructed with the same algorithm *Fch* for shorter and shorter strings, the base case being when t' is so short that its suffix tree can be trivially built. The algorithm then constructs the odd tree for the string of double size, which then is turned into a suffix tree. This procedure continues bottom up until finally $ST(t)$ is obtained. What remains tho show is thus how to derive $ST_e(t)$ from $ST_o(t)$, and then how to merge the two trees.

6.2.3 Constructing the Even Tree

The even suffixes of t are $t[2i : n]$ for $1 \leq i \leq \lceil n/2 \rceil$, the suffixes beginning at even indices in t. $ST_e(t)$ is the suffix tree representing only those suffixes. The second step carried out by *Fch* is deriving $ST_e(t)$ from $ST_o(t)$, and can be done in linear time.

Suppose the algorithm was given (1) an in-order traversal of its the leaves in $ST_e(t)$ and (2) the lowest common ancestor for each pair of two distinct leaves in $ST_e(t)$. Then $ST_e(t)$ would be straightforward to construct: each leaf represents a suffix and the lowest common ancestor of adjacent leaf pairs provides enough information to know what part of that suffix should be represented by the edge labels on its path. Similarly, suppose the algorithm was given (1) the *lexicographic*[7] ordering of the even suffixes and (2) the longest common prefix of each pair of two distinct suffixes. Then again, the suffix tree could be constructed in a straightforward way. Firstly, it is reasonable to assume that the edges are

[7]Ordering two strings lexicographically means ordering the strings in standard dictionary order: If the first characters of the two strings are equal, then the second character is compared and so on until two characters are different or one string has no more characters to compare. In that case, the shorter string is ordered lower than the longer one.

inserted with labels ordered lexicographically when constructing a suffix tree. This implies that the lexicographic ordering of the even suffixes is in fact the in-order traversal of the leaves of $ST_e(t)$. Secondly, since $ST_e(t)$ is a suffix tree, the depth of the lowest common ancestor can be derived from the longest common prefix. Let $lcp(i, j)$ be the length of the longest common prefix of the two suffixes starting at index i and j and let $lca(i, j)$ be the lowest common ancestor of two vertices representing those suffixes in $ST_e(t)$. By the basic structure of suffix trees, $lcp(i, j) = |lp(lca(i, j))|$, where lp is the *label path*.

This section shows how to from $ST_o(t)$ compute a lexicographic ordering of the even suffixes of t and the longest common prefix of any two even suffixes, the two pieces of information needed to construct $ST_e(t)$ in linear time.

First consider the lexicographic ordering of the even suffixes. An even suffix is a single character followed by an odd suffix. The lexicographic ordering for the *odd suffixes* is easily obtained by a linear time traversal of the odd tree. Thus, by radix sorting the characters at even positions together with the positions the odd suffixes correspond to, a lexicographic ordering for the even suffixes can be obtained. Let $l(t)$ be the list of elements $(t[2i], 2i + 1)$ in range $\leq i \leq \lceil n/2 \rceil$, the pairs whose first elements corresponding to the even characters and second being the index of the successive odd suffix. The ordering of $l(t)$ depends on the in order traversal obtained from $ST_o(t)$, so for the example Section 6.2.2, $l(t)$ would become $[(2, 3), (1, 5), (2, 7), (2, 11), (1, 9), (1, 13)]$. The pair for which the odd index is 1 is not included in $l(t)$, since it is not preceded by an even character. Now, a radix sort on the first component of the elements of $l(t)$ is performed, resulting in $l_{sort}(t) = [(1, 5), (1, 9), (1, 13), (2, 3), (1, 7), (2, 11)]$. Note that radix sort is stable, meaning the initial order of the elements with the same key is preserved. The sorted list immediately gives the lexicographic ordering of the even suffixes, just subtract one for every second element in $l_{sort}(t)$ to obtain $[4, 8, 12, 2, 6, 10]$.

Now consider the longest common prefix for two even leaves. The observation is that for the two leaves with numbers $2i$ and $2j$:

$$lcp(2i, 2j) = \begin{Bmatrix} lcp(2i+1, 2j+1) + 1 & \text{, if } t[2i] = t[2j] \\ 0 & \text{, otherwise} \end{Bmatrix}$$

Thus, given $lcp(2i+1, 2j+1)$, $lcp(2i, 2j)$ can be computed in constant time. However $lcp(2i+1, 2j+1)$ is already available, since it is the longest common prefix of two odd leaves, generated during the construction of $ST_o(t)$, already available. Furthermore, to derive the lowest common ancestors *Fch* uses a celebrated algorithm[20] doing exactly that in linear time. This implies that the following lemma.

Lemma 7.2.2. *Given a string t and its odd suffix tree $ST_o(t)$, $ST_e(t)$ can be constructed in $O(n)$ time.*

6.2.4 The Overmerge Procedure

$ST_o(t)$ and $ST_e(t)$ together represents all suffixes of t, and thus by merging them into one, $ST(t)$ emerges. First consider how one would merge $ST_o(t)$ and $ST_e(t)$ if every label in the trees had exactly one character. In that case, merging the trees could simply be done using a depth-first search tree on each tree, recursively merging the subtrees for equal labels and simply attaching the entire subtree corresponding to the lesser label for unequal labels. $ST_o(t)$ and $ST_e(t)$ do in general not have single character labels, and the algorithm cannot afford to completely compare the labels as in the above procedure. Furthermore, two labels may share a common prefix so that a new vertex needs to be inserted, splitting the merge after this prefix. If the edge labels are of different lengths, it might be that the shorter is a prefix of the longer. In that case, the leaf of the shorter would become an inner vertex in the merged tree.

Instead of employing a rather complicated recursive merging procedure, *Fch* uses a simple one that might result in an "overmerged" suffix tree that will be fixed at a later stage. For determining the equality of two labels, a "weak" oracle

is employed that in constant time declares if two edges are equal or not. This oracle always correctly compare two equal labels, but also might also declare two unequal labels to be the same. This weakness sometimes results in an overestimation in the vertex depth for merged vertices. On the other hand, the depth of a vertex cannot be underestimated, since its entire label in both $ST_o(t)$ and $ST_e(t)$ are merged directly.

Any weak oracle can be used for the overmerge procedure, so the simplest one, just comparing the first character of each of the two labels, is chosen. The new depth-first search merge thus starts at the root and compares the first characters of the two edge labels of the first root child in the two trees. Assume these edges have labels x and y and that every edge stores its length. Three cases can apply:

(i) If $x[1] \neq y[1]$, then the subtree corresponding to the lesser of the two characters is unique (due to the lexicographic ordering of edge labels) and attached as before to the root in the merged tree.

(ii) If $x[1] = y[1]$ and $|x| = |y|$, then the two edges are declared equal and thus merged into one single edge in the merged tree.

(iii) If $x[1] = y[1]$ and $|x| \neq |y|$, then the prefix of the longer edge with the same length as the shorter is merged with the shorter to a single edge. The rest of the longer edge is attached below the merged edge.

The merge procedure is constructed so that only edges of equal length are merged, and thus the only possible inconsistency after the merge is in the characters of the labels in the merged tree. When an edge is merged, the algorithm continues with recursively merging its subtree. In case (i) where the entire subtree is attached without merging, the algorithm continues with merging its right sibling. All three cases clearly take constant time to perform, so the overmerge takes linear time overall. The resulting tree is denoted $T_{om}(t)$.

6.2.5 Constructing the LCP Tree

As discussed in the previous section, it may be that some vertices were merged too far down the tree as a result of an overestimation due to the weak oracle. The final step of *Fch* is thus to "unmerge" these vertices by reducing this

depth. To know how, the algorithm employs an auxiliary data structure that determines the longest common prefix of two descendant leaves of a some vertex u. Accordingly, let u be a vertex in $T_{om}(t)$ and let $2i$ and $2j-1$ be the leaf numbers of be two descendants of u such that $lca(2i, 2j-1) = u$ and define $d(u) = lcp(2i, 2j-1)$. There could be multiple choices of $2i$ and $2j-1$, but it does not matter which of these are chosen, since they all share the same longest common prefix, in other words $d(u)$ has the same value no matter which even/odd leaf pair is picked.

The even/odd pair of leaves $(2i, 2j-1)$ must always reside in a merged subtree of $T_{om}(t)$ as only the parts that were directly attached without a merge exclusively contain even or odd leaves. Thus u is a vertex in a merged part of the tree and it may be that u is merged too far down in which case $|lp(u)| > d(u)$. On the other hand, $d(u)$ cannot be smaller than $|lp(u)|$ as a vertex cannot be merged to far up by the overmerge procedure. To turn $T_{om}(t)$ into a real suffix tree, *Fch* has to make sure that $|lp(u)| = d(u)$ holds (Section 6.2.3) for every such vertex u.

Consider the pair of suffixes shifted one index to the right, namely $(2i+1, 2j)$ and let $v = lca(2i+1, 2j)$ be the lowest common ancestor for those suffixes. The function *lcp-tree*$(u) = v$ defines a pointer from u to v for all vertices u in $T_{om}(t)$. The *lcp-tree* relation is interesting, because (as shown in the proof below) $d(u) = d(v) + 1$ as long as u is not the root of $T_{om}(t)$.

Lemma 6.2.1. *The lcp-tree function defines a tree on the vertices of $T_{om}(t)$. Furthermore, if u is a inner vertex in in $T_{om}(t)$ then $d(u)$ equals the depth of the corresponding vertex in this tree.*

Proof. First, it is shown that $d(u) = d(v) + 1$. Since *lcp-tree*$(u) = v$ there is a odd/even pair $(2i, 2j-1)$ with lowest common ancestor u such that the lowest common ancestor of $(2i+1, 2j)$ is v. Furthermore, u is not the root and because $d(r) = 0$ only for the root, $lcp(2i, 2j-1) = 1 + lcp(2i+1, 2j)$ holds by a similar observation as in Section 6.2.3. Finally, by the definition of *lcp* and the observation that all odd/even pairs give the same d, $d(v) = lcp(2i+1, 2j)$, so $d(u) = d(v) + 1$.

Note that $d(u) = d(v) + 1$ implies that the *lcp-tree* function defines a tree on the vertices of $T_{om}(t)$, since all pointers point "upwards" towards a vertex representing a shorter substring. Also, since $d(r) = 0$ only for the root r, $d(u)$ is the depth of u in this tree, and thus the lemma is proved. □

As a byproduct of the overmerge procedure, the algorithm pairs neighboring even and odd leaves of the result. Given two such leaves $2i$ and $2j - 1$, u can be computed in constant time after linear preprocessing[20]. The d tree can be traversed in linear time to compute $d(u)$ for every vertex u. Moreover, $|lp(u)|$ can be stored by either $ST_o(t)$ or $ST_e(t)$. This constitutes the information needed by the overmerge procedure, and based on the discussion above it can be obtained in linear time.

6.2.6 The Unmerge Procedure

Let u be a vertex such that $|lp(v)| = d(v)$ for all ancestors v of u, but $|lp(u)| > d(u)$. The vertex u is thus overmerged, $lp(u)$ and should actually represent two subtrees, one originating from the $ST_o(t)$ and the other one from $ST_e(t)$. With a linear time traversal of $T_{om}(t)$, the algorithm finds all such vertices u in linear time.

To unmerge a particular vertex u, the following procedure is followed. Since u is a merged vertex, it originates from both an even and an odd subtree, each of which needs to be restored individually to obtain the structure as before they were merged. To do this, set the parent of each vertex in the subtrees under u to the parent in its original tree ($ST_o(t)$ or $ST_e(t)$). Let $ST_o^u(t)$ and $ST_e^u(t)$ denote these two subtrees. A new vertex u' is inserted with an edge from *parent*(u), such that $|lp(u')| = d(u)$. Finally, attach $ST_o^u(t)$ and $ST_e^u(t)$ to u' in lexicographic order. The edge label of $(parent(u), u')$ as well as the edge labels of the first edge of $ST_o^u(t)$ and $ST_e^u(t)$ changes according to Figure 3. When this procedure is complete for all vertices that needs to be unmerged, the tree is correctly transformed into the suffix tree for $ST(t)$. The total time for the unmerge is linear like all other steps of the algorithm. The suffix tree for t can thus be constructed in linear time and space.

6.2.7 The Final Result

Theorem 7.2.3. *Given a string t of length n, the suffix tree $ST(t)$ can be constructed in $O(n)$ time and space.*

Proof. By Lemma 7.2.1 and 7.2.2 and the discussion of Section 6.2.4, 6.2.5 and 6.2.6 □.

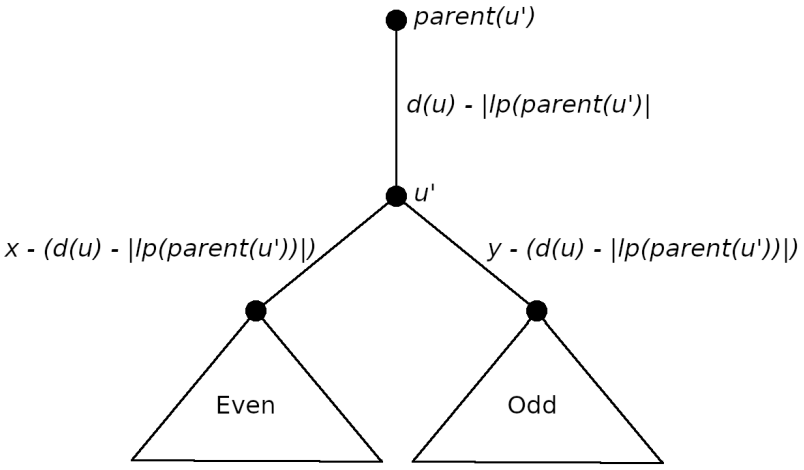

Figure 3: *Fch* A vertex u is unmerged removing it and inserting a new vertex u' and updates the edges as in this figure. x (respectively, y) is the length of the edge from u to the even (respectively, odd) subtree that was originally merged under u.

7 Implementation Techniques

7.1 Choice of Algorithms

In this section, two implementations of suffix tree construction are described. The implementation of *Mcc* uses linked lists for tree traversal and the implementation of *Gk* uses counting sort[8] for grouping vertices. *Mcc* was chosen for this section since it is one of the most conventional methods for linear time suffix tree construction, and because it is slightly faster than *Ukk*. The online property of *Ukk* is interesting, but *Gk* also obtains this property with no additional effort, as long as it is implemented lazily. Prior experiments have indicated that *GK* has comparatively good performance in practice[15][17]. This, together with its lazy property made it interesting to study in practice. Additionally, implementing *Gk* with counting sort makes it is independent on size of the alphabet, which in fact is the main advantage of *Fch*.

7.2 *GK* in Haskell

Before describing the main implementations, this section presents a prototype of *GK* in HASKELL, the implementation language of the original authors[15]. HASKELL provides many high-level abstraction capabilities, allowing the programmer to focus on the problem at hand and quickly expressing ideas in code. The language is lazy by default, so no explicit synchronization between evaluated and unevaluated data is needed. This allows for the very concise and simple implementation of the *GK* algorithm listed in Appendix A[9]. The program was compiled with GHC[10] version 8.2.2 with optimization level O3 and executed to build a suffix tree for a 4 Mb string. For the details details on the computer hosting the experiments, see Section 8. As seen in Figure 4, the resulting process required over 200 Megabytes of working memory, about 55 times the size of the input! Being a highly abstracted, garbage collected programming language, HASKELL offers limited support in controlling exactly what gets allocated and

[8] https://en.wikipedia.org/wiki/Counting_sort

[9] The program is also listed at:

`https://github.com/lsund/lazy-suffix-trees-haskell`

[10] The Glasgow Haskell Compiler `https://www.haskell.org/ghc/`

not[11]. Consequently, it seems reasonable to resort to a lower level language with better capabilities in reducing space consumption.

To increase the efficiency of suffix tree implementations, a natural choice of programming language would be one designed to give the most control possible over the data representation and program execution. The C programming language[24] was designed to minimize overhead, partly by reducing safety checks at runtime such as array index out of bounds exceptions. Furthermore, it provides many techniques for directly manipulating small bits of data and direct access to their underlying representation in memory. Its minimal design and raw efficiency give it a timeless ambiance as it is one of the oldest programming languages still used extensively today[32]. The C programming language is a small language; it provides no "fancy" libraries and data structures. It turns out, however, that anything other than the most basic language primitives is unnecessary and even suboptimal when implementing suffix trees — any redundant bytes or unnecessary computations can be critical.

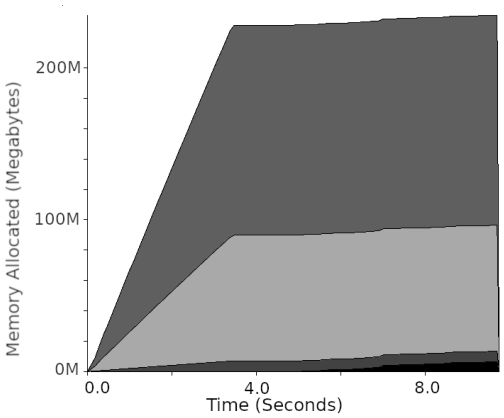

Figure 4: Space required when executing the haskell implementation of *GK* with a 4 megabyte input text file, evaluating full suffix tree. The different layers of grey represent different sub-procedures of the program.

[11] In fact, one of the main "problems" of Haskell has always been unpredictability in memory allocation, primarily due to its laziness[2].

7.3 *Mcc* in C

The program described in this section is based on[26] and referred to as `MCC`. The description in Section 5.2 translates well into the C programming language, as they are both very imperative by nature. The main challenge is representing the suffix tree itself as efficiently as possible and keeping an eye on the details during the execution of the algorithm, even for constant time operations. This section describes the most important parts of the programs in direct correspondance with real, working C code. The intent is to highlight essential details in the implementation which could be obscured when describing the program through pseudo-code. For practical reasons, the programs are not fully described in this section. For complementary code, see Appendix B and for the full program, see `https://github.com/lsund/linear-suffix-trees`.

7.3.1 Basic Data definitions

Consider the minimal set of elements `MCC` needs to represent. The vertices of the tree constitute the basic building blocks; unique integers can encode these. Edges and suffix links are essentially relations between two vertices and implemented by a pointer from one vertex to another. Edge labels can be represented as plain integers as well, more on that in Section 7.3.3. Thus, the primitive data types necessary to represent a suffix tree in C are just the following:

```
1  typedef unsigned long Uint; // alias `unsigned long` to `Uint`
2  typedef Uint Vertex;
3  typedef Uint *VertexP; // VertexP stands for `VertexPointer`
```

To represent a set of vertices, a dynamic array structure called a `Table` is used, mapping memory addresses to vertices. Its fields provide access to the first occupied address, the next unoccupied address, the size (number of vertices in the table) and a next index integer that increments as vertices are added to the table.

```
typedef struct table {
   VertexP fst; // Base address
   VertexP nxt; // Next unoccupied address
   Uint nxt_ind; // Next unoccupied index
   Uint size;  // Number of vertices stored in the table
} Table;
```

The initial structure for the suffix tree `STree` is defined, containing one table for leaf vertices and one for inner vertices. It is necessary to separate the two types of vertices in McCreight's algorithm, as they have different characteristics.

```
typedef struct stree {
   Table ls; // The leaves
   Table is; // The inner vertices

   // More fields
} STree;
```

The series of tries generated by *Mcc* is referenced to by a single variable named `st`, which continuously gets updated during execution.

The text is represented by storing a pointer to the start of the memory block allocated for the text, as well as its length. `MCC` also keeps a pointer to the last address in this memory space representing the sentinel character. Each character is in this case represented by a C wide character[12]. Note that if the text only contains ASCII[13] characters, then a character can instead be represented as a plain `char`, possibly reducing the memory required by the text.

[12]The program uses wide characters as specified by the `wchar.h` library in the C90 standard. A C wide character has varying size depending on the compiler specifics but supports distinct codes for all supported locales. Thus by setting locale to for example `en_US.utf8`, the `wchar_t` type provides the possibility to represent all UTF-8 characters, in this case a 64-bit integer.

[13]ASCII stands for 'American Standard Code for Information Interchange'.

```
#include <wchar.h>

typedef Wchar wchar_t;

typedef struct text {
   Wchar *fst;    // first character of the text
   Wchar *lst;    // last character of the text
   Uint len;      // its length
} Text;
```

7.3.2 Retrieving a Vertex From the Table

The index of a vertex in the table it resides in and its address has a one-to-one relationship, implying implicit retrieval of indices by subtracting the table base address from vertex addresses.

```
// The symbol `R` refers to an address, also called a reference.
//
// Inner vertices
#define REF_TO_INDEX(R)     ((Uint) ((R) - st->is.fst))
// Leaves
#define LEAFREF_TO_INDEX(R) ((Uint) ((R) - st->ls.fst))
```

Conversely, one can retrieve the address of a vertex by adding its index to the base address in the corresponding table.

```
// The symbol `V` always refers to a vertex
//
// Inner vertices
#define VERTEX_TO_REF(V)     stree->is.fst + INDEX((V))
// Leaves
#define VERTEX_TO_LEAFREF(V) stree->ls.fst + INDEX((V))
```

The special macro `INDEX` is used for the following reason. The maximal value of an `Uint` is much larger than any reasonable amount of memory. To save space a vertex stores metadata about itself, whether the vertex is a leaf and whether it is small or large (discussed in Section 7.3.4). The metadata is stored by setting or clearing `MSB`[14] and `SECOND_MSB`, the two most significant

[14] `MSB` is a plain integer represented in binary as single '1' followed by sixty-three '0's, under the assumption that `Uint` is represented by 64 bits (and similar for `SECOND_MSB`).

bits of a vertex. To get the index of a vertex, these bits first has to be stripped off, as done by `INDEX`.

```
1   #define INDEX(V)        ((V) & ~(MSB | SECOND_MSB))
```

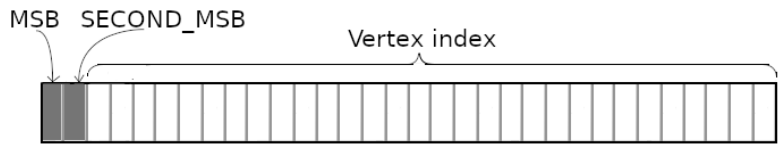

Figure 5: Each vertex is represented by an integer, in the figure represented by 32 bits. The two most significant bits are used to store metadata and the unevaluatedto store the vertex index.

7.3.3 Vertex Data

A leaf vertex needs no other information than a reference to its right sibling. An inner vertex v on the other hand needs to store:

1. A reference to its first (leftmost) child, allowing vertical tree traversal.
2. A reference to its first (right-hand-side) sibling, allowing horizontal tree traversal.
3. The length of $lp(v)$, also known as its *depth*.
4. Its *head index*, which is the smallest index i such that $head_i = v$. Every inner vertex correspond to $head_i$ for a specific iteration i during the algorithm and the head index for v is retrieved by the relation *head-index*(v). This index is useful for several reasons. First off, it can be used together with *depth*(v) to retrieve *label*(v), since it references the starting index of the correct label in the text. The head indices also imply a numbering for the inner vertices in the `is` table, if $v_1, v_2 \dots v_q$ are the inner vertices in the tree, then *head-index*$(v_i) <$ *head-index*(v_{i+1}) for any $i \in \{1, 2, \dots q-1\}$. The indexing of the vertices in the tables respects the index in this ordered sequence.
5. The suffix link given by *link*(v).

Reference (1) and (2) is results in a simple implementation of a linked list for accessing the children of a vertex. No additional data structure is needed for iterating over the alphabet, but the time to access a given child has a worst case of $O(|\Sigma|)$.

7.3.4 Reducing Redundant Data

Section 7.3.3 suggests that `MCC` needs 5 integers to represent each inner vertex. As shown in [26], this is usually not the case: the authors state and prove a set of observations that motivate a division of inner vertices into "small" and "large" vertices.

Observation 1. The head indices for two inner vertices cannot be equal. If $\overline{av}$ is a inner vertex for a single character a, then $\overline{v}$ is also a inner vertex in the suffix tree (Lemma 5.1.1), and thus the inequality *head-index*$(\overline{av}) + 1 \geq$ *head-index*$(\overline{v})$ holds. This suggests two cases: A vertex is small if and only if *head-index*$(\overline{av}) + 1 =$ *head-index*$(\overline{v})$ and large if and only if *head-index*$(\overline{av}) >$ *head-index*$(\overline{v})$. The root is neither small nor large.

The next observation indicates that a small vertex always is followed by another inner vertex, and that the last vertex always is a large vertex.

Observation 2. Let x be a string, a be a single character and $v_1, v_2 \dots v_q$ the sequence of vertices represented by the table `is`, ordered by increasing address numbers. If $v_{i-1} = \overline{ax}$ is small, then $v_i = \overline{x}$. Moreover, the last vertex in `is` is large, if it is not the root.

One can thus partition the set of inner vertices into *chains* of zero or more small vertices followed by a single large vertex. If $v_1, v_2, \dots v_q$ is the sequence inner vertices as above, then a chain $C = v_l, v_{l+1}, \dots v_r$ is a subsequence for $2 \leq l \leq r \leq q$ such that: v_{l-1} is not small, v_r is large and all other vertices in C are small. This allows for reducing redundant information:

Observation 3. Let $v_l, v_{l+1}, \ldots v_r$ be a chain. Then the following holds for any $i \in \{l, l+1, \ldots, r-1\}$.

1. $\mathit{depth}(v_i) = \mathit{depth}(v_r) + (r - i)$
2. $\mathit{head\text{-}index}(v_i) = \mathit{head\text{-}index}(v_r) - (r - i)$
3. $\mathit{link}(v_i) = v_{i+1}$.

Thus, if the distances to the last vertex in the chain is known, and the last vertex stores its depth, head index and suffix link, it is not necessary to store that information for any other vertex in the chain as it can be computed in constant time for each vertex. Consequently, a small vertex only stores its first child, first sibling and a new entity, the *distance* to the end of the chain, requiring 3 integers in total. A large vertex stores (as before) its first child, right sibling, depth, suffix link and head index, thus requiring 5 integers as before. A leaf vertex only needs to store its right sibling, so it needs one integer.

```
#define LEAF_VERTEXSIZE 1
#define SMALL_VERTEXSIZE 3
#define LARGE_VERTEXSIZE 5
```

`MCC` can find out whether a vertex is small or not by inspecting its most significant bit. If this bit is not set, the program considers the vertex large.

```
#define IS_SMALL(V)      ((V) & MSB)
#define IS_LARGE(V)      (!((V) & MSB))
#define WITH_SMALLBIT(V) (V) | MSB // Creates a small vertex.
```

A similar idea is used for distinguishing leaves from inner vertices.

```
#define IS_LEAF(V)       ((V) & LEAFBIT)
#define MAKE_LEAF(V)     ((V) | LEAFBIT) // Creates a leaf vertex
```

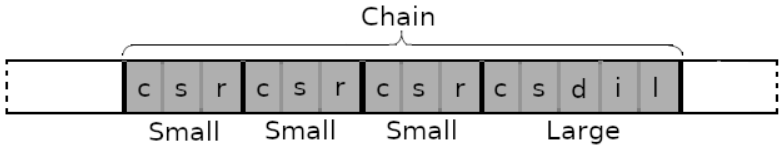

Figure 6: A chain is a sequence of small vertices followed by a large vertex. Each small vertex requires 3 integers to store and each large vertex requires 5 integers to store. The small vertex maintains its distance to the large vertex so that their depth, head index and suffix links can be retrieved implicitly. In the figure c stands for *child*, s for *sibling*, r for *distance*, d for *depth*, i for *head-index* and l for *suffix link*.

7.3.5 Retrieving the Vertex Information

Once the address of an small, large or leaf vertex has been found in its table, one can retrieve information related to the vertex in question. Because small, large and leaf vertices store a different amount of information, a varying number of subsequent addresses are reserved for each vertex. For example, if the first address of an inner vertex v is `0xFF00000`, the program occupies the addresses `0xFF00000`, `0xFF00001`, `0xFF00002` and `0xFF00003` and `0xFF00004` for this vertex. After storing v in `is`, `is.nxt` gets the value `0xFF00005`, `is.nxt_ind` gets incremented 5 times and `is.size` gets incremented once.

Given the address of a leaf, the address of its right sibling can be retrieved with `LEAF_SIBLING`, simply dereferencing the leaf reference.

```
1  #define LEAF_SIBLING(R) (*(R))
```

Given the address of an inner vertex, the address of its first child can be retrieved with `CHILD`. As inner vertices store information on whether they are large or small their most significant bit, this has to be stripped off before it is dereferenced.

```
1  #define CHILD(R)        ((*(R)) & ~(MSB))
```

Given the address of an inner vertex, the address of its right sibling can be retrieved with `SIBLING`, dereferencing the successor address.

```
#define SIBLING(R)      (*((R) + 1))
```

Given the address of an small inner vertex, the distance to the end of the chain can be retrieved with `DISTANCE`, dereferencing the second successor address.

```
#define DISTANCE(R)     (*((R) + 2))
```

Given the address of a large inner vertex, its depth, head index and suffix link can be retrieved with `DEPTH`, `HEADPOS` and `SUFFIX_LINK`.

```
// P is the address of a large vertex.
#define DEPTH(P)        (*((P) + 2))
#define HEADPOS(P)      (*((P) + 3))
#define SUFFIX_LINK(P)  (*((P) + 4))
```

Figure 7: The layout of the data of a large vertex in `sb.is`. Each rectangular box represents an integer in the figure.

7.3.6 The Main Loop

The previous sections described how to completely describe the structure of a suffix tree using only unique integer references. For the actual construction process though, the suffix tree needs to maintain a few other references.

```
typedef struct stree {
   // The table of inner vertices.
   Table is;
   // The table of leaf vertices.
   Table ls;
   // The list of root children.
   VertexP rs;
   // The current depth of the most newly inserted inner vertex.
   Uint c_depth;
   // The current chain of small vertices.
   Chain chain;
   // The location to be split after finding a new head location.
   SplitLoc split;
   // The lacation of the current head in the tree
   HeadLoc hd;
   // Points to the tail
   Wchar *tl;

} STree;
```

The new field `rs` provides access to the root edges of the suffix tree. These are the entry point when starting a match from the root. `SplitLoc` and `HeadLoc` represents locations for where an edge is to be split, and for where the current head is at. `Chain` is the current chain of vertices. The algorithm only ever needs to store one chain at a time — when a large vertex is inserted, the data for the vertices in the chain is stored and the chain reset. For the implementation of these new, complementary structures, see Appendix B.

Now the high-level procedure for McCreight's algorithm can be described. The main entrance point is the `construct` procedure.

```
1 void construct(STree *stree)
2 {
3    stree_init(stree);
4    while(!IS_SENTINEL(stree->tl)) {
5       find_next_head(stree);
6       insert_tailedge(stree);
7    }
8 }
```

`construct` takes a reference to an empty suffix tree structure. After initializing the suffix tree by setting default values and allocating initial memory, it performs one single loop, finding the heads and inserting new vertices as specified by McCreight's algorithm by populating the `is` and `ls` tables. `IS_SENTINEL(stree->tail)` controls weather `stree->tail` points to the last character of the string, only true if the last character was individually inserted by the previous phase. In that case, the algorithm should terminate, and when the last vertex has been inserted, the loop breaks and the program terminates. The `find_next_head` procedure does the main work in the program.

```
1  static void find_nxt_head(STree * st) {
2     if(head_is_root(st) && base_is_vertex(st)) {
3        st->tail++;
4        scan_tail(st);
5     } else {
6        if (is_head_old(st)) {
7           set_head_to_suffixlink(st);
8           scan_tail(st);
9        } else {
10          find_base(st);
11          if(base_is_vertex(st)) {
12             finalize_chain(st);
13             scan_tail(st);
14          } else {
15             append_chain(st);
16          }
17       }
18    }
19 }
```

In the i'th iteration, the task of the algorithm is to locate $head_{i+1}$ and then insert a new leaf edge at that location. To do that, it first considers $head_i$. Line 2 starts this procedure, considering the case when $head_i$ is the root. If true, $tail_i$ spells out the entire suffix. Thus to find $head_{i+1}$, increment the tail pointer to get the next suffix and scan a maximal prefix from the root. Otherwise suppose $head_i = aw$ and recall that w is a prefix of $head_{i+1}$. w can be found using suffix links but one first has to distinguish between the two cases where $head_i$ was created by the last phase and $head_i$ was created by some other phase. This is tested by the `is_head_old` function. If created by some other phase, then $head_{i+1}$ can directly be found directly by following the suffix link to w and then scanning $tail_i$. If $head_i$ was created by the previous phase, the problem is that $w = link(\overline{head_i})$ does not yet exist and has to be found in another way. Since $head_i$ is the only vertex that does not yet have a suffix link, its parents link can be used instead. This is exploited by `find_base`, which finds the location w.

```
1  static void find_base(STree *stree)
2  {
3     if (head_is_root(st)) {
4        if (label_empty(st->hd.label)) {
5           st->hd.label.end = NULL;
6        } else {
7           st->hd.label.start++;
8           skip_count(st);
9        }
10    } else {
11       set_head_to_suffixlink(st);
12       skip_count(st);
13    }
14 }
```

Suppose as in Section 5.2 that $u = \mathit{parent}(\mathit{head}_i)$ and let $av = \mathit{label}((u, \mathit{head}_i))$, then $w = \mathit{link}(u)x$. `find_base` first considers the case that head_i was created on a root edge. If x is empty, then nothing has to be done. Otherwise v is a prefix of head_{i+1} and the function uses $\mathit{skip\text{-}count}(r, x)$ to find w. If u was not the root, then the suffix link for u can be followed, and *skip-count* x down from $\mathit{link}(u)$ to find w. Once w is found with `find_base`, the next head can finally be found.

Now consider line 11 in `find_nxt_head`. If $\overline{w}$ is a vertex in $TR_i(t)$, then it was inserted in some prior phase and has *head-index* smaller than i. Hence, head_i is a large vertex with $\mathit{link}(\mathit{head}_i) = \overline{w}$. Suppose that the chain ending with head_i has length p and address q. Then, for each $d \in \{1, 2, \ldots, p-1\}$, the vertex at address $q - 2d$ is small and its distance to head_i is d. These distances are set with the function `finalize_chain`. Then, the new head is found by scanning $\mathit{tail}(i)$ from w as before. On the other hand if $\overline{w}$ is not a vertex in $TR_i(t)$, then $\mathit{head}_{i+1} = w$ (Section 5.2.3). Once inserted, $\mathit{head\text{-}index}(\mathit{head}_{i+1}) = i + 1$. Since $\mathit{head\text{-}index}(\mathit{head}_i) = i$, head_i is small and the current chain of small vertices can be expanded by one.

```
static void append_chain(STree *stree)
{
   if (!st->chain.fst) {
      init_chain(st);
   }
   st->chain.size += 1;
   st->is.nxt  += SMALL_VERTEXSIZE;
   st->is.nxt_ind += SMALL_VERTEXSIZE;
}
```

The procedures `skip_count` and `scan_tail` implement *skip-count* and *scan* without in a conventional way. For the details, refer to the full program. To find the suffix link for u, the procedure `follow_link` is used. It distinguishes between the root, small and large vertices. If the vertex is large, it looks up the suffix link with `SUFFIX_LINK`. Otherwise, the suffix link is the next vertex in the chain.

```
static Uint* head_suffix_link(STree *stree)
{
   VertexP root = st->is.fst;
   if (st->hd.depth == 1) {
      return root;
   } else {
      return root + SUFFIX_LINK(st->hd.v);
   }
}

void set_head_to_suffixlink(STree *stree)
{
   if (IS_LARGE(*st->hd.v)) {
      st->hd.v = head_suffix_link(st);
   } else {
      st->hd.v += SMALL_VERTEXSIZE;
   }
   st->hd.depth--;
}
```

The second part of the main loop is how to insert the new leaf edge labeled $tail_{i+1}$ to the tree. This is carried out by the `insert_tailedge` procedure, performing a mechanical and straightforward modification on the tree. For its implementation, see the full program.

7.4 *GK* in C

The program described in this section is based on [17] and referred to as `GK`. Due to its lazy nature, the algorithm best suited when only the existence of a pattern needs to be queried. Of course, the entire subtree (and thus all matching positions) can forcibly be evaluated once the match is found. This is however, not described in this section. The intent is instead again to highlight essential details and design choices in the implementation. For complementary code, see Appendix C and for the full program, see `https://github.com/lsund/lazy-suffix-trees`.

7.4.1 Basic Data Definitions

The basic building blocks for the suffix tree are represented by `Vertex` and `VertexP`, the same as in Section 7.3. Converting between addresses, indices and vertices are also done the same way as before. The first difference is how `GK` lays out the vertices in memory: because both inner and leaf vertices are processed in the same way, a single table suffices for representing the vertices.

```
typedef struct stree {
   // Vertices of the tree
   Table vs;
   // Root children
   Uint rs[MAX_CHARS + 1];
   // Is the root evaluated?
   bool root_eval;
} STree;
extern STree st;
```

The text is represented almost as in Section 7.3 although `GK` algorithm stores some additional information about the text during execution, used when evaluating vertices.

1. The suffixes of the text are represented as a list of pointers `ss` into the text itself.

2. The alphabet is represented as an array of characters `cs`, its size `asize` as an integer, and the maximal value in `cs`.

```
typedef Wchar *Suffix; // A suffix is an array of characters

typedef struct text {
   Wchar *fst;        // First character
   Wchar *lst;        // Last character
   Uint len;          // Length of the text

   // New fields
   Suffix *ss;        // Suffixes
   Uint asize;        // Size of the alphabet, based on the text
   Wchar cs[MAX_CHARS + 1]; // Characters in alphabetical order
   Uint maxc;         // Max value of a character
} Text;
extern Text text;
```

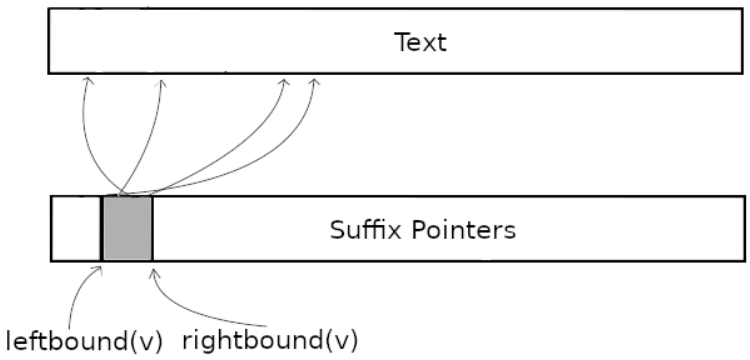

Figure 8: The `text.ss` component contains pointers into the text itself. The unevaluated suffixes of a vertex v lies in a specific scope within this component and is retrieved with the relations *leftbound*(v) and *rightbound*(v) as defined in Section 7.4.3.

7.4.2 Storing and Retrieving Edge labels

Let the relation *lchild*(v) denote the address of the leftmost child of an inner vertex v in $ST(t)$ and consider the edge $e = v \xrightarrow{x} lchild(v)$. Since x is a substring of t, it can be represented by the leftmost index where x occurs in t. Let *leftbound*(v) be the leftmost position of x and *rightbound*(v) =

leftbound$(v) + |x|$. Now observe that *rightbound(v)* = *leftbound*(*lchild(v)*), in other words *leftbound*(v) together with *lchild*(v) is sufficient for retrieving x.

How then, does one access labels of the subsequent children of v, i.e. the siblings of *lchild(v)*? The answer lies in how `GK` lays out the vertices in memory. Since the children for v are created sequentially, it is possible to position all children for for each newly evaluated vertex at consecutive addresses in the table `vs`. Now, once *lchild(v)* is known can all children of v be accessed implicitly by adding an offset to its pointer. *leftbound* and *lchild* are thus essentially the only relations needed in order to retrieve all edge labels, given that there is some way of detecting if a specific vertex is the last child or not.

For storing metadata, the most significant bits of a vertex are once again exploited. `SECOND_MSB` is used to determine whether or not a vertex is the rightmost child and `MSB` to determine whether the vertex is a leaf. `GK` also needs some way of distinguishing between evaluated and unevaluated vertices. This can also be encoded in `MSB` — since leaves cannot be evaluated further it would be redundant to use `MSB` for these. As before, `MSB` and `SECOND_MSB` need to be stripped off, before retrieving the left bound of a vertex.

```
1  // Create vertices with metadata
2  #define MAKE_LASTCHILD(V)      ((V) | SECOND_MSB)
3  #define MAKE_LEAF(V)           ((V) | MSB)
4  #define MAKE_LEAF_LASTCHILD(V) ((V) | MSB | SECOND_MSB)
5  #define MAKE_UNEVAL_VERTEX(V) ((V) | MSB)
6
7  // Accessing leftbound and rightbound (lchild)
8  #define LEFTBOUND(R)           ((*(R)) & ~(MSB | SECOND_MSB))
9  #define RIGHTBOUND(R)          (*((R) + 1))
10 #define CHILD(R)               RIGHTBOUND(R)
```

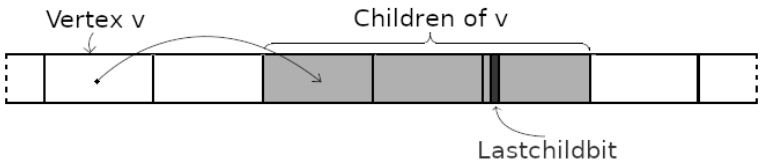

Figure 9: The children of a vertex v is always laid out sequentially in memory. Once the first child is known, the others follow implicitly. The last child is marked by setting its second most significant bit

7.4.3 Evaluating Unevaluated Vertices

Since C is a strict language, explicit synchronization between evaluated and unevaluated vertices need to be implemented. The basic idea is to use `MAKE_UNEVAL_VERTEX` whenever a new inner vertex $\overline{x}$ is inserted and evaluate it as follows. The subtree below $\overline{x}$ is determined by the set of all suffixes of t that have x as prefix. In order to evaluate $\overline{x}$, the program thus need a set of "unevaluated" suffixes $U(\overline{x}) = \{s \mid t \int xs\}$. Assuming this set is available for $\overline{x}$, the unevaluated suffixes for each child of $\overline{x}$ are obtained by first grouping $U(\overline{x})$ according to the first character of each suffix, i.e for each character $a \in \Sigma$, let $\{w \in \Sigma^* \mid aw \in U(\overline{x})\}$ be the a-group of $U(\overline{x})$. If an a-group contains only one string aw, the subtree beginning with w under $\overline{x}$ is a leaf edge labeled aw. If an a-group contains at least two strings, the algorithm computes the longest common prefix of all strings in this a-group; this becomes the label of a new edge leading to a new unevaluated inner vertex, whose unevaluated suffixes consist of the remainder of the strings in the a-group after the longest common prefix has been removed. The root-set of unevaluated suffixes, $U(r)$, consists of all suffixes of t.

The essential operation for generating new sets of unevaluated suffixes is grouping vertices according to their first character in some way. The method chosen here uses *counting sort* on first character values, resulting in a sequence of suffixes laid out in memory, sorted by to their first character, as desired. This sorted list of suffixes can then be scanned to in linear time generate all suffix groups. Note that using counting sort, there is no alphabet size factor to con-

sider. However, one has to assume an integer alphabet, that is, all characters can be uniquely mapped to an integer value.

The unevaluated suffixes are for each vertex represented as a continuous subsequence of `text.ss`. When the algorithm executes, parts of `text.ss` are sorted so that the a-groups for the corresponding vertices can be accessed. Sorting requires a second buffer before inserting the suffixes back into `text.ss`, represented by the `Sortbuffer` structure.

```
typedef struct sortbuffer {
   Suffix *fst;             // First element
   Uint groupsize[MAX_CHARS + 1]; // Groupsizes in the current vertex
   Uint size;               // Number of elements in the buffer
   Uint allocsize;          // Size allocated
} Sortbuffer;

extern Sortbuffer sb;
```

The subsequence of suffixes to be sorted is easy to obtain: the *leftbound* and *lchild* of a vertex determine the start and end address in `text.ss`.

```
#define SUFFIX_LEFTBOUND(R)    (text.ss + LEFTBOUND(R))
#define SUFFIX_RIGHTBOUND(R)   (text.ss + (CHILD(R) & ~MSB))
```

It is now possible to understand the procedures evaluating a vertex.

```
void eval_branch(VertexP vertex)
{
   Wchar **leftb;
   Wchar **rightb;
   get_unevaluated_suffixes(vertex, &leftb, &rightb);

   // The third argument to `insert_edges` is flagging whether or not
   // `vertex` is the root.
   insert_edges(leftb, rightb, false);
}
```

`eval_branch` takes a reference to a vertex and evaluates it by inserting the correct edges. It first calls `get_unevaluated_suffixes`, which stores the sorted subsequence of suffixes between the bounds `leftb` and `rightb`.

```
static void get_unevaluated_suffixes(
      VertexP vertex,
      Wchar ***leftb,
      Wchar ***rightb
   )
{
   *leftb = SUFFIX_LEFTBOUND(vertex);
   *rightb = SUFFIX_RIGHTBOUND(vertex);
   SET_LEFTBOUND(vertex, SUFFIX_INDEX(*leftb));
   CHILD(vertex) = INDEX(st.vs.nxt);
   counting_sort(*leftb, *rightb);
}
```

Counting sort is implemented in a straightforward way. The exact implementation is listed in Appendix C. When the unevaluated suffixes are known, `insert_edges` are used to extend the suffix tree.

```
static void insert_edges(Wchar **leftb, Wchar **rightb, bool isroot)
{
   if (!isroot) {
      alloc_extend_stree();
   }
   Uint *lchild;
   insert_edges_aux(leftb, rightb, &lchild, isroot);
   if (is_last_suffix(&rightb)) {
      insert_sentinel_vertex(rightb, &lchild);
   }
   if (isroot) {
      *st.vs.nxt = MAKE_LEAF_LASTCHILD(text.len);
      st.vs.nxt += LEAF_VERTEXSIZE;
   } else {
      *lchild = MAKE_LASTCHILD(*lchild);
   }
}
```

```
1
2  static void insert_edges_aux(
3        Wchar **leftb,
4        Wchar **rightb,
5        Uint **lchild,
6        bool isroot
7     )
8  {
9     Wchar **curr_suffix = NULL;
10    Wchar **group_rightb = NULL;
11    for (curr_suffix = leftb;
12         curr_suffix <= rightb;
13         curr_suffix = group_rightb + 1)
14    {
15       Wchar fst = **curr_suffix;
16       get_groupbound(&group_rightb, curr_suffix, rightb, fst);
17       *lchild = st.vs.nxt;
18       if (group_rightb > curr_suffix) {
19          insert_inner_vertex(fst, curr_suffix, group_rightb, isroot);
20       } else {
21          insert_leaf_vertex(fst, curr_suffix, isroot);
22       }
23    }
24 }
```

`insert_edges` dynamically allocates more space for the tree if needed, and performs a few complementary operations for special cases. The main work is done by `insert_edges_aux`, called on line 7. `insert_edges_aux` iterates over the entire subsequence given by `lbound` and `rbound`. In every loop, the loop `get_groupbound` increases the scanning pointer to the first suffix of a new group. Then, either `insert_inner_vertex` or `insert_leaf_vertex` is called, depending on if the group had multiple elements, or not. These procedures extend the tree by setting *leftbound* and *lchild* accordingly. For their implementation, see the full program listing. The last part of implementing *GK* is matching a pattern in the tree, calling `eval_branch` when needed. Its implementation is again close to what one would expect, for details see the program listing.

8 Experiments

The experiments were performed on a 64-Bit Arch Linux operating system, Linux kernel version 4.16.13–2. The CPU was a 4 Core Intel i5–6200U with 2.8GHz clock speed and the amount of RAM available was 8 Gb. The programs were compiled with the GNU C compiler version[15] 8.1.1 with optimization level O3. This compiler specified `sizeof(Wchar)` as 4 bytes, and `sizeof(Uint)`, `sizeof(Uint *)` and `sizeof(Wchar *)` as 8 bytes.

8.1 Data

The experiments in this section were carried out based on a large UTF-8 encoded XML document, called `members.xml`, consisting only of elements of the following structure:

```
<member>
    <firstname>FNAME</firstname>
    <lastname>LNAME</lastname>
    <phone>PHONE</phone>
</member>
```

This test data was chosen to resemble a typical "real-world" document, a database of members. From `members.xml`, a range of sub-documents were generated by copying its first 5, 10, 15, ..., 50 megabytes and naming the files `members5.xml`, `members10.xml`, `members15.xml` and so on, their name corresponding to their size in megabytes. One megabyte disk space could store around 1 000 000 characters. The experiments were carried out by measuring the resources required for constructing a suffix tree from various text inputs and then querying a large number of patterns on the resulting tree using the respective procedures from `MCC` and `GK`. The primary function of the pattern queries was to force the evaluation of the tree in `GK`, as it otherwise would not consume any resources to execute. The patterns were sampled uniformly at random over the entire `members.xml`. All experiments were performed ten times, and the figures display the average of the measurements.

[15] `https://gcc.gnu.org/`

8.2 Experiments

8.2.1 Fixed Patterns: Time

The execution time requirement of the programs was measured by recording a timestamp before initializing the suffix tree and after having queried all patterns. The time was recorded in milliseconds and then rounded to seconds with two decimal points. Figure 10 displays the time required for `MCC` and `GK` to query a fixed number of patterns with lengths in a small, fixed range. Both programs displayed a linear behavior, but `MCC` was more efficient, requiring only about two-thirds computational time compared to `GK`.

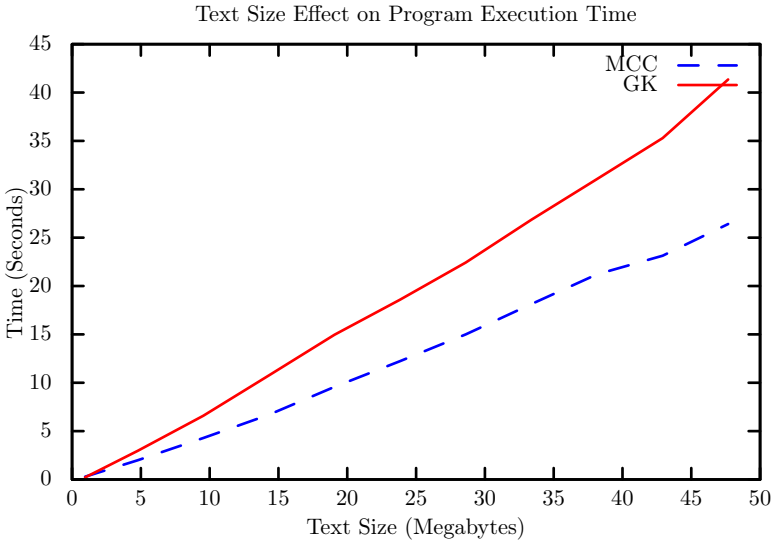

Figure 10: Time required to execute `MCC` and `GK` with inputs of different sizes. Both programs were queried 10 000 random patterns with lengths between 10 and 20 characters.

8.2.2 Fixed Patterns: Space

Space requirement were analyzed by counting the raw bytes allocated by the programs. `MCC` allocates space for

- $5q + 3p + l$ integers for the vertices where q is the number of large vertices, p the number of small vertices and l the number of leaves

- n characters for the text
- k pointers for accessing the root children

In these experiments, integers and pointers required 8 bytes and characters 4 bytes. Thus, `MCC` needs a total of $8(5q + 3p + l + k) + 4n + c$ allocated bytes, where c the extra, constant bytes needed to execute the program.

`GK` on the other hand allocates space for

- $2q + l$ integers for the vertices, where q is the number of evaluated inner vertices and l the number of evaluated leaves
- n characters for the text
- n pointers for the statically allocated array of suffixes
- Space to store the sort buffer `sb`, i.e. one pointer for each unevaluated suffix in the current vertex, bounded by n.
- k pointers for accessing the root children
- k integers for storing the sizes of the unevaluated suffix groups
- k characters for storing the alphabet

Thus, `GK` requires at most $8(2d + l + 2n + 2k) + 4n + 4k + c$ bytes, where c is the extra, constant number of bytes used by the program. The sort buffer is large when evaluating the top of the tree but never larger than any a-group. By also considering that n was much larger than k in these experiments, the space requirement was dominated by storing the text, vertices, and the static suffix array.

Figure 11 displays the memory allocated by `MCC` and `GK` when searching a fixed number of random patterns of lengths in a small, fixed range. Both programs displayed a linear space consumption, although `GK` required only about half the space compared to `MCC`. Without the inclusion of vertex chains (Section 7.3.4) however, the space profile would look even worse for `MCC`. For `members.xml` the average distribution between small and large vertices were approximately 5 to 1. This implies a chain length of 6, which reduces the total

space consumption to two-thirds of what it would have been, had all vertices been large. The distribution between small and large vertices differs from text to text though. The difference can be drastic, as shown by the following example: *aaaaaaaaaaaaaaaaaaaa*\$ results in 18 small and one large vertex while *aabbabaaababbaabaabb*\$ results in 3 small and 14 large vertices.

GK only evaluates the tree enough to match the current pattern, and does therefore require fairly little space, even when the number of patterns is large. If one wanted to compare the two programs space performance finding all *occurrences* of the patterns, then `GK` would be forced to evaluate the full subtree under each match, and the space consumption would look very different. Assuming that the patterns are many and not too long, `GK` could potentially need to evaluate the full tree. To get an idea of this situation, Figure 12 displays the memory allocated for the two programs when `GK` evaluates the full suffix tree. Here, `GK` even requires slightly more space than `MCC` even though each inner vertex can be stored using only two integers. The reason is the need to represent `text.ss`, storing one pointer for each character in the text. Note that one does not need to enforce the complete evaluation for `GK` in order to do this experiment since after running `MCC` the total number of inner vertices is known, and their bytes can simply be added to the total allocation number.

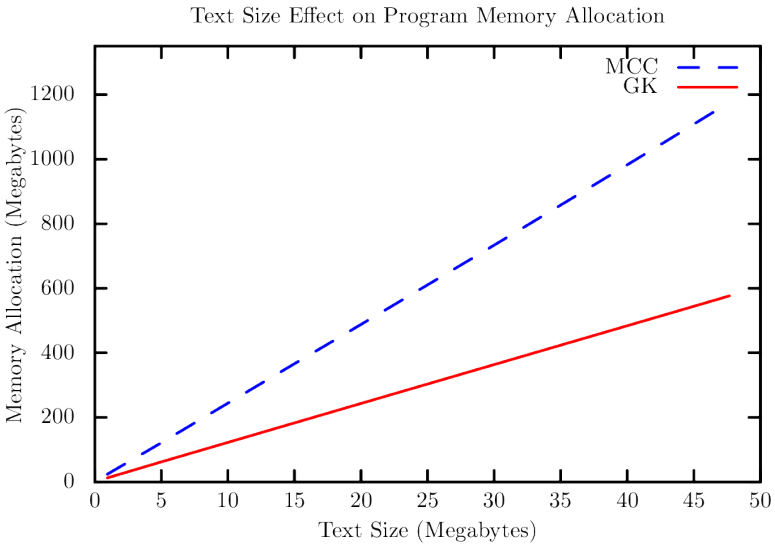

Figure 11: Space required when executing `MCC` and `GK` with inputs of different sizes. Both programs were queried 10 000 random patterns with lengths between 10 and 20 characters.

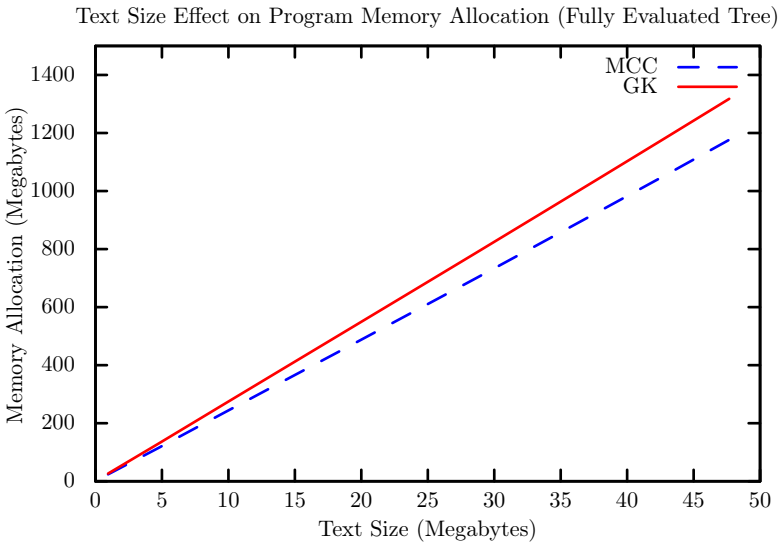

Figure 12: Space required when executing `MCC` and `GK` with inputs of different sizes. Both programs were queried 10 000 random patterns with lengths between 10 and 20 characters. The figure displays the situation when both programs fully evaluated their respective suffix tree.

8.2.3 Varying Pattern Lengths: Time and Space

So far in this section, only patterns of a small, fixed length have been queried. Figure 13 instead displays the result of querying the two programs a fixed number of patterns with varying lengths. Expectedly, querying patterns of longer lengths cause `GK` to run slightly slower, while `MCC` has steady performance. However, at a certain length, the runtime-curve for `GK` seems to flatten out, likely due to pattern overlap for very short patterns.

The same shape is seen in Figure 14, showing the number of allocated vertices with an increased pattern length. Again, short patterns are likely to overlap, and the algorithm does not need to evaluate the same amount of vertices as when querying longer patterns. Perhaps more interesting is that the curve stays flat for longer patterns, confirmed by the expreriments for patterns up to 100 000 characters long. This observation indicating that the search mechanisms are very fast for both programs once the relevant part of the tree has been evaluated.

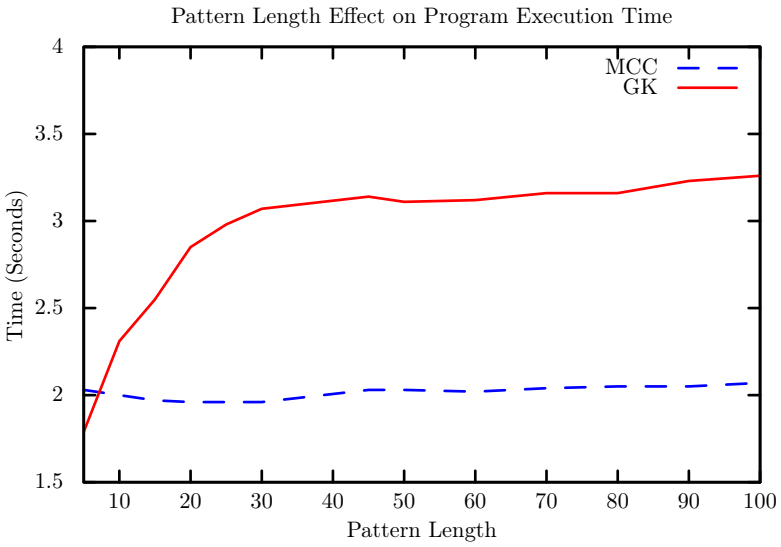

Figure 13: Time required for executing `MCC` and `GK` with input file `members5.xml`. Both programs were queried 10 000 random patterns of varying lengths.

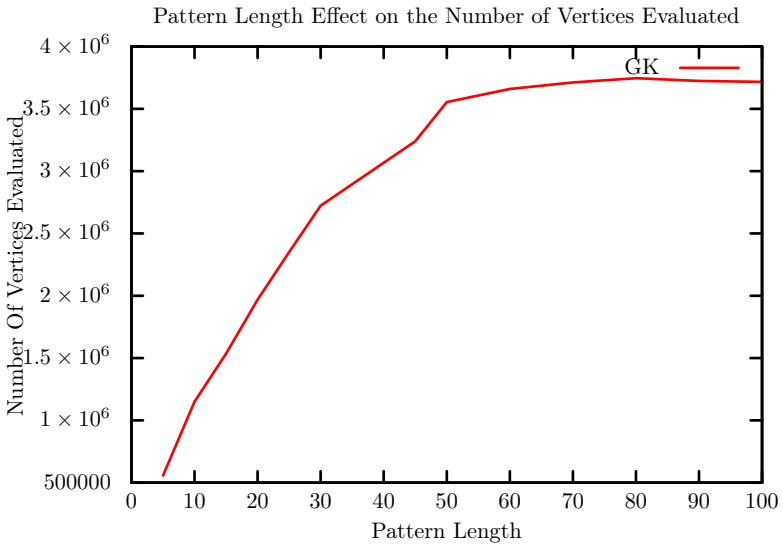

Figure 14: Number of evaluated vertices for `GK` with input file `members5.xml`. The program was queried 10 000 random patterns of varying lengths.

8.2.4 Varying Number of Patterns: Time and Space

Figure 15 displays the time requirement and Figure 16 the number of vertices evaluated when instead of changing the pattern length, one changes the number patterns queried for a specific file. As `MCC` always evaluates the full suffix tree, the number of evaluated vertices is always constant regardless of the number of patterns, and thus not shown in Figure 16. The number of patterns did also not influence the execution time of `MCC` by any significant amount, but `GK`'s runtime increased with an increased number of patterns, even if slight. The curve in 16 can seem steep, but the increase in evaluated vertices does in fact only correspond to a small part of the total memory allocated. This correspondence is seen in Figure 17, showing the memory allocation distribution for `GK`, averaged over all runs used to produce Figure 14 and 16. A similar chart for `MCC` is shown by Figure 18.

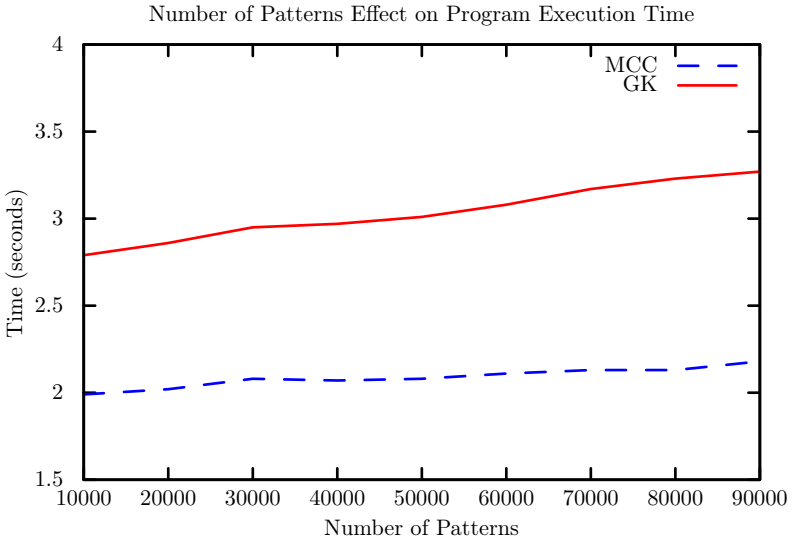

Figure 15: Time required for executing `MCC` and `GK` with input file `members5.xml` Both algorithm were queried a varying number of random patterns with lengths between 10 and 20 characters.

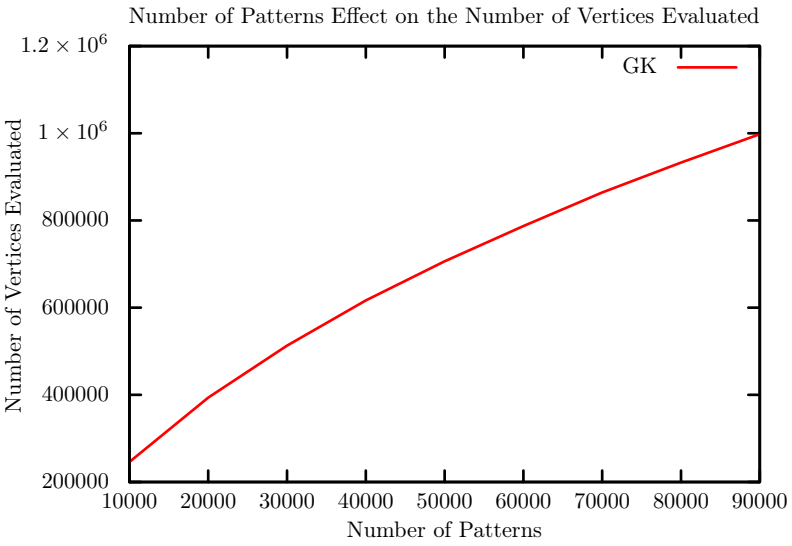

Figure 16: Number of evaluated vertices for `GK` with input file `members5.xml`. The program was queried a varied number of random patterns with lengths between 10 and 20 characters.

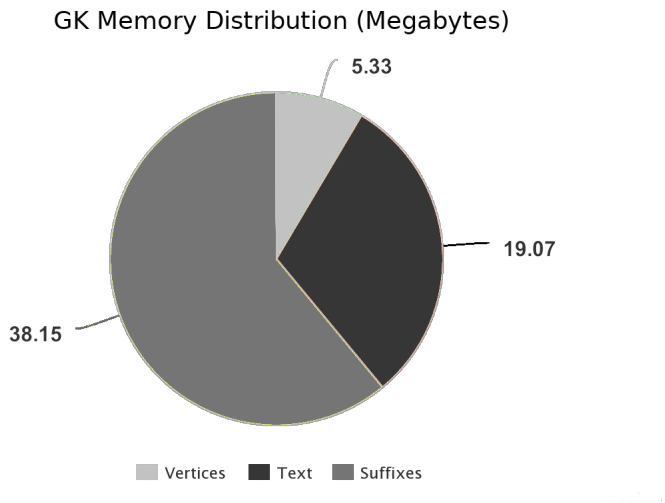

Figure 17: Memory allocation distribution for `GK` with input file `members5.xml`, averaged over all runs in figure 14 and 16.

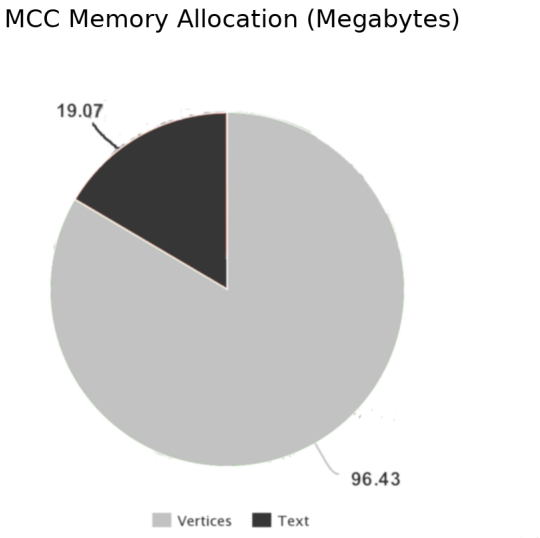

Figure 18: Memory allocation distribution for `MCC` with input file `members5.xml`.

8.2.5 Varying Alphabet Size: Time and space

The next experiment included a change in the text instead of the patterns, namely the size of the alphabet. Based on `members10.xml`, a set of new texts were generated by the following strategy: Each character within the content of `firstname`, `lastname` or `phone` tag were possibly replaced with a new character $c \notin \Sigma$ with probability $1/k$ where k is the size of the alphabet. Thus, after scanning the full text and maybe replacing the characters, the new character c was distributed uniformly at random over the entire document. This process was carried out 19 times, each time with a new unique character, to obtain a set of 20 XML documents with varying alphabet sizes.

As seen in Figure 19, the performance of `MCC` clearly worsen when constructing the suffix trees for texts with a larger number of unique characters. The curve does in fact look to be linear in the size of the alphabet, as expected due to the linked list implementation of accessing the children. `GK` is not affected by the alphabetic factor — its performs even faster as the alphabet grows. This is likely due to the length expected longest repeated substring, $\log_k(n)$, which decreases when increasing k. Since no suffix is read beyond the point it becomes unique, and because `GK` uses counting sort to group the suffixes the execution time decreases.

Figure 20 displays the same experiment but measuring space consumption instead of time. Interestingly, the space consumption decreases for `MCC` with an increased alphabet size, while the space consumption for `GK` is largely unaffected. Even if the tree becomes denser at the root for large alphabets could it become sparser in the lower parts. In particular, the average length of the leaf edges often becomes longer for larger alphabets. This explains why no significant change is observed for the space consumption of `GK`, as it mainly needs to evaluate the top part of the tree, but `MCC` operates in less space, as less overall inner vertices are needed.

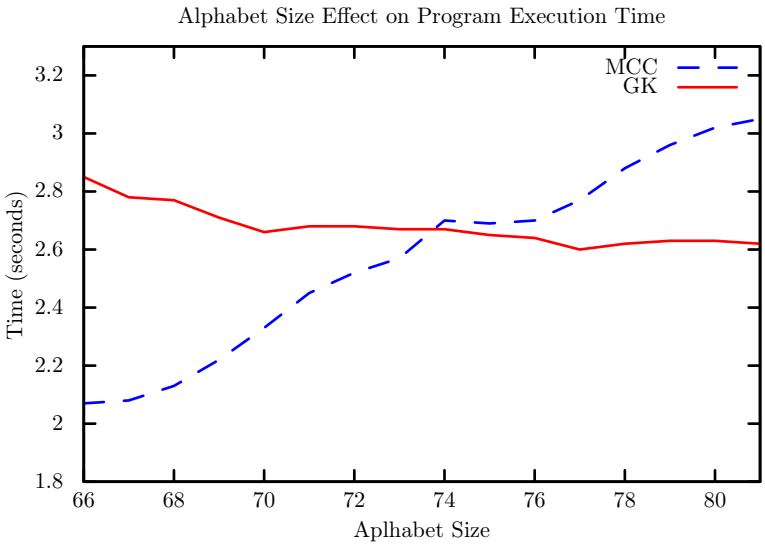

Figure 19: Time required for executing `MCC` and `GK` with input files with different alphabet sizes. Both programs were queried 10 000 random patterns with lengths between 10 and 20 characters.

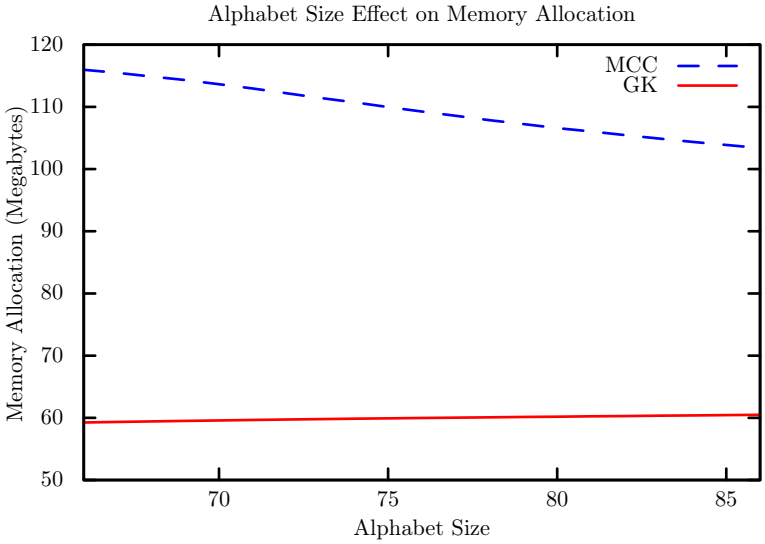

Figure 20: Space required when executing `MCC` and `GK` with 5 megabyte input files with different alphabet sizes. Both programs were queried 10 000 random patterns with lengths between 10 and 20 characters.

8.2.6 A Real-World Scenario

Since `members.xml` were chosen to replicate a real-world data document, it could be interesting to also replicate a more real-world related application of the data. Thus, a last experiment was performed on `members.xml`, but instead of querying random patterns (that may span parts of many XML tags), only the content of a specific tag was queried. `members.xml` were thus scanned and extracted for the contents of all `firstname` tags, each of which were strings drawn from the standard English alphabet of lengths ranging between 3 and 16. A subset of 100 000 patterns were then chosen uniformly at random from the full set of names and queried the two programs as before.

Figures 21 displays the result of this new experiment. The execution time of `MCC` largely the same as in Section 8.2.1 but the performance of `GK` changes drastically, as the patterns now are very similar, compared to before, a much smaller branch of the suffix tree needs to be evaluated in order to query the patterns.

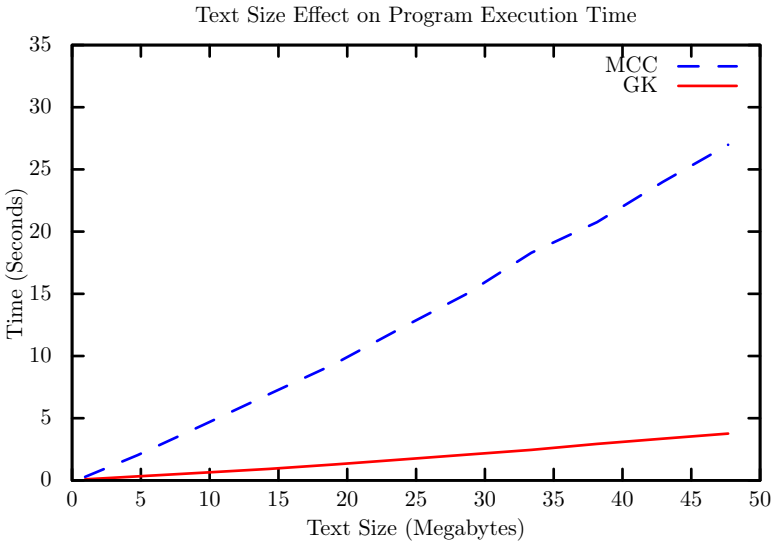

Figure 21: Time required for executing `MCC` and `GK` with the input file `members10.xml`. Both programs were queried 100 000 patterns consisting of the content of the `firstname` tags of `members.xml`.

9 Conclusions

Compared to other pattern matching methods, suffix trees generally require a large effort to understand and implement. The only algorithm of the four discussed in this thesis that can be reasonably understood without too much difficulty is *GK*. *Ukk* and *Mcc* are similar in many ways and can with benefit be discussed side by side. *Fch* is more intricate than its recursive counterpart *GK*, and is like the imperative algorithms based on local modifications of the tree.

The understanding of suffix tree algorithms often has to be translated into a computer program. The implementation process requires precision and accuracy for anything other than trivial applications. *Mcc* allows for exploiting the structure of the suffix tree by defining "chains", thus reducing space requirement significantly. *GK* has a simpler description compared to *Mcc* and is thus simpler to implement. There are, however, a few non-trivial details concerned with grouping suffixes and implementing lazy evaluation in a strict language. By using counting sort for grouping suffixes, *GK* becomes independent on the alphabet size, just like *Fch*. This is a good choice when the alphabet is of integer type and the difference between the delimiting integer values is not too large.

`MCC` requires on average 20 bytes per input character and `GK` requires 12 bytes per input character. `MCC` requires 4.3 seconds for creating a suffix tree from a standard, 10 megabyte "database-like" XML file. The time requirement of `GK` differs greatly considerably depending on what kind of patterns it is presented, because it only evaluates the part of the tree needed for the patterns. If the patterns are similar with respect to the entire text, `GK` can be extremely efficient, both in time and space, even if the full subtree has to be evaluated. For random patterns, it performs similar to `MCC` as long as only the first occurrence of the patterns has to be found. `GK`'s performance is practically linear, even if the number of patterns are very large and very long. An increasing alphabetical factor increases the time requirement for `MCC` considerably, but decreases its space requirement slightly. `GK` implemented with counting sort is not influenced negatively by character size — it performs better on longer alphabets.

Remembering "Occam's razor", the principle that the simplest solution often tends to be the right one, and taking the promising results of Section 8 into account, it is reasonable to claim that GK should be the first and foremost algorithm when considering suffix trees in practice. It has the simplest description and implementation, has the online property, is local in its execution and is independent of the alphabet size.

Acknowledgments

This thesis was written under the supervision of Prof. Dr. Norbert Blum, Department of Computer Science (Chair V) at the Rheinische Friedrich-Wilhelms-Universität Bonn. The grading of the thesis was refereed by Prof. Dr. Norbert Blum and Prof. Dr. Heiko Röglin.

Ingo Küper and Holger Flörke gave many insightful ideas regarding implementation and usage in a production environment. Michael Sundström and Helena von Dewitz gave many helpful comments on the text. Stefan Kurtz provided some important C code that helped to shape the two programs `GK` and `MCC`. The construction of the computer programs was supervised by Adrian Schmitz.

Appendices

A Full program: *GK* in Haskell

```
module SuffixTree.Algorithm.LazyTree where

import       Data.Function
import qualified Data.List       as L
import       Data.Text.Lazy      (Text, cons)
import qualified Data.Text.Lazy  as T
import       Prelude             (init, String)
import       Protolude           hiding (Text)

type SuffixList = [T.Text]

data Edge = Edge
   { _label :: Label
   , _subtree :: STree
   } deriving (Eq, Show)

data STree = Leaf | Branch [Edge] deriving (Eq, Show)


heads :: [Text] -> String
heads = foldr (\x acc -> if T.null x then acc else T.head x : acc) []

removeHeads :: [Text] -> [Text]
removeHeads []   = []
removeHeads("" : xss) = removeHeads xss
removeHeads(xs : xss) = T.tail xs : removeHeads xss

filterSuffixes :: Char -> [Text] -> [Text]
filterSuffixes c = map T.tail . filter (\x -> T.head x == c)
```

```
dropCommonPrefix :: SuffixList -> (Int64, SuffixList)
dropCommonPrefix []  = (0, [""])
dropCommonPrefix ("" : xss) = dropCommonPrefix xss
dropCommonPrefix [s] = (T.length s, [""])
dropCommonPrefix (xs : xss) =
   let
      (h, t) = (T.head xs, T.tail xs)
      (lcp, xs') = dropCommonPrefix (t : map T.tail xss)
   in
      if compareHeads h xss
         then (succ lcp, xs')
         else (0, xs : xss)
   where
      compareHeads c = all ((==) c . T.head)


lazyTree :: Text -> STree
lazyTree x = lazyTree' (init $ T.tails x)
   where
      lazyTree' [""] = Leaf
      lazyTree' suffixes =
         Branch (foldr' (appendEdge suffixes) [] (L.nub $ heads
             suffixes))
      appendEdge suffixes a allEdges =
         let
            aSuffixes = filterSuffixes a suffixes
            (lcp, rests) = dropCommonPrefix aSuffixes
         in
            case aSuffixes of
               (mark : _) -> makeEdge mark lcp rests : allEdges
               []         -> allEdges
         where
            newLabel mark lcp = Label (a `cons` mark) (succ lcp)
            descendTree = lazyTree'
            makeEdge mark lcp rests = Edge (newLabel mark lcp)
                                           (descendTree rests)
```

B Program extracts from *MCC* in C

The definitions of `HeadLoc`, `SplitLoc` and `Chain` are listed below.

```
// A HeadLoc represents location of the current head.
typedef struct headloc {
   // The parent vertex of the edge containing the head
   // location.
   VertexP v;
   // The characters from the head location to the end of
   // the edge label it resides in. If the location is the
   // last character of the edge then the end component of
   // `l` is NULL.
   Label l;
   // The depth of the head location, that is, the
   // label-depth of the string it represents
   Uint d;
} HeadLoc;

// A SplitLoc represents the edge that is to be splitted
// after finding the new head location.
typedef struct splitloc {
   // Represents the endpoint (i.e) the child vertex of the
   // edge that is about to be split.
   Vertex child;
   // Refers to the branching vertex to the left of the
   // inner vertex about to be split.
   Vertex left;
} SplitLoc;

// A Chain is a sequence of small vertices terminating in a
// large vertex. The structure stores a reference to the
// address of the first vertex in the chain as well as the
// number of vertices in the chain.
typedef struct chain {
   VertexP fst;
   Uint size;
} Chain;
```

C Program extracts from *GK* in C

The module performing counting sort on intervals of suffixes.

```
#include "sort.h"

// The current sortbuffer
Suffix   *curr_sb;

// Determine the group sizes for the suffixes between left
// and right
static void set_groupsize(
      Suffix *left,
      Suffix *right, Uint
      plen
   )
{
   Suffix *curr_suffix;
   // Iterate the interval
   for (curr_suffix = left; curr_suffix <= right; curr_suffix++) {
      // drop the common prefix
      *curr_suffix += plen;

      Uint fst = **curr_suffix;
      sb.groupsize[fst]++;
   }
}
```

```
// Set the upper and lower bounds for each group, delimited
// by left and right
static void set_group_bounds(Suffix *left, Suffix *right,
      Wchar ***upper_bounds)
{
   // `curr_sb` is already allocated, a sufficiently large
   // memory block for all suffix pointers.
   Suffix *lower_bound = curr_sb;
   Suffix *curr_suffix;

   // Iterate the interval
   for (curr_suffix = left; curr_suffix <= right; curr_suffix++) {

      Uint fst = **curr_suffix;
      if (sb.groupsize[fst] > 0) {
         // 'allocate' the upper bound for the current
         // character. upper_bounds[fst] now points to a
         // allocated memory address, enough space in
         // distance from the last group.
         upper_bounds[fst] = lower_bound + sb.groupsize[fst] - 1;
         lower_bound   = upper_bounds[fst] + 1;
         sb.groupsize[fst] = 0;
      }
   }
}

// Insert the suffixes into the correct positions, depending
// on `upper_bonuds`
static void insert_suffixes(
      Suffix *left,
      Suffix *right,
      Wchar ***upper_bounds
   )
{
   Suffix *curr_suffix;
   for (curr_suffix = right; curr_suffix >= left; curr_suffix--) {
      // This fills up the slot allocated for this group
      // with suffix tree addresses, end to start.
      Uint fst         = **curr_suffix;
      *(upper_bounds[fst]--) = *curr_suffix;
   }
}
```

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.

[2] C. Allen and J. Moronuki. *Haskell Programming.* Online version, `http://haskellbook.com`, 2018.

[3] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software — Practice and Experience*, 25(2):192–141, 1995.

[4] A. Apostolico. The myriad virtues of subword trees. *[6]*, pages 85–96, 1985.

[5] A. Apostolico, M. Farach, M. Crochemore, and S. Muthukrishnan. 40 years of suffix trees. *Communications of the ACM*, 59(4):66–73, 2016.

[6] A. Apostolico and Z. Galil. *Combinatorial Algorithms on words.* Springer Verlag, 1985.

[7] A. Apostolico and W. Szpankowski. Self-alignments in words and their applications. *J. Algorithms*, 13:446–467, 1992.

[8] J. Vygen B. Korte. *Combinatorial Optimization Theory and Algorithms.* Springer Verlag, Berlin Heidelberg, Germany, 2002.

[9] M. Crochemore. *Algorithms on Strings.* Cambridge University Press, 2007.

[10] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings.* Cambridge University Press, 2001.

[11] M. Crochemore and W. Rytter. *Jewels of Stringology: Text Algorithms.* World Scientific Publishing Company, 2002.

[12] M. Farach. Optimal suffix tree construction with large alphabets. pages 137–143, 1997.

[13] M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. *[30], Lecture Notes in Computer Science*, 1099.

[14] R. Giegerich and S. Kurtz. Suffix trees in the functional programming paradigm. *ESOP*, pages 225–240, 1994.

[15] R. Giegerich and S. Kurtz. A comparison of imperative and purely functional suffix tree constructions. *Sci. Comput. Program*, 25(2-3):187–218, 1995.

[16] R. Giegerich and S. Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. 19(3):331–353, 1997.

[17] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Soft., Pract. Exper.*, 33(11):1035–1049, 2003.

[18] D. Gusfield. An "increment-by-one" approach to suffix arrays and trees. *Report CSE-90-39, Computer Science Division, University of California*, pages 192–141, 1990.

[19] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, 1997.

[20] D. Harel and R. H. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Computing*, 13(2):338–355, 1984.

[21] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2), 1989.

[22] S. P. Jones. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13, 2003.

[23] T. kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. *CPM Lecture Notes in Computer Science*, 2089, 2001.

[24] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall Ptr, Upper Saddle River, New Jersey 07458, 1988.

[25] D. Knuth, J. Morris, and R. Pratt. Fast pattern matching in strings. *SIAM J. Comput*, 6(2):323–350, 1974.

[26] S. Kurtz. Reducing the space requirement of suffix trees. *Softw., Pract. Exper*, 29(13):1149–1171, 1999.

[27] S. Kurtz. Fundamental algorithms for adeclarative pattern matching system. *Forchungsberichte der Technischen Fakultät, Abteilung Informationstechnik / Universität bielefeld*, Report 95-03, Bielefeld: Univ.1995.

[28] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *Proceedings of the 1st ACM-SIAM Annual Synposium on Discrete Algorithms*, pages 319–327, 1990.

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

[30] F. Meyer and B. Monien (eds). *Automata, Languages and Programming. ICALP*. Springer Verlag, Heidelberg, 1996.

[31] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Comput.*, 60(10):1471–1484, 2011.

[32] Survey of most used programming languages. `https://insights.stackoverflow.com/survey/2018/#technology-programming-scripting-and-markup-languages`. Online, accessed 2018-05-16.

[33] S. S. Skiena. Who is interested in algorithms and why? *Proceedings of the 2nd Workshop on Algorithm Engineering (WAE)*, pages 204–212, 1998.

[34] B. Smyth. *Computing Patterns in Strings*. Pearson Education Limited, Harlow, 2003.

[35] D. Turner. An overview of miranda. *ACM SIGPLAN Notices*, 21(12):158–166, 1986.

[36] D. Turner. Some history of functional programming languages. *Trends in Functional Programming. TFP 2012. Lecture Notes in Computer Science*, 7829:1–20, 2013.

[37] E. Ukkonen. On-line construction of suffix-trees. *Algorithms, Software, Architecture. J.v.Leeuwen (Ed.), Information processing*, 1(92):484–492, 1992.

[38] C. P. Wadsworth. Semantics and progmatics of the lambda calculus. *PHD. Thesis*, 1973.

[39] P. Weiner. Linear pattern matching algorithms. *IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.