Implementation and Comparison of suffix tree representations

Alexander Aumann
Bachelorarbeit in Informatik

Implementation and Comparison of suffix tree representations

Implementierung und Vergleich verschiedener Suffixbaum-Repräsentationen

Author: Alexander Aumann
Supervisor: Prof. Dr. Ernst W. Mayr
Advisor: Dipl.Inf. Johannes Krugel
Date: November 15, 2011
I assure the single handed composition of this bachelor’s thesis only supported by declared resources.

Munich, November 15, 2011

Alexander Aumann
Abstract

Searching through large amounts of sequence data plays an important role in computer science in general and in bioinformatics specifically. Thus efficient indexing techniques are required, one of them being the suffix tree. Suffix trees are very large though in relation to the input sequence and this becomes a growing problem as the gap between disk storage capacity and memory size increases. Luckily there has been intensive research on this topic in the last 20 years and the purpose of this paper is to present some of the latest ideas for disk based suffix trees as well as implementing one memory based and one of the proposed disk based algorithms within the SeqAn library which is free to use for everybody who in need for sequence analysis tools. These implementations are finally evaluated, with a strong focus on the implementation of the disk based algorithm, the STTD64 proposed by Halachev, to show that they are of practical usage.
Contents

Abstract iv

I. Introduction and Background Theory 1

1. Introduction 2
   1.1. Suffix trees as a means of searching large strings of text ...................... 2
   1.1.1. Terminology .............................................................................. 2
   1.1.2. Pattern Search and its worst case performance .............................. 2
   1.1.3. Suffix trees .............................................................................. 3

II. Indices 6

2. Indices 7
   2.1. Kurtz’s space reduced suffix tree .................................................... 7
       2.1.1. How to store the suffix tree information? .................................. 7
       2.1.2. The Scheme of nodes in the Kurtz tree - A first linked list implemen-
               tation .................................................................................. 8
       2.1.3. Small nodes and large nodes ..................................................... 9
       2.1.4. The improved linked list implementation .................................... 10
       2.1.5. The construction algorithm ..................................................... 11
       2.1.6. Possible extensions and limitations .......................................... 11
   2.2. STTD64 .......................................................................................... 12
       2.2.1. How to store the nodes? ............................................................ 12
       2.2.2. The Tree Top Down Construction algorithm .............................. 14
       2.2.3. Partitioning and buffers ............................................................ 15

III. Implementation 20

3. Implementation 21
   3.1. The SeqAn library ............................................................................ 21
       3.1.1. Template subclassing ................................................................. 21
       3.1.2. SeqAn classes correlated to suffix tree implementation ............... 23
       3.1.3. Useful data structures ............................................................... 24
   3.2. The implementation of the space reduced suffix tree ......................... 25
       3.2.1. The Index class requirements ..................................................... 25
   3.3. The implementation of STTD64 ....................................................... 29
4. Evaluation and future work
   4.1. Setup
   4.2. Comparison
      4.2.1. Comparing my implementation of STTD64 to the reference implementation
      4.2.2. Constructing the suffix tree for larger data size
      4.2.3. The construction time for the Kurtz index
      4.2.4. What still remains undone
      4.2.5. A remark on the leaf depth filling
      4.2.6. Conclusion
Part I.

Introduction and Background Theory
1. Introduction

There are many applications that require an efficient method of searching a text for the appearance of a certain pattern. The most obvious one maybe is the full text search functionality offered by a text editor or web browser. While in these cases the text to be searched is reasonably small to find patterns very fast in with just a sophisticated algorithm there also are applications where this would take far too much time and thus some kind of indexing structure is required. The entire human genome downloadable at [9] for example requires nearly 3 GB of space and is over 3 billion characters long. Researchers willing to analyse the structure of the human genome need methods to quickly find patterns even in such a large text and suffix trees provide these, as will be shown in the following section.

1.1. Suffix trees as a means of searching large strings of text

This section briefly outlines the use of suffix trees especially in applications of bioinformatics and their main advantages (e.g. compared to other indexing structures) as well as how they are composed and structured in general. The first part will define some vocabulary needed for discussing suffix trees.

1.1.1. Terminology

Before reasoning about pattern search let us fix the terminology required. As far as this work is concerned the (large) text to be searched will be referred to as the string $S$ and its length $|S|$ will be called $n$. The length of the (smaller) pattern $P$ that is searched for within the string will be called $m$. A string is a concatenation of characters (or letters) $x_0 x_1 \ldots x_{n-1}$ over an alphabet $\Sigma$ with the special letter $\epsilon$ denoting the empty string (or word). Throughout this paper lower case letters will be used for single characters ($a, b, c, d, \ldots$) while upper case letters will be used for (sub)strings (in addition to $S$ and $P$ mainly $U, V, W, X$ and $Y$). Of a string $S = UVW U$ is called a prefix of $S$ and $W$ is called a suffix of $S$. $W$ is a proper suffix of $S$ if and only if $UV \neq \epsilon$ and vice versa for proper prefixes. The suffix $x_j \ldots x_{n-1}$, i.e. the suffix starting at position $j$ in the string, is abbreviated as $S_j$.

When describing a tree structure representing some string $S$ the following notations are fixed. Nodes are named with lower case letters $u, v$ and $w$. A node $w$ in the tree, which represents the substring $aW$ is also simply denoted as $(aW)$. A tree edge labelled $U$ outgoing from a node $v$ and leading to a node $w$ is represented by $v \xrightarrow{U} w$. When presenting a string, i.e. its actual characters, this string will be put in single quotes: ‘This is a string’.

1.1.2. Pattern Search and its worst case performance

There are shortly spoken two approaches for quickly finding a pattern in a string, the first one being preprocessing the pattern and the second one being preprocessing the text.
1. Introduction

One algorithm of the first category is the *Knuth-Morris-Pratt-Algorithm* which already has a linear worst case runtime of $O(m + n + k)$, with $k$ being the number of occurrences of the pattern in $S$. Details on the algorithm can be found e.g. in [4, p. 89]. Note, however, that the most simplistic approach of simply aligning the pattern with the first character of the text and comparing character by character until a mismatch is found and then successively shifting the pattern one character to the right would have a worst case runtime behaviour of $O(nm)$. The idea of optimized algorithms like the *Knuth-Morris-Pratt* is to do the shifting part in a more sophisticated way using the information gained by preprocessing $P$.

1.1.3. Suffix trees

The most popular method of the second category is indexing the text with the use of suffix trees. Such a tree can be constructed in $O(n)$ worst case time and allows for searching a pattern in $O(m + k)$ time leading to the same worst case one time search performance as the algorithm mentioned before. The real virtue of suffix trees is, however, that the preprocessing of the usually much larger string $S$ has only to be carried out once and thereafter provides a search time bound by the size of the relatively small pattern $P$. This allows to efficiently solve a vast number of more sophisticated string problems and off course simply searching for various different patterns in a large string like the human genome.

What exactly is a suffix tree now? A suffix tree of a string $S$ is a tree of all suffixes of $S$ with the following characteristics:

- The tree has exactly $n$ leaves, numbered from 0 to $n - 1$
- Every tree edge is labelled with a substring of $S$ such that each path from the root to a leaf spells the suffix of $S$ beginning at index $i$ when concatenating the edge labels on that path.
- Each inner node of the tree except for the root has at least two children where the first character of each child edge label must be distinct
- Optional: Many construction algorithms additionally provide (and rely on) suffix links. Consider a node $(aW)$: the suffix link of this node is defined as a link to the node representing its suffix, i.e. the node $(W)$.

Figure 1.1 shows such a suffix tree for the DNA sequence ‘AGAGAGCTT$’ enhanced with suffix links.

An interesting fact to note is that such a suffix tree does not exist for every string. It does, however, for every string where no suffix is a proper prefix of another suffix as then the one to one relationship between suffixes and leaves is assured. To be able to construct the suffix tree for an arbitrary string one (virtually) appends a so called *sentinel character* to the string, i.e. one character that is guaranteed not to appear in the original string. In the above example this sentinel character is denoted as ‘$’ and leaving it out would lead to a tree with no leaf corresponding to the suffix ‘T’.

Before going into further detail on the construction of a suffix tree in linear time let us take a closer look at its benefits. What is clearly visible is that, as soon as such a suffix tree has been created for $S$, a pattern $P$ can be found in time $O(m)$. One only has to descend the
Figure 1.1.: The suffix tree for ‘AGAGAGCTT$’. Suffix links are represented by dashed red arrows.

tree character-wise in the direction $P$ suggests until there either is one character in $P$ that cannot be reached from the current node or the pattern $P$ has been completely consumed. In the first case the pattern is not part of the text, in the second case it is and all occurrences can be found by visiting every leaf that can be reached from the current node. This last step is guaranteed to be in $O(k)$ because every node has at least 2 children and thus the tree depth is directly linked to the number of occurrences of $P$ (note that the string depth, i.e. how long the real suffix is, may of course be significantly greater).

**On the construction of suffix trees**

One naive idea of constructing a suffix tree would be to simply start with the suffix tree containing only the suffix starting at the first position, i.e. the whole string. The second step would be to insert the suffix with length $n - 1$ and so on. To insert a new suffix one would simply descend the tree starting at the root by matching the suffix along the edges of the existing tree until it is no longer possible and then insert the suffix at this position. This would either be done by adding a new child to an inner node or by inserting an inner node to split the edge label at the appropriate position. This approach would however be of quadratic complexity as which can easily be seen when looking at a string like ‘aaaaaa$’. The whole existing tree would always have to be traversed to insert a new suffix.

As suffix trees are of practical importance mostly where the string $S$ is relatively large, a runtime of complexity $O(n^2)$ is a huge drawback. Luckily there are, as already mentioned, algorithms that allow for linear suffix tree construction time.

The first one was already introduced in 1973 by Weiner [13] but was quite complex and
1. Introduction

difficult to understand. Later on McCreight developed an improved algorithm that is easier to understand as well as less space-consuming during construction time [8]. Ukkonen provided a linear “online” construction algorithm that is able to construct a suffix tree up front, i.e. beginning with the first character of the string followed by the second one and so on [12]. Finally in [3] it was shown that each of these methods is in essence based on the same algorithmic idea and that McCreight’s algorithm is the most efficient in terms of construction time space requirement and construction time itself, with a slight edge over Ukkonen. In [7] a further improvement on McCreights algorithm was proposed which especially reduces the space requirement for the full suffix tree by a sophisticated node representation. Details on this improved algorithm by Kurtz will be shown in part 2.

Drawbacks and limitations of the linear algorithms

Though linear construction time seems at first glance far superior to quadratic construction time there is one problem with these linear time algorithms. They all expect that the whole suffix tree rests in main memory and heavily use random access to construct the tree. This slows them down massively for very large input strings as, when hard disk accesses are concerned, random access leads to great search time overhead compared to transferring successive chunks of data. It has been shown that some quadratic algorithms have a far reduced construction time as soon as the tree does not entirely fit in the main memory due to greater locality in their node accesses during build time. Moreover many of these algorithms may have a worst case runtime of \(O(n^2)\) but in practice average at \(O(n \log n)\).

The rest of this paper will be structured as follows. In the following chapter two index structures are presented. The first will be a memory based index proposed by Kurtz, the other one is a disk based structure, namely STTD64 introduced originally by Halachev.
Part II.

Indices
2. Indices

In this work I will implement one memory based index data structure and one that is disk based. This chapter presents both of them in detail before the implementation is presented in chapter 3. The first part describes the memory based idea of the “space reduced suffix tree” proposed by Kurtz in [7]. After that the disk based approach of the “STTD64” originally described by Halachev in [5].

2.1. Kurtz’s space reduced suffix tree

Stefan Kurtz’s paper of 1998 proposes an optimized storage of the information stored in the suffix tree nodes to save a significant amount of space. The construction of the tree itself is thereafter based on the algorithm of McCreight. This section outlines the central ideas of the algorithm and gives a glimpse on the implementation detail. Moreover many general suffix tree ideas are presented, so this section thus presents a good overview of these, too.

2.1.1. How to store the suffix tree information?

As described in the introduction chapter there have in principle to be stored the following information in each node:

- The children of the node
- The characters that label the edge to each child
- The suffix link of the node.

The first thing that every practical suffix tree implementation uses is, that instead of the edge label string only the start and (often times indirectly) end position of the label are stored in the nodes. This way the size for one node is constant (and the overall tree size is a multitude smaller). The letters are retrieved by looking up the positions in the original text string. Instead of storing the end position of each edge in e.g. the child node these are usually retrieved indirectly. A common approach is that the child node stores only the edge start position and each node additionally contains its own depth which already reduces memory consumption for a node with only two children, which is the minimum number of children, and the effect increases with the number of children. The positions delimiting an edge can then easily be determined by the start position and the difference between the depths of father and child node.

One thing to consider is that the start position of an edge from one node to another is not clearly defined. Consider the tree in image 1.1: The leftmost child node of the root (with the incoming label ‘AG’) might have start position 0, 2, or 4 as ‘AG’ is a prefix of all the
2. Indices

Indices suffice starting at these positions. The common approach to this ambiguity was to choose the position of the first occurrence of the label, which would in the example be 0. Kurtz, however, proposes a different convention, namely to use the index of the first branching occurrence. That means the start position of the one suffix the insertion of which caused the creation of the node. In the example tree this would be the position 2 and Kurtz called this position the headposition of a node. The difference in defining the start position and how it is stored in a node, minor as it might seem, has huge practical purpose, as this can be used to reduce the size of the whole suffix tree, as will be shown in subsection.

2.1.2. The Scheme of nodes in the Kurtz tree - A first linked list implementation

The idea is to use two different types of structures, one for leaves and one for nodes. Each node consists of five integer values:

- The depth of the node
- The headposition of the node
- The branchbrother reference, i.e. the reference to the right brother of the node or a nil reference if the node has no right brother
- The suffix link of the node
- The firstchild reference, i.e. the reference to the leftmost child of the node

It is easy to see, that this suffices to describe the whole suffix tree structure but let us illustrate what the content of a node would look like at the example of the aforementioned node with incoming edge AG: The first child reference of this node would refer to node AGAG, its branchbrother reference would point to node G, as would its suffix link. Its headposition is 2 and coincidentally so is its depth.

A leaf only needs to store one information item: the start position of the suffix it represents. What is to be still shown yet is how this information can be used to represent every edge accurately.

If there are two nodes connected with an edge in the following form \( w \xrightarrow{u} wu \), then \( u \) is \( x_{i \ldots i + l - 1} \) where \( i = wu_{\text{headposition}} + w_{\text{depth}} \) and \( l = wu_{\text{depth}} - w_{\text{depth}} \).

If there is an edge \( w \xrightarrow{j} wu \) where \( wu \) is the leaf with index \( j \) (i.e. it represents the suffix \( S_j \) ) then \( u \) is \( x_{i \ldots j \ldots} \) where

\[
i = j + w_{\text{depth}}
\]

To store the nodes and leaves in the main memory two tables are used. \( T_{\text{Nodes}} \) stores all nodes and thus all references to a node are simply a position in this table. \( T_{\text{Leaves}} \) is the table for storing all leaves, so a reference to a leaf refers to a position in this table. All of this implies that a reference to a leaf is signalled by a special bit placed in the branchbrother and firstchild references.

Now that the ground work has been laid and it has been shown that the head position approach works the next subsection shows how it can be used to reduce the space of the suffix tree by critical observation.
2. Indices

2.1.3. Small nodes and large nodes

There are two important observations about headpositions:

1. If two nodes \( w \) are different branching nodes, then \( w_{\text{headposition}} \neq u_{\text{headposition}} \).

2. If \( aw \) is a branching node, then \( w \) is also a branching node and the inequality
   \( aw_{\text{headposition}} + 1 \geq w_{\text{headposition}} \) holds.

The first part of the observation is obvious as this is more or less directly implied by the definition of the headposition. That the existence of a branching node \( aw \) implies the existence of another branching node \( w \) can easily be understood: if a branching node \( aw \) exists then there are two characters b and c so that the words \( awb \) and \( awc \) are also contained in the tree, and so are \( wb \) and \( wc \). This implies that there must also be a branching node \( w \). The proof of the inequality is not that easy to grasp and as only the result is important for understanding the implementation I recommend to look up the original proof at [7, p.11], if the reader is in need for a precise proof.

These two observations can now be used to reduce the space consumption of a suffix tree by first discriminating all nodes into one of two categories: a node \( aw \) is small if and only if \( aw_{\text{headposition}} + 1 = w_{\text{headposition}} \) otherwise it is a large node. The following holds now for a small node:

- The suffix link for the small node \( aw \) refers to the node \( w \) where \( aw_{\text{headposition}} + 1 = w_{\text{headposition}} \).
- The depth of the small node is the depth of node \( w + 1 \).

In summary this means that a small node stores redundant information as there is a strict correlation between headposition of the node, its depth and its suffix link. The next step now is to use this observation to decrease the space consumption of such a small node. We begin by defining a so called “node chain”:

A node chain is a chain of nodes \( b_{l-p}; \ldots; b_{l-1} b_l \) where

- \( b_{l-p-1} \) is a large node
- \( b_{l-p}; \ldots; b_{l-1} \) are small nodes and
- \( b_l \) is the large node at the end of the chain

\( p \) is called the length of the node chain. Now, for each small node \( b_x \in [b_{l-p}; b_l] \) and \( d \in [1; p] \) (the distance of a small node to the big node at the end of the chain \( d = l - x \)) the following facts can easily be verified:

- \( b_x_{\text{sufflink}} = b_{x+1} \)
- \( b_x_{\text{depth}} = b_{l_{\text{depth}}} + d \)
- \( b_x_{\text{headposition}} = b_{l_{\text{headposition}}} - d \)

Consequently instead of storing the depth, the suffix link and the headposition of a small node only the distance to the large node at the end of the chain has to be stored, while the rest can be calculated in constant time, thereby saving two integers worth of space.
2. Indices

2.1.4. The improved linked list implementation

To implement the idea of small and big node, Kurtz proposes the following bit layout shown in figure 2.1.

![Figure 2.1.: The node layout for the implementation of Kurtz's suffix tree](image)

Only large nodes consist of the full four integers, for small nodes the first two integers suffice. Only five bits are reserved for storing the distance of a node to the last node of a node chain, as chains of a length greater than 32 are rare in practice. If a longer chain should occur nonetheless an artificial big node is inserted. There are 29 bits for the firstchild reference (distributed over integer 1 and 2) and for the branch brother reference. The first of these bits is in each case reserved to indicate if the reference goes to a leaf or to a branching node (as these are in different tables).

There is one more space reducing trick implemented around the storage of the suffix link of a (large) node. As the last child of a node has no branch brother (it is itself the rightmost node), this space can be used to store the suffix link of the father (only one extra bit is necessary to mark the branch brother reference as a “nil reference”, indicated by the ↓ in integer 2). Note, that this would not even mean a theoretical performance decrease while building the suffix tree, as there is already an alphabet factor due to the realisation as a linked list per se. Nevertheless Kurtz states that it would result in a performance penalty in practice if implemented as is. Thus a small trick is used: it is obvious that nodes close to the root have in the average case more children than nodes deep down the tree. Moreover these nodes are the ones that are most frequently traversed when searching in the constructed suffix tree later on, which means that a significant speed up is gained if these nodes store their suffix links themselves. As the nodes close to the root are also the ones which are less deep, these require less bits which can again be used for the suffix link.

To sum up this leads to the following layout of integers 3 and 4 in a big node: the first bit (marked with ↓) is set if the depth of the node is less than 1024 (or $2^{10}$ and thus can be
represented by the last 10 bits of integer 3. In this case 21 bits of the suffix link are stored in the remaining bits of integer 3, the remaining 5 are stored in integer 4, which also always contains the node’s head position in its last 27 bits. If the node has a depth of at least 1024 it is stored in the least significant 27 bits of integer 3 with the resulting rest remaining unoccupied.

The final question that remains unanswered is the upper limit of the length of the input sequence with this node layout and we will see that it is $2^{27} - 1 = 134,217,727$ characters.

The branchbrother and first child references each have 28 bits at their disposal (one of the 29 bits indicates the reference target type (leaf or branching node)). Each node has a maximum size of 4 integers which means that the maximum size of the node table is $2^{29} - 4$ and, as the node size is always a multitude of 2, $2^{28}$ bits suffice to address every possible node position in the table. The leaf table has the size of the length of the input sequence, so $2^{27} < 2^{28}$ bits would suffice. As the depth and the headposition of a node are also strictly limited by the maximum size of the input the 27 bits for each of these is enough to finally prove that the limit for the indexed text is indeed $2^{27} - 1$.

### 2.1.5. The construction algorithm

Kurtz proposes to use a modified version of McCreight’s construction algorithm. We will first sketch the essential ideas of this construction algorithm without going into detail, however, as Gusfield provides a terrific insight to the general idea’s in [4] where he devotes some chapters to suffix trees and their use. The general idea is to insert the suffixes from long to short beginning with the whole text as the first inserted leaf, continuing with the suffix at start position 1 and so on, see figure 2.2. To get away from quadratic runtime the search for the next insert position has be optimized away from scanning down from the root each time a new suffix is inserted which is done by using suffix links on the one hand and some other observed regularities to avoid this, as will be seen in the resulting construction algorithm. What is of great importance beforehand is that McCreight’s algorithm guarantees node insertion in the order of their head position’s and therefore only Ukkonen’s or McCreight’s algorithm (not the one of Weiner) can be used as the basis of Kurtz proposed construction idea. McCreight’s gets the upper hand due to a slight practical performance advantage over Ukkonen’s algorithm.

Some additional terminology needs to be given to be able to comprehend the construction of this tree in III. Each suffix $S_i$ that is newly inserted into the tree can be divided into two parts, the head and the tail, where the concatenation $\text{head}_i \text{tail}_i$ is the whole suffix and $\text{head}_i \in ST_{i-1}$ is the largest prefix of $S_i$ still in the suffix tree constructed in the last step. For the construction we need two pointers into the tree, headloc and tailptr. $\text{headloc}_i = [\text{head}_i]_{ST_{i-1}}$, i.e. it represents the (possibly not yet existing) node for $\text{head}_i$ in the suffix tree constructed in step $i - 1$. $\text{tailptr}_i$ points to the first character of $\text{tail}_i$.

The details on the construction itself can be found in the implementation chapter 3.2 or if a more comprehensive algorithm description is wanted, it is found at [7].

### 2.1.6. Possible extensions and limitations

The implementation of Kurtz’s algorithm can easily be adopted to using hash tables instead of linked lists to represent the tree. The choice is non trivial, as each implementation
has its own drawbacks and advantages over the other, which is explained in greater detail in the original paper. I chose to implement the linked list version. Greater than the limitations of hash tables or linked lists is, however, the overall limit of the input sequence. The limit can be lifted significantly on a 64 bit system with larger address space and by using 64 bit integers to be able to address more leaf and node positions in references. Nonetheless the whole input sequence as well as the tree have to fit in the main memory as the algorithm assumes (and heavily uses) constant time random access to tree nodes as well as input sequence characters. As soon as the tree does not fit in main memory the algorithm dramatically slows down because random disk access is significantly slower than local (sequential) disk access would be.

In the next section we will see an algorithm that tries to get away from this presumption and follows a disk aware approach in order to be able to index very large sequences.

2.2. STTD64

The STTD64 (Suffix Tree Top Down 64bitx) suffix tree construction algorithm was proposed in [5]. It is designed for sequences of a length up to \(2^{32} - 1\) and is suitable for building suffix trees that do not fit in main memory. It is mainly based on the ideas in [2] and [11] trying to overcome the greatest limitations of these two algorithms. In the next part of this work the ideas of the STTD64 index will be outlined. Details in on its implementation in SeqAn are found at 3.3.

2.2.1. How to store the nodes?

The STTD64 suffix tree is fundamentally dissimilar from Kurtz’s tree presented in the above section. It uses a completely different node layout as well as construction idea. As it is important to be able to understand the algorithm let us first take a look at the node layout. It is based on the node layout proposed by Giegerich et al. in [2]. Giegerich proposed to store the suffix tree information by first defining the leaf set of a node \(w\) as the set \(l(w) = \{x | x\) is the start position of any leaf that is a child of \(w\}\). The leaf set of a leaf \(l_0\) representing the suffix \(S_j\) is simply \(l(l_0) = \{j\}\). Using this leaf set definition an ordering on child nodes is defined as follows: For two different children \(w\) and \(u\) of the same node \(x\):

\[ w \prec u \iff \min(l(w)) < \min(l(u)) \]

As the leaf sets of two different children of the same node are distinct this ordering is total. Prosaically this ordering could also be read as “one node \(u\) is smaller than another node \(w\) if the starting position of the leftmost suffix it represents is smaller than the leftmost suffix represented by the other node”. Now a node and its incoming edge \(U \xrightarrow{V} UV\) can be stored as follows:

- Define \(lp(UV) = \min(l(UV)) + |U|\) as the left-pointer of the node \(UV\) (convince yourself that this means that \(lp(UV)\) is the text position of the first character of the edge incoming to \(UV\)). In Giegerich’s terminology concerning this tree data structure \(|U|\) (i.e. the sum over all edge lengths leading from the root to the parent of \(UV\)) is called the depth of the child \(UV\).

- Assume that \(UVW\) is the smallest child of \(UV\). This means that \(\min(l(UVW)) = \min(l(UVW)) + |UV| = \min(l(UV)) + |U| + |V| =\)
\[ lp(\overline{UV}) + |V|. \] Now let \( i = lp(\overline{UV}) \) and \( r = lp(\overline{UVW}) \) then \( V = x_i \ldots x_{i+|v|-1} = x_i \ldots x_{lp(\overline{UV})+|v|-1} \).

What that means is in order to determine the edge incoming to a node in constant time we only need the left-pointer of the node as well as the left-pointer of its smallest child. Consequently the only things stored for a node are its left-pointer and a first-child pointer to its smallest child. As the label incoming to a leaf is strictly defined by its start position alone we only store the left-pointer of a leaf. Additionally we need one bit to distinguish between a node and a leaf and one bit to mark the right most child of a node. To sum up we need two integers for each node and one integer to store a leaf.

The advantage of this representation regarding storage space over the layout proposed by Kurtz is obvious: In this representation the worst case storage requirement is \((2 \times 4n + 4n)\) byte = 12n byte over a worst case storage requirement of 24n byte (or 17.2 bytes in the unproven practical worst case) for Kurtz’s representation. As we will see later on this comes at the cost of an increased construction time memory requirement leading to a maximum overall space requirement of 22n bytes during construction in the worst case [2].

One limitation of the above node representation is, however, that only 30 bits are available to store the lp-value (in a leaf at least) and, as the lp-value needs to represent values from 0 up the text length \( n \), this limits the input length to \( 2^{30} - 1 \). The second disadvantage of the above representation to Kurtz’s representation concerns searching in the tree: Note that the depth of a node is not explicitly stored in each node but can only be retrieved by descending into the tree from the root to a node/leaf. This is problematic if we consider for example exact-match-queries (which are very common and build the basis for various more complex search routines): Say we search for a pattern \( P \) and this search ends at an inner node \( u \): then the occurrences of the pattern in the string are represented by the leaf-set of \( u \), i.e. all of its child leaves. As the depths are not stored in the leaves/their father nodes we need to explicitly iterate over all children of \( u \) to get every occurrence of the \( P \) in the text, which has been shown to have bad locality properties [6].

Halachev tries to tackle both of these drawbacks with a change to the node table. He proposes to store two integers for the leaves as well as for the nodes. The first integer for nodes stores their left-pointer and the second integer the first-child pointer as well as the leaf- and the right-most flag. The lp-value of leaves also goes into their first integer and the two flags into their second. The most important difference to Giegerich’s layout is, however, that the depth of a leaf is stored in its second integer for more efficient depth-first-search (see chapter IV) at the cost of an increased tree size of 4n additional bytes for the second node integer (leading to 16n bytes in total) in the worst case.

The resulting layout is depicted in 2.3. The STTD64 node table for our example suffix tree can be seen in image 2.4 [5]. Note that the maximum input sequence length storable in this layout is \( 2^{32} - 1 = 4GB \) assuming that there are no repetitions of length \( 2^{30} \) or greater as there are only 30 bits available for the depth value.

Now that the general storage layout of the STTD64 suffix tree is at hand the next section will shed some light on the algorithm in use for construction.
2. Indices

2.2.2. The Tree Top Down Construction algorithm

You may have noticed in the above section that no suffix links whatsoever are stored in the node tables. As construction relying on suffix links is inherently bad in regards of disk (or RAM) access locality an algorithm disk based construction must avoid using them. And even though they have some practical use in searching this use is nearly nullified if the tree is stored on disk as the random node access leads to many non sequential accesses. Consequently as they are of no use during construction time and of vastly reduced use for searching they are abandoned for the benefit of reduced storage space. How, then, is the tree constructed without the use of suffix links? The STTD64 construction algorithm uses, as already stated, a combination of the ideas in [2] and [11], where the first of the two proposed the tree construction algorithm and second one proposes a partitioning and buffer management technique for large trees. We first take a look at the construction algorithm and afterwards show how the task can be efficiently split for vast trees.

The write only top down algorithm (wotd)

The general idea of this algorithm is to first sort all suffices a node represents (for the root this would be the set of all suffices) into a suffix array. Note that, of course, only their start positions are stored as that suffices to describe a suffix accurately and that in the following suffix is a synonym for its start position. Then the longest common prefix \( lcp \) of all these suffices is determined. Now assume the \( lcp \) is 2 then all suffices are sorted into one of \(|\Sigma| + 1 \) buckets according to their third character (i.e. is the first one where some mismatch occurs).

The sorting can be done with a linear algorithm, called count-sort. As we have a fixed alphabet size which is several orders of magnitude smaller than the input size a double pass search is used. In the first step all suffices are iterated over and the size of each bucket is calculated by simply summing up the occurrences of the corresponding letter and at the same time copying all suffices to a temporary array (in this step the \( lcp \) is added to the begin position of each suffix which henceforth represents only the part relevant for future processing). Now that we know each bucket’s size we can simply go through the temp array from start to end and sort each suffix to its final and sorted position. This kind of sorting has the additional advantage of preserving the ordering of the suffices from long to short (i.e. from low start position to high start position) for each bucket, which is essential for the whole construction algorithm.

After sorting each of the buckets now represents one of the node’s outgoing edges (i.e. one of its child nodes) while the length of the node’s incoming edge is fix to be the same as \( lcp \). Repeating these steps recursively for every child node supplies the whole tree.

Recursion, however, would not be very efficient, so a different approach is used and it works as follows: We first need a stack-like data structure like it is provided for example in the std template library of C++ or like the one SeqAn offers. After the suffices are sorted into their respective buckets one iterates over the buckets and has three choices depending on the number of suffices contained in a bucket \( X \), where bucket \( X \) is the bucket containing all suffices that had an ‘X’ at position \( lcp + 1 \).

- If bucket \( X \) contains no suffices, nothing has to be done as there is no edge outgoing from the current node that starts with ‘X’.
2. Indices

- If bucket $X$ contains one suffix $x$ the bucket represents a leaf which is immediately appended to the suffix tree in its final state. Note that the $lp$-value of the leaf is simply the start position of the $x$ as the correct ordering is always guaranteed by count-sort.

- If bucket $X$ contains two or more suffixes it represents a node. In order to be able to evaluate this node as soon as all current buckets are finished, and for the node to be at the right place in the suffix tree a placeholder is appended to the tree: The left pointer of the node (i.e. its first integer) stores the position of the first suffix in the bucket and the second integer stores the position of the last suffix. After adding this marker node to the tree its position is pushed onto the stack for later evaluation.

At the end of this process the current node is completely evaluated and will never be needed again during construction time which is in sharp contrast to suffix link based construction where nodes all over the tree are accessed at any stage. Now that this node was taken care of the upper most node is popped from the stack and the procedure repeats until the stack is empty and the tree stands proudly (or rather hangs top down from the imagined ceiling).

Figure 2.5 illustrates the first few steps of the wotd-algorithm.

Regarding running time this algorithm is quadratic in the worst case. You can easily conceive this if you think of a worst case string like ‘AAAAAAAS’$. Then the $lcp$ will be 1 x for every node evaluation providing one leaf and one node that is evaluated next leading to a number of count sorts in the order of $\Theta(\frac{n^2}{2})$. In practice, however, the running time is of lower order and the algorithm is well usable, as will be shown in the evaluation chapter.

So far so good - we have got an algorithm that builds trees without using suffix links. In the next section this algorithm will be embedded in a framework that allows it to be used for constructing large trees that do not entirely fit in main memory.

2.2.3. Partitioning and buffers

From the above explanation of the wotd-algorithm we noticed that there are four kinds of data structures needed. Each of these data structures is accessed in a different manner during construction which is important to consider when thinking about buffer sizes and swapping strategies in case of low memory.

- The string of which we want to construct the suffix tree is the first data structure, henceforth simply called String in accordance to the nomenclature in [11]. String has the worst access pattern during construction time, namely: random.

- The data structure for all suffixes of the tree (Suffices). During construction time, e.g. in count-sort it is accessed first sequentially when its containments are copied into TEMP. The second phase of count-sort does not access Suffices completely random, either, but has some locality as there are $|\Sigma|$ different locations where suffixes are then sequentially added.

- A data structure where the suffixes are temporarily copied to during count-sort (TEMP). TEMP is accessed sequentially in both phases of count-sort.
2. Indices

- And last but not least the tree data structure itself wants to be stored somewhere (Tree). The tree data structure writes are all sequential with some relatively rare reads to nodes that are popped from the stack.

Considering the different read and write patterns [11, p.285-287] proposed the following strategy to distribute memory pages to the buffers: the minimum number of pages for the Tree buffer is two, to provide for a scenario at the borders of a page, when after writing a node to the beginning of a new page its father, which resides yet on the old page, is popped from the stack. With two or more pages available this can be done without leading to two page faults back-to-back. For both TEMP and Suffices a minimum of \( |\Sigma| \) pages is reserved, as buffer sizes below this lead to a disproportionally high rate of pager faults. Now the remaining pages are distributed as follows: As the string is accessed randomly and has the smallest size of all these data structures as many pages as necessary for the string are assigned to String. The remaining pages are assigned in this order to Suffices up to the size it requires, then to TEMP, and if any are left these are given to Tree.

Tian et al. suppose the following strategies for swapping when it becomes necessary. They compared took into account the LRU(least-recently-used), the MRU(most-recently-used), the CLOCK and the RANDOM swapping strategy, as is explained in detail at the above location in their paper. For String, Tree as well as Suffices they propose to use the LRU swapping strategy and only for TEMP they found MRU to be the most efficient (it was by orders of magnitude the worst for the other buffers but seems to be by a factor 2-3 better for TEMP due to the strictly sequential access pattern).

Using these handmade buffers alone would already improve the construction time for large text size but this would still lead to a lot of page misses if only the minimum number of pages is available to TEMP and Suffices. To be able to construct suffix trees of size in the order of the entire human genome (3GB) this would still lead to an unbearable construction time. So there is one additional step prepended: The set of suffices is partitioned into \( |\Sigma|^{\text{prefixlen}} \) partitions. This means that we take a prefix of a length between 0 and 7 dependent of the size of the input string. Before we go over to constructing the tree manner described above we search in one pass through the set of all suffices and for each starting position \( x_i \) we look at the next \( \text{prefixlen} - 1 \) characters to determine the partition for this suffix. Let us look at an example to clarify what this means. Lets say we take a \( \text{prefixlen} \) of 2 for an input sequence of DNA type (\( |\Sigma| = 4 \)) then we get \( 4^2 \) partitions, namely: ‘AA’, ‘AC’, ‘AG’, ‘AT’, ‘CA’, ‘CC’, ‘CG’, ‘CT’, ‘GA’, ‘GC’, ‘GG’, ‘GT’, ‘TA’, ‘TC’, ‘TG’, ‘TT’. Now all suffices that start with the two ‘A’s would go into the first partition, those starting with ‘AC’ would go into the second and so on.

Using this partitioning strategy we can then construct the suffix tree for each partition independently of the other partitions, thus greatly reducing memory requirements. If we take, for example, an input string of size 500MB (524,288,000 characters) we would need about 4 times this space for each Suffices and TEMP (as each integer storing a start position needs 4 Bytes), 500MB for the string itself and up to 8 GB for the suffix tree leading to a space requirement of at least 4.5 GB + 8 GB. If we partition the data set using a prefix length of 2 each of these data structures (except for the string which should reside in memory completely, if anyhow possible) only needs about \( 1/16 \) of this size leaving a requirement of about \( 780MB + 500 MB \) for each partition. This easily fits into the RAM of modern machines and allows for complete in-memory construction for these medium sized strings.
2. Indices

For the STTD64 we use a third step (after partitioning and constructing) which is filling in the depths for all leaf nodes by traversing completely traversing the tree. This step takes $O(n)$ additional time but for lots of search queries (which is the overall purpose for constructing suffix trees in the first place) this pays off, as will be shown later in chapter three.
2. Indices

Figure 2.2.: The construction steps in the McCreight algorithm

Figure 2.3.: The STTD64 bit layout for a) nodes and b) leaves
Figure 2.4.: The STTD64 node table for the input string ‘AGAGAGCTT$’

Figure 2.5.: The node evaluation process. Red parts are completely evaluated.
Part III.

Implementation
3. Implementation

3.1. The SeqAn library

This section gives a short overview of the SeqAn C++ full text search library and its specialities.

The SeqAn project (SeqAn is short for Sequence Analysis) devotes itself - the name already suggests it - to implementing a rich tool set for fast and easy sequence analysis. The main work of the project is the SeqAn sequence analysis library, a C++ template library that features many classes providing the necessary functions to manipulate and search large sequences. This section will briefly outline its specialities and the main programming paradigms for contributions to the library.

3.1.1. Template subclassing

The SeqAn library deviates from the usual way object orientated programming works, especially concerning the way class sub-typing and inheritance are implemented. Let us look at a small example, first on how one would normally realize inheritance in C++ and afterwards on what it looks like in the SeqAn library.

```cpp
class Vehicle {
    public:
        public virtual moveForward() {
            cout << "The vehicle moves forward in an abstract way!" << endl;
        }
}

class Car : public Vehicle {
    public:
        public virtual moveForward() {
            cout << "The car drives forward." << endl;
        }
}

class Plane : public Vehicle {
    public:
        public virtual moveForward() {
            cout << "The plane flies forward." << endl;
        }
}
```
Here the virtual method resolving mechanism guarantees that when moveForward() is called on a Vehicle object, the right sub class method is called instead of the method of the Vehicle super class. This approach has, however, a slight performance penalty ensued as the lookup has to be done dynamically at runtime. Normally, this performance penalty is far outweighed by the structural cleanliness of the inheritance approach and of the fully object oriented concept. As with the SeqAn project, however, the problem is that it is intended for very calculation intensive applications, where slight performance losses quickly sum up and eventually drastically slow down the whole program. Thus a different approach for modelling virtual method lookup and inheritance is chosen: template based subclassing. The following listing shows the vehicle example in the template based approach.

```cpp
template <typename TSpec>
class Vehicle<TSpec> {
};

struct Car;

template <typename TSpec>
class Vehicle<Car<TSpec>> {
};

struct Plane;

template <typename TSpec>
class Vehicle<Plane<TSpec>> {
};

template <typename TSpec>
void moveForward(Vehicle<TSpec> & vh) {
    cout << "The vehicle moves forward in an abstract way!" << endl;
}

template <typename TSpec>

./listings/example_inheritance.cpp
```
3. Implementation

```cpp
void moveForward (Vehicle<Car<TSpec>>& vh) {
    cout << "The \_car\_drives\_forward.\" << endl;
}

template <typename TSpec>
void moveForward (Vehicle<Plane<TSpec>>& vh) {
    cout << "The \_plane\_flies\_forward.\" << endl;
}
```

With this approach the specialization happens within the template argument TSpec and could be arbitrarily deep. One could imagine a type Vehicle<Car<Roadster<TSpec>>>& for example. The advantage of this approach is, that template binding happens at compile time and thus no lookups have to be done at runtime, leading to less overhead. Nonetheless one can give special method “overwrites” whenever necessary and sensible, just like with regular C++ inheritance. Note that the methods working on the Vehicle objects consequently are not member functions of the Vehicle (sub)class. It is thus a logical convention that the classes do not have any private member variables, but that instead all their data members are public.

3.1.2. SeqAn classes correlated to suffix tree implementation

All indexing structures, and suffix trees are of this category, are sub classes of the Index class with classes for the enhanced suffix array (ESA) or the Wotd(lazy) already implemented, for example. Moreover, to be able to use the suffix tree for e.g. pattern search, there needs to be some type of iterator given for the tree.

Let us look at a standard approach for exact match searching using the Index structure and top-down-iterators taken from here [10] and minimally adopted:

```cpp
int main ()
{
    typedef Index<CharString, IndexEsa<> > TIndex;
    TIndex index("How\_many\_wood\_would\_a\_woodchuck\_chuck.\");
    Iterator<TIndex, TopDown<> >::Type it (index);

    CharString pattern = "wood";
    while (repLength(it) < length(pattern))
    {
        // go down edge starting with the next pattern character
        if (!goDown(it, pattern[repLength(it)])) return 0;
        unsigned endPos = min(repLength(it), length(pattern));
        // compare remaining edge characters with pattern
        std::cout << representative(it) << std::endl;
        if (infix(representative(it), parentRepLength(it) + 1, endPos) !=
```
3. Implementation

At line 3 the type of the index is defined and we see that an Index takes two template arguments. The first one is the type of the sequence it will hold like String<CharString> which is equal to CharString or String<Dna> for holding DNA sequences. The second template argument defines the index specialization (i.e. the index type). In the example this would be IndexEsa<>. In line 4 the index object is created and the string to be indexed is passed as an argument to the constructor.

An iterator over the index is defined in line 5. The Iterator type is retrieved by a metafunction (a special structure definition that defines the type of the iterator as its member Type). The metafunction approach is used so that an iterator (or other useful operators) can at any time be added to an arbitrary data structure without modifying the code of the data structure that is iterated over (C++ template metafunction programming is by the way turing complete but that is not the point here). The Iterator template arguments are the type of its index and its specialization which is simply TopDown<> for top-down-only iterators, which suffice for exact match pattern searching.

In the while-loop we try to goDown the tree while matching the pattern defined in line 8. The repLength() returns the length of the substring of the node the iterator represents and lstinlinerepresentative() returns this substring. The infix (string, start, end) function returns an infix of its first argument, starting at the position of its second argument and going upto the last character before its third position (i.e. end − start is equal to the length of the infix).

If any occurrences of the pattern were found their start positions in the string are printed in the final for-loop (getOccurrences() returns a list of all these).

The next brief section will describe other useful SeqAn constructs needed during the implementation.

3.1.3. Useful data structures

One of the most important data structures is the String<TValue, TSpec> container. This is the general purpose data structure for storing string like types and maybe comparable to std::vector. The first of its template arguments specifies the value type it holds (char, int, Dna...). The second argument type is the specialization type where the most important ones are
3. Implementation

• Alloc<> This is the standard specialization: An in memory string which can grow
  and shrink at runtime.

• Block<BLOCKSIZE>: A string optimized for stack behaviour (push on top, pull the
  top), where BLOCKSIZE is the size of memory which is reserved in one block if the
  stack outgrows its boundaries

• MMap<External<FILE_ACCESS, PAGE_SIZE, FR_NO>>: A memory mapped string
  which uses the OS functionality (mmap() on linux systems) to map a file into mem-
  ory. According to its documentation the first parameter defines the file access man-
  ner, where the default is Async<> supporting synchronous and asynchronous ac-
  cessing. The second argument configures the page size and the third the number
  of frames. It seems, though, that only the first two parameters have any effect and
  that PAGE_SIZE is the overall size of the mapping. This String specialization is very
  efficient if one wants to read a huge file into memory.

• External<External<FILE_ACCESS, PAGE_SIZE, FR_NO>>: An external string, where
  the three arguments are the same as above but the two last ones are fully taken into
  account. This string implements an internal swapping mechanism and PAGE_SIZE
  configures the size of one string-internal memory page. FR_NO defines how many
  of the Strings pages are kept in memory before some are swapped to the file that is
  linked to the container. This string seems to come close enough to the buffers men-
  tioned in section 2.2 as they use an LRU strategy for swapping.

Every string provides an Iterator<STRING_TYPE> to iterate over its content as well
as functions to append, get and set a particular value at a random position. There also
are infix, prefix and suffix functions for retrieving these special substrings in the manner
described in the example above.

Now that our tools are at hand we look at the implementation details of Kurtz’s and Ha-
lachev’s algorithms in the upcoming two sections, where the focus will be on the STTD64
tree algorithm. The full source code is on the compact disc attached to this work.

3.2. The implementation of the space reduced suffix tree

The following section outlines the implementation of the suffix tree construction algorithm
proposed in [7] and already presented in section 2.1.

3.2.1. The Index class requirements

In order to store the tree data structure for the linked list implementation of the space
reduced suffix tree two data structures are required, one for the node table and one for the
leaf table. For both of these a String<unsigned> seems feasible. As Kurtz already gives
a good pseudo-code algorithm for the tree construction [7] I will just quote it here 3.1 and
after that show what the corresponding code looks like.

The resulting code in C++ looks like follows.

```
1 template<typename TText, typename TConfig, typename TSpec>
```
inline void _buildIndexSTKurtz(Index<TText, IndexSTKurtz<TConfig, TSpec>> & index)
{
    SEQAN_CHECKPOINT;
    typedef typename TConfig::NODE_TYPE TNodes;
    typedef typename TConfig::LEAF_TYPE TLeaves;
    typedef STKurtzBitMasks BM;

    SEQA_ASSERT_EQ(length(index.nodeTable), 4u);  // the root is there
    STKurtzHeadLoc<TText, TNodes, TLeaves> headloc;
    headloc.fatherPos = 0;
    headloc.rest = infix(index.text, 0, 0);
    STKurtzTailPtr<TText> tailptr;
    STKurtzTarget<TNodes, TLeaves> tempNode;
    typename Iterator<TNodes, Rooted>::Type nodeIt = begin(index.nodeTable);
    tailptr.tail = suffix(index.text, 0);
    unsigned baseAddressQ = 4;  // node insertion address
    unsigned chainLengthP = 0;
    _insertLeaf(index, headloc, tailptr);

    appendValue(index.nodeTable, 0);  // always two trailing node ints necessary
    appendValue(index.nodeTable, 0);
    unsigned i = 0;
    unsigned lastNodeInsertionAddress = 0;

    // tail begin is at most the position of the last real character (the one before e.g. $)
    while (beginPosition(tailptr.tail) <= length(index.text) - 2
            // still characters || headloc != root
            || !(headloc.fatherPos == 0 && length(headloc.rest) == 0))
    {
        if (headloc.fatherPos == 0 && length(headloc.rest) == 0)
            // root
            setBeginPosition(tailptr.tail, beginPosition(tailptr.tail) + 1);
            _scanPrefix(index, headloc, tailptr,
                        lastNodeInsertionAddress);
        else
            {
                if (length(headloc.rest) == 0)
                    {  // is a node with suffix link already complete
3. Implementation

```c
unsigned slink = _getOwnSlink(index.nodeTable, index.leafTable, headloc.fatherPos);
tempNode = _followReference(index.nodeTable, index.leafTable, slink);
SEQAN_ASSERT(_nodeValid(tempNode));
headloc.fatherPos = tempNode.posNode;
_scanPrefix(index, headloc, tailptr, lastNodeInsertionAddress);
}
else
{
    if (headloc.fatherPos == 0)
    { //the root is the father (intended use of no && length(...) addition!!
        setBeginPosition(headloc.rest, beginPosition(
            headloc.rest) + 1); //chop of the first char
        _rescan(index, headloc,
            lastNodeInsertionAddress);
    }
    else
    {
        unsigned slink = _getOwnSlink(index.nodeTable, index.leafTable, headloc.fatherPos);
tempNode = _followReference(index.nodeTable, index.leafTable, slink);
SEQAN_ASSERT(_nodeValid(tempNode));
headloc.fatherPos = tempNode.posNode;
_rescan(index, headloc,
    lastNodeInsertionAddress);
    }
    chainLengthP++;
    if (length(headloc.rest) == 0 || chainLengthP == 31)
    {
        //is a node
        //set own distance to zero
        setPosition(nodeIt, baseAddressQ);
        unsigned int1 = *nodeIt;
        int1 &= (~BM::AllBitsSet - BM::DistanceInt1);
        assignValue(nodeIt, int1);
        _setOwnSlink(index.nodeTable, index.leafTable, baseAddressQ, headloc.fatherPos);
        for (unsigned d = 1; d < chainLengthP; d++)
        { //set distances
...}
```
3. Implementation

```c
setPosition(nodeIt, baseAddressQ - 2 * d);

unsigned int1 = *nodeIt;
int1 &= (BM::AllBitsSet - BM::DistanceInt1);
int1 += (d << BM::ShiftDistanceBitsInt1);
assignValue(nodeIt, int1);
}

chainLengthP = 0;
baseAddressQ += 4;
appendValue(index.nodeTable, 0); // always two trailing node ints necessary
appendValue(index.nodeTable, 0);
_scanPrefix(index, headloc, tailptr,
         lastNodeInsertionAddress); // seems to change little

} else
{
    // SEQAN_ASSERT_EQ(_getHeadPosition(index.
    nodeTable, baseAddressQ - 2, BM::NoPos), i)
    baseAddressQ += 2;
}

if (length(headloc.rest) == 0)
{   // is a node
    _insertLeaf(index, headloc, tailptr,
            lastNodeInsertionAddress);
} else
{
    _insertNode(index, baseAddressQ, headloc, tailptr);
    if (lastNodeInsertionAddress + 2 == baseAddressQ)
    {
        // set maximum distance of last node until real
distance is set later
        setPosition(nodeIt, lastNodeInsertionAddress);
        unsigned int1 = *nodeIt;
        int1 &= (BM::AllBitsSet - BM::DistanceInt1);
        int1 += (31 << BM::ShiftDistanceBitsInt1);
        assignValue(nodeIt, int1);
    }
```

3. Implementation

One additional modification (addition) not mentioned in Kurtz’s paper was necessary in order to make the construction work correctly. It was found necessary that the headposition of a small node is available during construction even at times when the chain this node belongs to is not yet completed. This occurs for example in the familiar tree in figure 1.1 when the node (′G′) is inserted: in order to be able to insert the new node and the corresponding new leaf at the right place, the characters labelling the edge have to be known which is only possible if the depth and headposition of the original child node (′GAG′) is readily available. So in order to accomplish the availability of any head position and depth at any time the following work around was established: whenever a new node is inserted into the table it is initially a big node providing the full 4 integers of information. If it turns out later on that the node was indeed a small node then its last two integers are overwritten by the next inserted node and its distance is temporarily set to the maximum of 31. When the construction algorithm comes across this distance at a later time it is first tested if a node exists at the assumed position and if not a reference to the last inserted node is assumed. This last inserted node’s position has, as a consequence, to be passed to the scanPrefix, rescan and insert functions (directly or indirectly by making assumptions about the position relative to the current length of the node table). The (on the surface) simpler approach of only inserting big nodes in the first place (without immediate over-ride) and then moving them back once the end of the chain is reached is not feasible as this would lead to incorrect branchbrother and firstchild references, and updating those is virtually impossible without increasing the overall complexity (note that these references are not local to the chain of small nodes but also refer to nodes outside the chain, and vice versa the small nodes are referred to from the “outside”).

Before evaluating the algorithm the next section will present a (more detailed) description of the implementation of the second proposed algorithm, the STTD64.

3.3. The implementation of STTD64

This section will give a detailed insight into the implementation process and will show some of the pitfalls one can meet when implementing applications for very huge data size in general and in SeqAn. The first part will explain the outline of the data structure and list the requirements for the index class. The second part will explain how the improvement of the construction time for the index was decreased by 1/16 by fixing some of the major problems of the original implementation.
3. Implementation

3.3.1. The Index class requirements

The index structure needs the following member data structures at least:

- A container for the text the index is created for. In the first implementation I chose to use an instance of `String<External>` for this purpose.

- One data structure where the suffixes go to in the partitioning phase. I put them into a `String of String<Position, External> strings`, one contained string for each partition.

- One data structure where the suffix trees `Suffices` are stored. I chose again a string of external strings but this time the value type is a node (i.e. is a struct of two integers).

During the second phase of the suffix tree construction, namely the construction of the trees themselves there are two temporary required:

- The `TEMP` data structure that is needed during count-sort. As it has the same structure as `Suffices` one external string of the same value type is used

- The stack for memorizing the unevaluated nodes. I used a `String<Block> >` as this seems tailored to the needs of a stack data structure.

There rose the need for one additional data structure required to provide for a consistent Iterator-interface: I called it the Virtual Prefix Tree and it is necessary in order to be able to iterate over all of the trees as if they were just one suffix tree. This Virtual Prefix Tree provides methods for iterating over the very dense virtual suffix tree that would result when building the suffix tree only for the set of prefixes and after that adding the last `prefixlen - 1` suffixes of the string. Note that these suffixes can not be sorted into any of the proper partitions as they are not long enough and consequently would not be found during the iteration process.

Now that the general needs are clarified the next section will give a comprehensive overview of the implementation process.

3.3.2. On optimization - Finding bottle necks

(A warning to the readers that are only in for very dry text: this section contains informality)

There came the day when my first working implementation was ready and tested within test cases of a volume of 1000+ lines and I wanted to feed the algorithm something bigger. I decided to start out with the human Y genome of size 25 MB as it is the smallest human genome and yet is of non-trivial complexity (i.e. length especially). After setting up a small benchmarking program I let the algorithm construct the suffix tree and was quite taken aback by the fact that it took 14.5 minutes on my machine (a 64Bit machine with an `intel dual core (2.1MHz)` processor with about 3 GB memory available to the tree construction algorithm). This sounded like a very long time, and it is, as Halachev states that his implementation took around 72 seconds to build the tree. This meant that my implementation was slower by a factor of approximately 12 which is an order of magnitude
and thus unbearable. Moreover it was expected that this factor increases for larger data structures which would rend the algorithm entirely useless.

So I wondered what is doing the mischief: Is it the SeqAn buffers? This was not very probable as they should be exhaustively tested but considering the fact that the algorithm took only 1 minute when I switched to the slightly newer MMap<> specialization I thought I had spotted the bottle neck.

I was proven partially wrong. After seeing that memory mapped strings are no feasible solution. The running time is super fast for small data sets but as soon as the input size surpasses a certain threshold it is vastly inferior to tailored buffer management. The construction for the largest single human chromosome (the first one) was not possible with memory mapped strings nor with the other configuration in finite time (¡ 6 h). Taking all this into account I decided to go for profiling tool support and oprofile can only be recommended in my opinion, at least it pointed out the first major bottle neck.

oprofile is simply taking snapshots of the system state every x clock cycles and most importantly looks at the current instruction pointer. It thereby counts which instructions are encountered how often when taking snapshots which results in a probabilistic report of which instruction took the most overall calculation time.

Below is an excerpt of the topmost part of the profile.

```
1 CPU: Core 2 , speed 800 MHz (estimated)
2 Counted CPU_CLK_UNHALTED events (Clock cycles when not halted)
   with a unit mask of 0x00 (Unhalted core cycles) count 100000
3 samples % image name app name symbol name
4 4212342 26.0461 index_suffix_trees_benchmark
   index_suffix_trees_benchmark seqan::ExtStringFwdIterator<seqan::String<unsigned int>, seqan::External<seqan::ExternalConfig<seqan::File<seqan::Async<void>>, 1048576u, 256u>>, seqan::ExtStringFwdIterator>()
5 2396358 14.8174 index_suffix_trees_benchmark
   index_suffix_trees_benchmark seqan::String<seqan::SimpleType<unsigned char, seqan::Dna5_>, seqan::External<seqan::ExternalConfig<seqan::File<seqan::Async<void>>, 1048576u, 64u>>, seqan::getPage<int, seqan::PageFrame<seqan::SimpleType<unsigned char, seqan::Dna5_>, seqan::File<seqan::Async<void>>, seqan::Fixed<1048576u>>>, seqan::Priority<seqan::PageFrame<seqan::SimpleType<unsigned char, seqan::Dna5_>, seqan::File<seqan::Async<void>>, seqan::Fixed<1048576u>>>, int>
6 2349886 14.5300 index_suffix_trees_benchmark
   index_suffix_trees_benchmark unsigned int seqan::_longestCommonPrefix<seqan::String<seqan::SimpleType<unsigned char, seqan::Dna5_>, seqan::External<seqan::ExternalConfig<seqan::File<seqan::Async<void>>, 1048576u, 64u>>, seqan::Segment<seqan::String<unsigned int>, seqan::External<seqan::ExternalConfig<seqan::File<seqan::Async<void>>, 1048576u, 256u>>, seqan::InfixSegment> >, seqan::String<seqan::SimpleType
```
It is obvious that the most costly operation is the repeated deconstruction of an ExtStringFwdIterator. These iterators are used when an iterator on an external string is requested. Now look at the code below responsible for computing the longest common prefix of a set of suffixes, and try if you find the bad code line.

```cpp
template<typename TText, typename TSuffixesInfix>
inline unsigned longestCommonPrefix (TText const & text, TSuffixesInfix & nodeSuffixes, unsigned minLCPGuaranteed) {

    typename Iterator<TSuffixesInfix const>::Type suffixesIt;
    typename Position<TText>::Type startOfFirstSuffix;
    typename Position<TText>::Type startOfCompSuffix;

    suffixesIt = begin(nodeSuffixes);
    startOfFirstSuffix = *suffixesIt;

    for (suffixesIt = begin(nodeSuffixes) + 1; suffixesIt != end(nodeSuffixes); ++suffixesIt) {
        startOfCompSuffix = (*suffixesIt) + i;
        SEQAN_ASSERT_LEQ(startOfCompSuffix + i, textLen);
        if (startOfCompSuffix + i == textLen) {
            return lcp;
        }
        startOfFirstSuffix = getValue(text, startOfFirstSuffix + i);
        if (startOfFirstSuffix + i == textLen) {
            return lcp;
        }
        compareChar = getValue(text, startOfCompSuffix);
    }
```
### 3. Implementation

The most critical line in is line 22 when in the for loop the test if the end of the current infix of suffices is already reached. This is done by a call to the iterator’s end-function which is semantically correct but leads to the construction and destruction of this iterator once for every pass through the loop. As the \( lcp \) is calculated for every node this are lots of unnecessary destructors. The destruction is particularly expensive for external-string-iterators as each destruction invalidates the page the iterator points to. So I decided to not only take out the critical statement but also change the manner of passing the suffix range to the functions \( \text{count-sort} \). Instead of passing only an infix of \( \text{Suffices} \) to these functions I now passed two iterators, one pointing to the beginning of the range and one pointing to its end. This way these iterators need to be created and destructed only once during the whole construction phase and only their positions have to be adjusted.

After this change to the code the construction time for the Y chromosome could be reduced to one third of the original construction time, namely to 5.4 minutes.

The second optimization was also concerning this function. In the state as it is printed above it iterates over all suffices, reads its first character and checks if it matches the first character of the first suffix. This is OK, as in [11] it is stated that the average \( lcp \) is only slightly above one and thus reading more than one additional character would be unnecessary overhead as the first pass already finds a mismatch. But I have noticed that at some points during the construction it occurs that the \( lcp \) is far greater than 1, for my experiments with the Y chromosome it was beyond 100 for certain suffix sets. So I implemented a two-phase-lcp method. For the first pass the method described above is used but if no mismatch is found (i.e. \( lcp > 1 \)) a second function is called which starts to block-wise compare: for each suffix in the above loop a block of string characters is read instead of just one and these are compared to a block of the string for the next suffix. I found out that it leads to good results to start of with a block size of 2 and increase the the size by factor 2 for every time the current block size produced no mismatch. This change brought the construction time down by another 15% to 4.6 minutes.

As this was not yet anywhere near the time given by Halachev I was looking for further reasons that could slow down my implementation and found that the DNA data Halachev used was of alphabet size 4, but my DNA data had some gaps in it which were marked with the character ‘N’. Even though I used the fitting value-type of SeqAn (\( \text{Dna5} \) is offered for cases like this one) such blocks of ‘N’ lead to the quadratic worst case runtime for these suffix regions. And as these block had often times a length of about 50,000 this slowed

```cpp
31  if (charFirstSuffix != compareChar)
32  {
33      return lcp;
34  }
35  ++lcp;
36  }
37  SEQAN_ASSERT(false);
38  return std::numeric_limits<int>::max();
```
down the construction if compared to Halachev’s numbers. What I additionally changed in my algorithm is the type of String<> I used for the input text. As is shown in the paper on DiGeST [1] the TDD-algorithm in general does not perform well as soon as the input string no longer fits into main memory and as Halachev states that he used a compressed string representation when constructing the trees for the whole human genome this suggests that the string also fit into memory (2GB in his case) for the numbers he gives. Thus the overhead of buffering the string is unnecessary in the general case and I decided to get rid of it. Furthermore this change can easily be made undone as the string-type is quickly switched and could be made user defined for future versions of the implementation.

Eliminating the invalid areas from the test data files and changing the String<> specialization reduced the construction time by over 50% to only slightly over 2 minutes and now we are close to the desired time.

Only now I realized that when I wanted to construct the suffix trees for 1.5 GB input length that there was something wrong with size of my suffix partitions. They were larger in sum than 4 times the input string size and opening them in a Hex editor I saw that somehow there are 8 bytes stored for each value instead of the expected four. To explain this fact I first want to show you the general way used in SeqAn to get the size-type or in this case the position-type:

```cpp
//template<typename TSeq>
typename Position<TSeq>::Type determineMidPosition(TSeq const & seq) {
    typename Position<TSeq>::Type endPosition = length(seq);
    return endPosition / 2; // return the (left) middle value
}
```

This way our variables always have a type large enough to store the position independently of the type of the sequence that is given as the argument. Unfortunately I was working under a 64bit system and the size of String<> is a 64bit integer which is of course the right size in the general case. As, however, the maximum input text length can be stored in 32 bits this leads to a large overhead.

Changing the type for representing a position to unsigned manually reduced the space of Suffices and TEMP to the expected size and most importantly also reduced construction time to 97 seconds. This value is already comparable to the 72 seconds and the overhead might possibly come from the wrong page-swapping strategy used for TEMP or the overhead induced by the prioritized LRU strategy in use for external strings. This time is good enough, however, to be able to index large input as will be seen in the evaluation chapter 4.

The next chapter will round up this work by presenting some construction times and proposing some future work.
3. Implementation

\[ \text{headloc} := \text{root} \]
\[ \text{tailptr} := 1 \]
\[ q := 0 \quad /* \text{base address} */ \]
\[ p := 0 \quad /* \text{length of chain} */ \]
\[ T := \text{insertleaf}(ST_0, \text{root}, 1) \]
\[ \textbf{while} \quad \text{tailptr} \leq n \quad \text{or} \quad \text{headloc} \neq \text{root} \quad \textbf{do} \]
\[ \quad \textbf{if} \quad \text{headloc} = \text{root} \quad \textbf{then} \quad /* \text{case (1)} */ \]
\[ \quad \quad \text{(headloc, tailptr) := scanprefix}(T, \text{headloc}, \text{tailptr} + 1) \]
\[ \quad \textbf{else} \quad /* \text{case (2)} */ \]
\[ \quad \quad \textbf{if} \quad \text{headloc} \quad \text{is a node in} \quad T \quad \text{then} \]
\[ \quad \quad \quad \text{headloc} := \text{headloc}.\text{suffixlink} \quad /* \text{case (2.1)} */ \]
\[ \quad \quad \quad \quad \text{(headloc, tailptr) := scanprefix}(T, \text{headloc}, \text{tailptr}) \]
\[ \quad \quad \textbf{else} \quad /* \text{case (2.2)} */ \]
\[ \quad \quad \text{let} \ (\overline{v}, av) := \text{headloc} \]
\[ \quad \quad \textbf{if} \quad \overline{v} = \text{root} \quad \textbf{then} \]
\[ \quad \quad \quad \text{headloc} := \text{rescan}(T, \text{root}, \overline{v}) \quad /* \text{case (2.2.1)} */ \]
\[ \quad \quad \textbf{else} \]
\[ \quad \quad \quad \overline{v} := \text{headloc}.\text{suffixlink} \quad /* \text{case (2.2.2)} */ \]
\[ \quad \quad \quad \text{headloc} := \text{rescan}(T, \overline{v}, av) \]
\[ \quad \quad \quad p := p + 1 \quad /* \text{chain is extended} */ \]
\[ \quad \quad \textbf{if} \quad \text{headloc} \quad \text{is a node in} \quad T \quad \text{then} \]
\[ \quad \quad \quad q.\text{suffixlink} := \text{base address of} \quad \text{headloc} \quad /* \text{case (2.2.3)} */ \]
\[ \quad \quad \quad \textbf{for} \ d := 1 \ \text{to} \ p - 1 \ \text{do} \]
\[ \quad \quad \quad \quad (q - 2d).\text{distance} := d \]
\[ \quad \quad \quad p := 0 \]
\[ \quad \quad \quad q := q + 4 \]
\[ \quad \quad \quad \quad \text{(headloc, tailptr) := scanprefix}(T, \text{headloc}, \text{tailptr}) \]
\[ \quad \quad \textbf{else} \quad /* \text{case (2.2.4)} */ \]
\[ \quad \quad \textbf{if} \quad \text{headloc} \quad \text{is a node in} \quad T \quad \text{then} \]
\[ \quad \quad \quad T := \text{insertleaf}(T, \text{headloc}, \text{tailptr}) \quad /* \text{case (a)} */ \]
\[ \quad \quad \textbf{else} \]
\[ \quad \quad \quad T := \text{insertbranch}(T, q, \text{headloc}, \text{tailptr}) \quad /* \text{case (b)} */ \]
\[ \quad \quad \text{/*} \quad q \quad \text{is the base address of the branching node to be created} \quad */ \]

Figure 3.1.: The pseudo code for the modified McCreight algorithm
Part IV.

Evaluation and future work
4. Evaluation and future work

4.1. Setup

I ran all the following tests on my laptop with an Intel core 2 duo processor, 4 GB of RAM and under a Linux environment (a 64bit system). The tests were compiled using g++ with a cmake generated configuration as proposed on the SeqAn project page. The ‘CMAKE_BUILD_TYPE’ option was set to ‘Release’.

4.2. Comparison

4.2.1. Comparing my implementation of STTD64 to the reference implementation

The first table compares the construction times for the first 8 human chromosomes and chromosome Y to the ones given by Halachev for his implementation.

<table>
<thead>
<tr>
<th>Chromosome Nr</th>
<th>STTD64 Seqan implementation</th>
<th>reference implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Part(s)</td>
<td>Trees(s)</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>955</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
<td>1006</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>711</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>975</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>741</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>1144</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>1168</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>755</td>
</tr>
<tr>
<td>Y</td>
<td>2</td>
<td>89</td>
</tr>
</tbody>
</table>

As we can see my implementation of the algorithm is between factor 1.2 to 2.3 slower than the reference implementation. This is still in order, though, as I used the same configuration for all chromosomes (except for Y) which is: 1MB Page size, 300 MB for each Suffices and TEMP and 1GB for Tree. I furthermore used a prefix-length of 1 for all of these, which would have been unnecessary (this results from the increased storage requirement for the original 64-bit-integer configuration). In consequence this leads generally to worse performance compared to the reference implementation the smaller the input data gets, as the more overhead the unloading and reloading of the partitions that would otherwise be one structure in memory, is.
4. Evaluation and future work

4.2.2. Constructing the suffix tree for larger data size

As the construction time was in the same order of magnitude for my implementation compared to Halachev’s, I wanted to test this for larger data structures and built the tree for the concatenation of the first 8 chromosomes. The overall size of the input data then sums up to about 1.5 GB. Once again the page size was 1MB. The buffer sizes were configured as follows: 600 MB for each Suffixes and TEMP, and 256 MB for the Tree. I was able to build this tree in 193 minutes which is only slightly more than 3 hours. The overall tree size was 19.2 GB which averages to 12.8 byte per input character (exactly the same average Halachev reports for the entire human genome).

Taking into account that Halachev reports a construction time of 16 hours for the entire human genome in a compressed representation (leading to around 1.5 GB again) on a system with less memory available (2.0GB RAM only) I assume that my implementation would easily be able to build the tree in the same timespan or probably even faster as more memory is available. This assumption is based on the fact that even for a 1.5 GB compressed string the memory requirements would not change and only a constant factor would be added to the construction time for the calculation overhead of decompressing the 4 Bit DNA values to regular characters.

4.2.3. The construction time for the Kurtz index

Unfortunately there was a very late found bug in the implementation - and as for me the external index had absolute priority, I implemented that one to work very well. The Kurtz-index will be fixed during the next weeks and should soon be available to SeqAn users.

4.2.4. What still remains undone

My STTD64 implementation is feasible for constructing suffix trees of a size of at least 3 GB in only a few hours which, taking into account that this is not possible at all with memory based indexing structures (at least not in finite time) and SeqAn mainly provided these upto now, is quite notable. There still remains some work to do though, which I was not possible of completing in the limited I had for writing this bachelor’s thesis.

First of all SeqAn requires indices to provide the finder interface (you give a pattern to the finder and it tries to find it in the index, using a requested strategy if available) and also more types of iterators (at least an iterator that can go back up in the tree, which would have specialization type TopDown<ParentLinks<>>). Moreover if given a set of strings the generalized suffix tree for all these strings together should be constructed which is not yet implemented. Furthermore the index at its current state requires the user to configure the buffers’ sizes when initializing the index type through template arguments. This could be done automatically using template meta-programming when only given the total size of memory that should be taken as the algorithm is quite simple. One last point is, that as soon as the tree is built and the leaf depths are to be filled in the TEMP and Suffixes buffers are not needed any longer and the additional memory could be assigned to the Tree buffer for filling in the leaf depths and for searching.
4. Evaluation and future work

4.2.5. A remark on the leaf depth filling

As already mentioned in the other chapters the filling in of the leaf depth improves search performance (and it has to, otherwise about 20% of the construction time could be saved). Here is why: As soon as an iterator points at a node representing the real pattern and requests to output all occurrences, without the depths the iterator would really have to traverse the tree, leading to many jumps in the tree and thus disk accesses. This way one can easily go to the iterators first child and then go right in the node the node table until more rightmost values and than nodes have been passed and output the depth of any leaf encountered. This works and is continuous access to the tree without any jumps necessary.

4.2.6. Conclusion

In the scope of this thesis I have implemented two indexing structures and shown that the implementation of the disk based STTD64 index is able to handle sequences up to the size of the human genome. One of the most promising current suffix tree structures, DiGeST with the $BS^2T$ algorithm [1], is currently being implemented in SeqAn by Johannes Krugel and it will be very interesting to see how STTD64 compares to DiGeST.
Appendix
A - Compact Disc

The complete source code as well as the DNA data used for evaluation is on the compact disc attached to this document.
Bibliography


