EFFICIENT SUFFIX TREE CONSTRUCTION

FREDERIK JERLOV

Master’s Thesis
Department of Computer Science
Science & Technology
Aarhus University

June 2019

Advisor: Christian Nørgaard Storm Pedersen
I would like to thank my advisor Christian Nørgaard Storm Pedersen for agreeing to advise me and guiding the writing of this thesis. Gudmund S. Frandsen also deserves a thanks for dragging me through my education.

For proof-reading my thesis I would like to thank my dad Mogens Jerløv, and friends Lauge Hoyer, Emma Hillgaard and FUES.

Furthermore my mom Gitte Troelsen also deserves thanks for moral support throughout my studies at Aarhus University.

Additionally I would like to thank my fellow students and neighbours at my dormitory Kollegium 2.3, Parkkollegierne for always being up for a movie, a conversation or a beer in time of distress.

A special thanks goes out to Tagekammeret for making my extended stay at Aarhus University both memorable and enjoyable.

Desuden mener jeg at Tagekammeret vil bestå!

Frederik Jerløv
Aarhus, June 2019
ABSTRACT

This thesis describes the theory, implementation and evaluation of two algorithms for suffix trees construction. In particular how we go from the naïve algorithm \cite[Section 5.4]{1} running in $O(n^2)$ to the efficient McCreight’s algorithm \cite{2} running in $O(n)$. McCreight’s algorithm obtains linear-time running time by using two tricks, suffix links and the fastscan algorithm.

This thesis contributes by explaining the theory of going from the naïve algorithm to McCreight’s algorithm in an elegant manner.

Furthermore this thesis provides a very simple and clean implementation of McCreight’s algorithm.

Exploration of the two algorithms are done by rigorous testing with different datasets, revealing connections with theory.

The naïve algorithm is found to be worst case $O(n^2)$ but converging towards $O(n)$ when repetition is low. McCreight’s algorithm is found to be linear for $|\Sigma| = 1$ and almost linear for $|\Sigma| > 1$ with the suspected evildoer being memory locality, which should be explored in future work.

All code and datasets for this thesis is freely available at \url{http://github.com/fjerlv/thesis/}. 
A.1 Naive .................................................. 39
A.2 McCreight ............................................. 40
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a) Trie and b) Compact trie for the strings TACT$ and TATCS</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>LCP of $S_i$ and $S_j$</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>LCA of $S_i$ and $S_j$</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Suffix link for the suffix tree for the string BANA$. The links being the dotted lines</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>The suffix tree for $S = $ BANANA$</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>head$(i)$ and tail$(i)$ with the terminal node i</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>Inserting ab$ into abaab$ resulting in splitting a node and inserting the new tail</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>Construction of suffix tree for the string bba$ with the naïve algorithm using slowscan</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>Visualising [Lemma 3.3.1]. This figure is inspired by slides from the course String Algorithms, 2017</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>How to find w. Start at head$(i)$, finding s(head$(i)$) is done by finding parent(head$(i)$) and using its suffix link to arrive in the “potential” other sub tree s(parent(head$(i)$)) where w, or s(head$(i)$) is a sibling. Now the search for head$(i+1)$ can start</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>Python IDE PyCharm running cProfile profiler on McCreight</td>
<td>21</td>
</tr>
<tr>
<td>12</td>
<td>Naïve on alphabet-a – Running time in seconds over input length n</td>
<td>25</td>
</tr>
<tr>
<td>13</td>
<td>Naïve on alphabet-a – Running time in seconds over input length n divided by $n^2$</td>
<td>26</td>
</tr>
<tr>
<td>14</td>
<td>Naïve – Comparing running time in seconds on alphabet-a and alphabet-acgt over input length n</td>
<td>27</td>
</tr>
<tr>
<td>15</td>
<td>Naïve on alphabet-ab, alphabet-acgt and alphabet-20 with running time in seconds over input length n</td>
<td>27</td>
</tr>
<tr>
<td>16</td>
<td>McCreight – Running time in seconds over input length n for all datasets</td>
<td>28</td>
</tr>
<tr>
<td>17</td>
<td>McCreight on all datasets – Running time in seconds, divided by n, over input length n for all datasets</td>
<td>29</td>
</tr>
<tr>
<td>18</td>
<td>Naïve and McCreight on dataset alphabets – Running time in seconds over alphabet size length $</td>
<td>\Sigma</td>
</tr>
</tbody>
</table>
Figure 19 Naïve and McCreight on dataset chr1 – Running time in seconds over input length n.

Figure 20 Naïve on dataset chr1 showing time on the first y-axis and number of comparisons in slowscan on the second y-axis.

Figure 21 Naïve and McCreight on alphabet-20 – Running time in seconds over input length n.

LISTINGS

Listing 1 Python-like Pseudocode for McCreight’s algorithm. The implementation of the pseudocode can be found in [Appendix A.2, line 69 – 102]. It is important to note that slowscan is responsible for splitting, adding tail and returning the related head and tail. Fastscan is responsible for returning the most recent node and the cursor location on that node.

Listing 2 Implementation: fastscan to find w

Listing 3 Implementation: w is a node

Listing 4 Implementation: w is on an edge

Listing 5 Slowscan from naive.py

Listing 6 Fastscan from mccreight.py

Listing 7 Split from naive.py

Listing 8 Reverse Split from mccreight.py

Listing 9 Old fastscan from mccreight.py

Listing 10 New fastscan from mccreight.py

Listing 11 How to use Naïve and McCreight

Listing 12 naive.py

Listing 13 mccreight.py
A suffix tree is a data structure solving a myriad of string searching and matching problems in an often efficient, simple and elegant manner. Some of these problems involve finding tandem repeats\(^5\), finding exact matches and solving the longest common substring problem \(^4\), Sec. 5.2.5.

These fast and efficient solutions come at a cost, which is preprocessing of the string. In other words – constructing the suffix tree. This demands both time and space. Fortunately this can be achieved in linear-time and space.

In 1973, the first linear-time algorithm for suffix tree construction with constant-size alphabets, were devised by Weiner\(^6\) and was called “the algorithm of 1973” by Donald Knuth.

A few years later McCreight\(^2\), gave us a simpler, space efficient linear-time construction algorithm.

In this thesis I will explain the journey from the naïve algorithm, running in worst-case \(O(n^2)\), to McCreight’s algorithm, running in worst-case \(O(n)\). McCreight’s algorithm is a clever and elegant solution building upon the naïve algorithm and doesn’t tamper much with the initial data structure.

1.1 \textsc{thesis outline}

We’ll start off in [Ch. 2] where we will look at the basic notation and framework for talking about suffix trees and their construction algorithms. If you feel well traversed within suffix trees, feel free to skip this chapter. It will be referred to often.

In [Ch. 3] we will look at suffix trees and how we can construct them. More specifically I will explain the theory allowing us to “travel” from the naïve algorithm to McCreight’s algorithm.

In [Ch. 4] we will talk about how I implemented the two algorithms naïve- and McCreight, and if I encountered any eventual complications in the process.

In [Ch. 5] we will experiment with the implementations and see if our theoretical expectations drawn from [Ch. 3] match the real world.

In [Ch. 6] we will conclude this thesis by talking about what have been done, and what future work could entail.
1.2 GitHub

All implementations, datasets, scripts etc. used in this thesis are publicly available at https://github.com/fjerlv/thesis.
Before we get our hands dirty with suffix trees, a few basic concepts are needed and will be explained in this chapter.

2.1 ALPHABET

An alphabet, denoted $\Sigma$, is a finite set whose elements are called characters. In this paper we assume a total ordering on the alphabet. The size of $\Sigma$ is denoted by $\sigma = |\Sigma|$. An example of an alphabet is the DNA alphabet

$$\Sigma = \{A, C, G, T\}.$$

2.2 PREFIX AND SUFFIX

Given a string $S$ of length $n$, we let $S[i \ldots j]$ denote the substring of $S$ that begins at position $i$ and ends at position $j$. $S[1 \ldots n]$ is thus the whole string $S$ and $S[1 \ldots j]$ is the prefix of $S$ that ends at position $j$ and $S[i \ldots n]$ is the suffix of $S$ that starts at position $i$. The suffix or prefix of $S$ is proper if it is not equal to the whole string $S$. Furthermore, $S[i]$ denotes the $i^{th}$ character of $S$.

2.3 TRIES AND COMPACT TRIES

A trie is a data structure for storing and retrieval of strings. The word trie coming from retrieval. If two strings have shared prefix they must also have shared initial path. To ensure that each leaf node corresponds to a root-to-leaf path we append a sentinel character $\$\$ to each string. For a number of unique strings, $m$, we end up with a trie containing $m$ terminal nodes. An example of a trie can be seen in [Fig. 1.a].

However, tries a lot of internal nodes relative to the string length which results in a high space consumption. This problem however can be improved upon.

A compact trie is a space-optimization of the trie. In a compact trie, nodes with one child are merged with its parents resulting in internal nodes with at least two or more children. Nodes then become responsible for substrings. The compact trie will then retain the same information and use less space than an uncompressed trie. An example of a compact trie, can be seen in [Fig. 1.b].
2.4 LCP AND LCA

Longest common prefix, LCP for short, for two strings \( u \) and \( v \), also denoted \( \text{LCP}(u,v) \) is the longest prefix which \( u \) and \( v \) have in common.

In this thesis I use LCP in the context of suffix trees [Sec. 3] and we denote these suffixes \( S_i = S[i...n] \) and \( S_j = S[j...n] \) for integers \( i, j \in [1...n+1] \) and write LCP for a string \( S = \text{BANANA} \) as such

\[
\text{LCP}(1,3) = \text{LCP}(S_1, S_3) = \text{LCP}('anana', 'ana') = 'ana'.
\]

An illustration of LCP can be seen in [Fig. 2].

We also need to talk about lowest common ancestor, LCA for short, which for two nodes \( u \) and \( v \) is the node farthest from the root that is an ancestor of both \( u \) and \( v \). Let’s call this node \( z \) and write \( z = \text{LCA}(i,j) \) for suffixes \( i \) and \( j \). This can be seen in [Fig. 3]. Since \( z \) is spelled out from the root of the suffix tree and it is the longest one we can write

\[
\text{LCA}(i,j) = \text{LCP}(i,j).
\]

2.5 SUFFIX LINKS

Suffix links connects a node representing \( x\alpha \) to \( \alpha \) where \( x \) is a single character and \( \alpha \) is a substring of \( S \). Note that the suffix link might be empty, and that this is the case for the root, which means that the root has a suffix link to itself.
What does this give us? If some string $x\alpha$ is an internal node then $\alpha$ must also be an internal node. The reasoning is this: If the string $x\alpha$ is an internal node, the string must occur more than once. Then the string $\alpha$ must occur at least as many times as $x\alpha$ and also be an internal node, i.e. a branching node. If such a link exists we call it a suffix link. So we can always find a prefix of a suffix, and for every single internal node $x\alpha$, we have a suffix link to $\alpha$. We say that when following a node, $u = \alpha x$, the suffix link $s(u)$ represents the string $\alpha$.

In [Fig. 4] we see the suffix links for the string $S\$ = BANANA$. Notice that two connected suffixes both have $\alpha$ in common.

We can now jump from one suffix to the next one.
A suffix tree is a compact trie containing all the suffixes, see [Sec(s) 2.2, 2.3], of a string $S$, including the empty string $\epsilon$. Suffix trees are one of the most important data structures for solving a myriad of string processing problems and are very fast at that once built. Let’s start with a definition of a suffix tree.

**Definition 3.0.1.** Suffix Tree

Let $S$ be a string of length $n$. A suffix tree for $S$ is a rooted tree $ST$ with the following properties:

1. $ST$ has exactly $n + 1$ leaves numbered $1$ to $n + 1$.
2. Each internal node in $T$ is branching, i.e., it has at least two children.
3. Each edge of $ST$ is labeled with a non-empty substring of $S$.
4. For each node $\alpha$ in $ST$ and each $\alpha \in \Sigma$, there is only one $\alpha$-edge $\alpha \xrightarrow{\text{av}} \beta$ for some string $v$ and some node $\beta = \alpha av$ in $ST$. In other words, no two edges out of a node can have edge labels beginning with the same character.
5. The key feature of $ST$ is that for any leaf $i$, the concatenation of the edge labels on the path from the root to leaf $i$ exactly spells out the string $S[i \ldots n]$, the $i$th suffix of the string $S$.

In Fig. 5 we see the suffix tree for the string $S = \text{BANANA}$.

You might wonder why we are appending a sentinel character $\$ to all our strings. Imagine having a suffix $s$ of $S$ and letting $s$ be a proper prefix$^2$ of another suffix of $S$. If we didn’t have $\$ then $s$ would be represented by an internal node instead of a leaf. Because the sentinel character does not occur in $S$ by definition, no suffix $s$ of $S$ can be a proper prefix of another suffix of $S$. This results in every single suffix of $S$ being represented by a leaf and thus every suffix is unique.

**What about space?** Let $n$ be the length of string $S$, then the suffix tree $ST$ of $S$ has exactly $n + 1$ leaves. Each internal node is branching so we have at most $n$ internal nodes. The number of edges is one less than the number of nodes and there are at most $2n + 1$ edges, so there are at most $2n$ internal nodes. When looking at the suffix tree for BANANASA in Fig. 5 it is tempting to let the nodes store the edge labels as it would be simple and intuitive, however

---

1 Borrowed from [3, Definition 4.4.1]
2 Not including the whole string.
doing so would result in a space consumption $O(n^2)$. Instead we store the string indices $(i, j)$ at the ingoing edge of the node, and arrive at space $O(2n) = O(n)$.

There are several ways to construct suffix trees, and in the following sections we will go through two of them. The first algorithm is the naïve approach [Sec. 3.1] running in $O(n^2)$ time, where $n$ is the length of the input, and thus the number of suffixes. The second one, and the more interesting one, McCreight’s algorithm in [Sec. 3.3] improves upon the naïve algorithm with suffix links and shortcuts using an algorithm called fastscan, achieving linear time.

### 3.1 Naïve Algorithm

This part is inspired by [1, Section 5.4]. To construct a suffix tree for a string $S$ we first create an initial tree consisting of the root node and a leaf node, letting the edge label between the two be suffix $S_1 = S[1...m]$. Then we iterate through all the remaining suffixes running from $i = [2...m]$ and insert suffix $S_i = S[i...m]$.

To figure out where to insert suffix $S_i$ we have to find the longest path possible where $S_i$ and $S_{i+1}$ share a prefix of $S[i+1...m]$, thus we need to find the longest common prefix [Sec. 2.4] also denoted $\text{LCP}(S_i, S_{i+1})$. This is the lowest point in our suffix tree that the path to the $i$th suffix could branch, from the branch to any previously calculated one. As a prerequisite to the next algorithm, let’s partition a suffix $i$ into two parts:

$$S[i..n] = \text{head}(i)\text{tail}(i),$$

where $\text{head}(i)$ denote the LCP$(i, j)$ of maximum length over all longer suffixes, $j < i$ and $\text{tail}(i)$ be the remaining part of the string so that $S[i..n] = \text{head}(i)\text{tail}(i)$. 

![Figure 5: The suffix tree for $S = \text{BANANA}$](image-url)
3.1 NAIVE ALGORITHM

The ability to locate \( \text{head}(i) \) fast becomes important later in McCreight’s algorithm as we shall have a look at later in [Sec. 3.3].

For an example see [Fig. 6].

We need to find \( \text{head}(i) \) and insert \( \text{tail}(i) \). We can find \( \text{head}(i) \) with the method called slowscan. Slowscan checks if the first character of \( S[i+1] \) is in the suffix tree starting at the root. If it’s not, we are done and we simply insert a new leaf node, \( \text{leaf}(i) \) onto \( \text{head}(i) \) which is just the root, with the edge label from the root being \( S[i+1...m]$ \) and this iteration is done.

![Figure 8: Construction of suffix tree for the string bba$ with the naïve algorithm using slowscan.](image)
If the first character is in the suffix tree, start scanning down along the path, comparing and matching every single character on the edges, until we see a mismatch.

When such a place is found, slowscan is done, we can say that no suffix of S$ from that point on is a prefix of any other suffix of S$. This place is either a node or in the middle of an edge. And we can argue that the path matching is unique, because no two edges out of a node can have the same labels that begin with the same character.

If we stop matching in the middle of an edge, \((u, v)\), we do a split by inserting a new node called \(w\) after the last character on the edge that matches our suffix and just before the first character on the edge that mismatched. This new edge \((u, w)\) is labeled with the part of \((u, v)\) that matched with \(S[i + 1 \ldots m]\) and the new edge \((w, v)\) is labeled with the last part of the \((u, v)\) edge label. Independent of doing the split or not, we insert a new edge \((w, i + 1)\) running from \(w\) to a new leaf labeled \(i + 1\) and label the new edge with the unmatched part of \(S[i + 1 \ldots m]\). When we have inserted all the suffixes we have a constructed suffix tree of string S$.

An example of this split can be seen in [Fig. 7]. And an example of the construction of the suffix tree for \(S = bba\) can be seen in [Fig. 8].

3.1.1 **Time and space complexity analysis**

This approach takes \(O(n^2)\) time, where \(n\) is the length of the string, which is also the number of suffixes \(m\), because we have to insert \(n - 1\) suffixes and for each suffix we potentially have to scan at most \(n\) characters before inserting.

Assuming \(|\Sigma| > 1\) and that the elements are randomly drawn from an alphabet, the naïve become faster with the size of LCP decreasing and converge against the time it takes to find a outgoing node. In other words, by chance, the larger the alphabet becomes, the less is the chance slowscanning all the way down an edge before finding a suitable match. We will explore this in [Chp. 5].

3.2 **Searching for patterns in a suffix tree**

Even if slowscan could becomes faster with increasing alphabet size, can we scan faster than slowscan? If we have node \(u\) and we are searching for a string \(v\) we are now searching for a prefix \(uv\). If we don’t know if \(uv\) is a prefix of our string \(x\) we have to continue with slowscan. If we hypothetically can somehow know that \(uv\) is a prefix of our string \(x\), then we only need to locate the exact position. We end up with the following two cases
1. If the length of \( v \) is larger than the edge we are looking at we can continue the search in the next node without slowscanning character by character.

2. If the length is lesser than the edge then \( v = u[i...|v|] \) we know that the scan should stop at the current edge and can jump the length of the search pattern we are missing.

This process of avoiding slowscanning by knowing that the prefix is in our string and jumping is called fastscan. We will see how this becomes useful when talking about McCreight’s algorithm in the next section.

### 3.3 McCreight’s Algorithm

This description of McCreight is inspired by \cite{4}, sec. 5.2.2 and "Suffix Tree Construction" slides from the course String Algorithms, 2017\cite{3}.

McCreight’s algorithm\cite{2}, from now on just McCreight, starts out with the same gist as the naïve suffix tree construction algorithm, it inserts the suffixes from \( i = 0 \) to \( n + 1 \) iteratively. However, it is very clever about finding \( \text{head}(i) \) and ends up spending \( O(n) \) time instead of \( O(n^2) \).

Before we talk about the algorithm we have to explain the McCreight trick and why it works.

First step in that process is [Lemma 3.3.1]. It says that the longest proper suffix of \( \text{head}(i) \), is a prefix of \( \text{head}(i + 1) \) and hence an ancestor of \( \text{head}(i + 1) \). This means that we can easily find the longest proper suffix of \( \text{head}(i) \) and take a shortcut.

**Lemma 3.3.1.** \cite[p. 116]{4} Let \( \text{head}(i) = x[i...i + h] \) for integers \( 1 \leq i \leq n \) and \( -1 \leq h \leq n - i \). Then \( x[i + 1...i + h] \) is a prefix of \( \text{head}(i + 1) \)

To illustrate [Lemma 3.3.1] see [Fig. 9]. *So why is this true?* First of all, let’s assume that \( h > 0 \), because \( h = 0 \) means that \( \text{head}(i) = \epsilon \). Let \( \lambda = x[i] \) and \( y = x[i + 1...i + h] \) so that \( \text{head}(i) = \lambda y \). By the definition of head, we know that there exists a \( j < i \) so that \( \text{LCP}(i, j) = \lambda y \) which means that suffix \( j + 1 \) and \( i + 1 \) share prefix \( y \) hereby \( y \) is a prefix of \( \text{LCP}(i + 1, j + 1) \) and thus \( y \) is a prefix of \( \text{head}(i + 1) \).

This lemma gives us two important things:

1. We can implement a suffix link function \( s \), see [Sec. 2.5], which connects any non-empty node \( u \) in \( T_x \) to the node \( v = s(u) \) being the longest proper suffix. This allows us to use \( s(\text{head}(i)) \) as a shortcut.

2. Knowing that \( \text{head}(i + 1) \) is an ancestor of \( s(\text{head}(i)) \) we can avoid slowscanning and fastscan instead. We know that both the string we are looking for and \( \text{head}(i + 1) \) is in the tree.

Figure 9: Visualising [Lemma 3.3.1]. This figure is inspired by slides from the course String Algorithms, 2017.

Listing 1: Python-like Pseudocode for McCreight’s algorithm. The implementation of the pseudocode can be found in [Appendix A.2, line 69 – 102]. It is important to note that slowscan is responsible for splitting, adding tail and returning the related head and tail. Fastscan is responsible for returning the most recent node and the cursor location on that node.

```python
def McCreight(string):
    root = Node()
    root.suffix_tree = root
tail = root.add(string)
head = root
for i in range(1, len(string)):
    if head.label == '\n':
        head, tail = slowscan(root, tail.suffix_link)
        continue
    u = head.parent
    if u.label != '\n':
        w, w_cursor = fastscan(u.suffix_link, head.label)
    else:
        w, w_cursor = fastscan(root, s(head).label)
    if w is on a edge:
        split_head, split_tail = w.split(w_cursor)
        w = split_head
        w.add(tail)
        head.suffix_link = w
        next_head = w
    if w is a node:
        next_head, tail = slowscan(w, tail)
        head.suffix_link = w
        head = next_head

Now we can focus on how McCreight works. We begin with a tree $T_1$, consisting of the root node, a node for $S_1 = S[1\ldots n]\$, and the label connecting the two nodes.

It is moreover clear that when the additions for $T_i + 1$ are done head$(i+1)$ is the only non-terminal node for which a suffix link can not yet be set.

Then for each $i = 1, 2, \ldots, n$, McCreight builds the following tree $T_i'$ from the preceding tree $T_i$ by inserting suffix$_i + 1$. It is worth noting that only the final tree $T_n'$ is a suffix tree for S.
In each iteration of the algorithm we must take the following steps:

1. Add node \( i + 1 \)
2. Add \( \text{head}(i + 1) \) if needed
3. Add \( \text{tail}(i + 1) \)
4. Add suffix link \( \text{head}(i) \rightarrow s(\text{head}(i)) \)

From 4) it follows that after each iteration \( T'_{i+1} \) is a linked suffix tree where only suffix links from non-terminal nodes are computed.

Of these four additions, 1) and 3) are trivial once \( \text{head}(i + 1) \) has been located, and 4) also follows from finding \( \text{head}(i + 1) \). So we should focus on 2) finding and possibly adding \( \text{head}(i + 1) \).

Let \( v = \text{label}(u, \text{head}(i)) \) and let us rewrite \( \text{head}(i) = uv \) so \( w = s(u)v \) has to be a prefix of \( \text{head}(i + 1) \). If \( \text{head}(i + 1) \) is not yet identified as a node in \( T' \), \( v \) must be spelled out by some path down from \( s(u) \), so our condition for fastscan is met and we can find \( w \) by fastscanning as such:

\[
w \leftarrow \text{fastscan}(s(u), v)
\]

As seen in my implementation [Listing 2].

Listing 2: Implementation: fastscan to find \( w \)

```python
u = head.parent
if u.end_index - u.start_index > 0:
    w, w_cursor = self.fastscan(u.suffix_link,
                                 head.start_index,
                                 head.end_index)
else:
    w, w_cursor = self.fastscan(self.root,
                                head.start_index + 1,
                                head.end_index)
```

So how do we quickly find \( w \) visually? Take a look at [Fig. 10]. We find ourselves at \( \text{head}(i) \) and want to find \( s(\text{head}(i)) = w \). Parent of \( \text{head}(i) \) gives us the ancestor which has a suffix link by invariant. Following the suffix link we get \( s(\text{parent}(\text{head}(i))) \) which has \( s(\text{head}(i)) = w \) as a sibling. We know by our [Lemma 3.3.1] that \( w \) contains \( \text{head}(i + 1) \). We find \( w \) from \( s(\text{parent}(\text{head}(i))) \) by fastscanning.

**Why does \( w \) exist in \( T_i \)?** \( \text{head}(i) \) is a prefix of \( x[j...n] \) for some \( j < i \). Hence \( w \) is a prefix of \( x[j + 1...n] \) for some \( j < i \). \( w \) is a prefix of some suffix \( j \leq i \) which means that \( w \in T_i \). So we can search for \( w \) using fastscan from \( s(u) \). \( w \) can now be a node, or on an edge.
Figure 10: How to find $w$. Start at $head(i)$, finding $s(head(i))$ is done by finding $parent(head(i))$ and using its suffix link to arrive in the “potential” other sub tree $s(parent(head(i))$ where $w$, or $s(head(i))$ is a sibling. Now the search for $head(i+1)$ can start.

In the first case where $w$ is a node. If $w$ already exist in $T'$ then some non-empty prefix of $x[|w|+1...n]$ might exist on a downward path from $w$. The longest such prefix is exactly $head(i+1)$, which may be computed by slowscanning as such:

$$head(i+1) \leftarrow slowscan(w, tail(i)).$$

As seen in my implementation [Listing 3].

Listing 3: Implementation: $w$ is a node.

```python
else:
    next_head, tail = self.slowscan(w, tail.start_index)
    head.suffix_link = w
```

In the second case $w$ is on an edge. This means that all suffixes $j < i$ with prefix $w$ agree on the next character. By our definition of $head(i)$ a $j < i$ exists such that two downward edges from $w$ spell out a non-empty prefix of $tail(i)$. So we split the edge by inserting $w$, updating $head(i)$ to $w$ and assigning $head(i+1) \leftarrow w$. As seen in my implementation [Listing 3].

Listing 4: Implementation: $w$ is on an edge

```python
if w.cursor < w.end_index:
    split_head, split_tail = \n        w.reverse_split(w.cursor,
                        self.string[w.cursor],
                        self.string[w.start_index])
    w = split_head
    tail = w.add(tail.start_index,
                   tail.end_index,
                   self.string[tail.start_index])
    head.suffix_link = w
    next_head = w
```
At the last step we need to add leaf \(i + 1\) and edge between head\((i + 1)\) and \(i + 1\). For the pseudocode of this algorithm see [Listing 1].

### 3.3.1 Time and space complexity analysis

Scanning is the only operating in the algorithm which is not done in constant time. So our worst case running time is essentially \(O(n + \text{slowscan} + \text{fastscan})\).

Let \(B\) be the maximum branch time over all nodes of the tree, defined as the maximum time required to select the correct matching downward path for the current search string.

Let's have a look at \(\text{slowscan}\) which we use to find head\((i + 1)\) after we've found \(w = s(\text{head}(i))\), or insertion in the case of \(\text{head}(i) = \epsilon\). Thereby we can be certain that \(s(\text{head}(i))\) is a prefix of head\((i + 1)\) so we can be sure that \(|\text{head}(i + 1)| - |\text{head}(i)| + 1 \geq 0\). This actually gives us the exact number of letters that slowscan need to process to find head\((i + 1)\) for \(i = 1, 2, \ldots, n\). Then summing over \(n\) steps we get number of letters scanned by slowscan in total. \(|\text{head}(n + 1)| - |\text{head}(1)| + n = n\) thus slowscan contributes \(O(nB)\) time.

Now to \(\text{fastscan}\). Fastscan jumps from node to node and thereby uses time proportional to the number of nodes. So we define \(d(v)\) as depth of for a node \(v\). And we quickly see that scanning increases the node depth and following parent and suffix links decrease the node depth. So fastscan is bounded by depth increase. Let’s start out with a proposition saying that \(d(v) \leq d(s(v)) + 1\). This is true because any ancestor \(u\) of \(v\) gives us that \(s(u)\) is an ancestor of \(s(v)\) except for the empty prefix and single letter prefixes of \(v\) in which case \(s(u)\) is different. Before calling fastscan we decrease depth by at most 2 because we go from head\((i)\) to parent(head\((i)\)), then from parent(head\((i)\)) to s(parent(head\((i)\)) this gives us a decrease of \(2n\). \(d(u) = d(\text{head}(i)) - 1\) and \(d(w) \geq d(u) - 1\). So fastscan is bounded by \(n\) plus the depth decrease. Time usage of fastscan is \(O(nB)\).

So what is the effect of the branch time \(B\) which we defined? Since our alphabet is ordered we can implement an ordered array or a search tree of the alphabet at each node. This node may contain as many as \(\Theta(|\Sigma|)\) entries which can be updated and searched in \(O(\log |\Sigma|)\). This completes the amortized analysis and the proof.
IMPLEMENTATIONS

In this chapter I will talk about how I implemented the two suffix tree construction algorithms, naïve [Sec. 3.1] and McCreight [Sec. 3.3]. I will also go through some problems I encountered while implementing them, how to use them, and how I compared them.

4.1 NAÏVE

I implemented the naïve suffix tree construction algorithm as described in [Sec. 3.1] and the implemntation was done in Python 3⁴.

A class called Node was created to hold variables start_index, end_index and edges. The first two variables allow us to store our suffixes in space $O(1)$, and the last variable is a dictionary² containing child nodes. Child nodes mapped to outgoing edges using their first character as key, which is unique by definition. An example of a dictionary containing outgoing edges:

```python
def slowscan(self, node, start_index):
    while True:
        if start_index == len(self.string):
            return node, node
        if self.string[start_index] in node.edges:
            edge = node.edges[self.string[start_index]]
            cursor = 0
            while start_index + cursor < len(self.string) and cursor < len(edge) and
                self.string[edge.start_index + cursor] == \
                self.string[start_index + cursor]:
```

Using a dictionary for storing the outgoing edges allows us to get the objects in time $O(1)$ on average³. An amortized worst case of $O(n)$, however this is acceptable considering our typical alphabet size being both constant and small, resulting a small number of outgoing edges from each node.

Inserting suffixes were done in a for loop from $2 \rightarrow n$ calling slowscan() for each suffix, where slowscan has the responsibility of both potentially splitting and inserting the tail as discussed earlier. See listing 4.1.

Listing 5: Slowscan from naive.py

---

2. https://docs.python.org/3/library/stdtypes.html#typesmapping
To test correctness of the algorithm I generated a handful of suffix trees for simple strings with the tool found at https://visualgo.net/bn/suffixtree and used them to write Python test cases for my algorithm. I also added a few test cases for a suffix trees found on lecture slides. This gave me confidence that the algorithm behaves as expected.

The source code of the implementation can be found at https://github.com/fjerlv/thesis/blob/master/src/suffixtree/naive.py and in appendix [A.1].

4.1.1 Problem 1 — Recursion

Slowscan were implemented using recursion. This caused problems pretty early on as I started getting `RecursionError`.

RecursionError: maximum recursion depth exceeded in comparison

This means that the recursion has hit a maximum recursion depth of the default value, 1000. This is meant to prevent infinite recursion causing an overflow in the underlying C stack. As a quick band aid I increased the limit to 5000 with `sys.setrecursionlimit(5000)` but I ended up with executions crashing.

This is because Python doesn’t have tail recursion and recursion as such should be avoided.

I ended up rewriting slowscan to be iterative instead, which in hindsight should have been my initial approach when using Python.

4.2 McCREIGHT

I have implemented McCreight’s algorithm as described in [Sec. 3.3] and the implementation was done in Python 3.

The implementation builds upon our Node class from the previous section and by extending the class with variables `suffix_link` and

---

4 [https://docs.python.org/3/library/exceptions.html#RecursionError](https://docs.python.org/3/library/exceptions.html#RecursionError)
5 [https://docs.python.org/3/library/sys.html#sys.setrecursionlimit](https://docs.python.org/3/library/sys.html#sys.setrecursionlimit)
parent allowing us to implement the McCreight’s algorithm. The suffix link variable is \texttt{None} if not set, and points to another \texttt{Node} object if set. The parent variable points to the parent unless the node is the root of the tree in which case it is set to \texttt{None}.

Slowscan from the previous section was used with minor changes. Fastscan [Listing 6] was also implemented.

Slowscan returns head and tail of the newly inserted leaf, and fastscan returns head, of the scan, and the cursor location.

Listing 6: Fastscan from \texttt{mccreight.py}

```python
def fastscan(self, node, v_start_index, v_end_index):
    while True:
        if v_end_index - v_start_index == 0:
            return node, node.end_index
        edge = node.edges[self.string[v_start_index]]
        if v_end_index - v_start_index >= edge.end_index - edge.start_index:
            node = edge
            v_start_index = v_start_index + edge.end_index - edge.start_index
            continue
        else:
            return edge, edge.start_index + v_end_index - v_start_index
```

To test the correctness of my McCreight implementation I did the easiest thing first, I duplicated the test cases from the naïve algorithm and saw that all the test cases passed.

Then I wrote a correctness-tester which compared the suffix trees of two implementations by going through the two trees \textit{depth-first}, comparing edge labels and outgoing edges.

I compared the two implementations, naïve and McCreight, on all permutations of \{$a, c, gt\}$ of size 1...8, all permutations of \{$a, b\}$ of size 1...12, mississippi$ and the first 500 characters of the dataset chr1 [Sec. 5.2.6]. Assuming the naïve implementation is correct, I gained confidence that my McCreight implementation was correct as well, as all the comparison tests passed.

This gave me confidence that both implementations were implemented correctly. However, it is important to note that this does not constitute a correctness proof, but strengthens my belief that the implementation are correct.

The source code of the implementation can be found at \url{https://github.com/fjerlv/thesis/blob/master/src/suffixtree/mccreight.py} and in the appendix [A.2].

4.2.1 Problem 1 — Splitting

While introducing suffix links and parents to \texttt{Node} I encountered a few problems. I started out splitting the same way as in the naïve implementation, see [Listing 7].

Listing 7: Split from \texttt{naive.py}

```python
def split(self, split_location, first_character):
```
This split introduced a new middle node and moved all the children to it, however, these children still had the previous node as parent. Furthermore, when doing the split on the current head, we would alter head and thus set the next head incorrectly, resulting in a non-functioning implementation of McCreight.

At first I tried updating the parent and suffix_link pointers by looping through all the outgoing edges of the splitting node, thus introducing average-case $O(n \log |\Sigma|)$ complexity for each split.

This seemed unnecessary and I ended up doing a reverse split by introducing the new node as parent. This allowed the children to keep their parent without introducing an alphabet-sized loop to correct the change. It also allowed suffix links to maintain the correct reference.

The new improved split `reverse_split()` can be seen in [Listing 8].

Listing 8: Reverse Split from mccreight.py

```python
def reverse_split(self, split_location, first_character, new_parent_character):
    new_parent = Node(self.start_index, split_location)
    new_parent.edges = {first_character: self}
    new_parent.parent = self.parent
    self.parent = new_parent
    new_parent.parent.edges[new_parent_character] = new_parent
    self.start_index = split_location
    return new_parent, self
```

4.2.2 Problem 2 — Slowscanning after fastscanning

Early in the implementation process my McCreight implementation were running in $O(n^2)$ for $|\Sigma| = 1$. Refactoring the suspected lines as `fastscan_slowscan()` and running a profiler on the algorithm, with the first 7000 characters of the dataset alphabet-a [Sec. 5.2.1] as input, shows that `fastscan_slowscan()` inside `fastscan()` is taking up all the resources for the computation, see [Figure 11]. The problem was, that I had ended up slowscanning in the first place! When searching for $v$ in fastscan, and running out of nodes to jump to, I resorted to slowscanning as seen in [Listing 9, line 10 – 14].

Listing 9: Old fastscan from mccreight.py

```python
def fastscan(self, node, v_start_index, v_end_index):
    while True:
        if v_end_index - v_start_index == 0:
            return node, node.end_index
        edge = node.edges[self.string[v_start_index]]
        if v_end_index - v_start_index >= edge.end_index - edge.start_index:
            node = edge
            v_start_index = v_start_index + edge.end_index - edge.start_index
            v_end_index = v.start_index + edge.end_index - edge.start_index
```

```python
def fastscan_slowscan(self, node, v_start_index, v_end_index):
    while True:
        if v_end_index - v_start_index == 0:
            return node, node.end_index
        edge = node.edges[self.string[v_start_index]]
        if v_end_index - v_start_index >= edge.end_index - edge.start_index:
            node = edge
            v_start_index = v_start_index + edge.end_index - edge.start_index
            v_end_index = v.start_index + edge.end_index - edge.start_index
```
When the naïve algorithm was given a dataset with $|\Sigma| = 1$, it resulted in the worst-case for the algorithm. This makes sense as we are doing a slowscan for the whole string each time fastscan has been called, as head is always the root node. On top of that we also did all the extra McCreight work. This was a mistake and since it is known that the last part of the search pattern $v$ is in fact on the edge, we can simply jump down the last edge without slowscanning at all. Replacing the slowscan from above with the following code [Listing 10, line 11] solved the problem.

Listing 10: New fastscan from mccreight.py

```
def fastscan(self, node, v_start_index, v_end_index):
    while True:
        if v_end_index - v_start_index == 0:
            return node, node.end_index
        edge = node.edges[self.string[v_start_index]]
        if v_end_index - v_start_index >= edge.end_index - edge.start_index:
            node = edge
            v_start_index = v_start_index + edge.end_index - edge.start_index
            continue
        else:
            return edge, edge.start_index + v_end_index - v_start_index
```

Figure 11: Python IDE PyCharm running cProfile profiler on McCreight.
4.3 HOW TO USE

Both algorithms are Python classes, *Naive* and *McCreight*, which take the input string as parameter. After initialization it contains the suffix tree of the input string plus a sentinel character, in our case $. The root can be retrieved and traversed with the class method *get_root()*.

Listing 11: How to use *Naive* and *McCreight*

```python
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
>>> from src.suffixtree.naive import Naive
>>> from src.suffixtree.mccreight import McCreight
>>> Naive('mississippi').get_root()
<src.suffixtree.naive.Node object at 0x7efd8cf57ef0>
>>> McCreight('missisippi').get_root()
<suffixtree.mccreight.Node object at 0x7f06d45b5ac8>
```
EXPERIMENTS

We started discussing the theory in [Chp. 3], we implemented the algorithms based on the theory from [Chp. 4] and now we want to set up experiments to test our implementations and answer a crucial question – Do the algorithms perform as expected?

5.1 ENVIRONMENT

The runtime benchmarks were done on a Lenovo Thinkpad T430 laptop running Python 3.6.7 with the following specifications:

**OS** Ubuntu 18.04.2 LTS

**CPU** Intel i5-3230M 4 cores @ 2.60 GHz

**RAM** 7 GB @ 1.33 GHz

Running time was measured inside the program and background tasks were minimized. It is also worth mentioning that the laptop was plugged into a power outlet to avoid battery saving features. The plots were made with the Python 2D plotting library matplotlib\(^1\). And to minimize the visual impact of outliers caused by CPU spikes, I sampled 10 executions, sorted the samples, removed the last 5 as potential outliers and took the mean of the remaining 5 samples.

5.2 DATASETS

I will use multiple artificial datasets and a single real dataset to test my implementations.

5.2.1 alphabet-a

An artificial string with 10,000,000 consecutive a’s is used for testing whenever our algorithms are in fact affiliated with the length of the input string, as stated in [Ch. 3]. This is the case where $|\Sigma| = 1$. The string can be found at https://github.com/fjerlv/thesis/blob/master/src/input/alphabet-a.

\(^1\) https://matplotlib.org/
5.2.2  *alphabet-ab*

An artificial string with characters drawn randomly from the alphabet \( \Sigma = \{a, b\} \) of length 10,000,000. The string can be found at https://github.com/fjerlv/thesis/blob/master/src/input/alphabet-ab.

5.2.3  *alphabet-acgt*

An artificial string with characters drawn randomly from the alphabet \( \Sigma = \{a, c, g, t\} \) of length 10,000,000. The string can be found at https://github.com/fjerlv/thesis/blob/master/src/input/alphabet-acgt.

5.2.4  *alphabet-20*

An artificial string with characters drawn randomly from an alphabet \( |\Sigma| = 20 \) of length 10,000,000. The alphabet used is also called the single-letter amino acid code:

\[
\Sigma = \{G, A, L, M, F, W, K, Q, E, S, P, V, I, C, Y, H, R, N, D, T\}.
\]


5.2.5  *alphabets*

A series of artificial strings. For alphabet size performances I generated strings of length 100,000 with randomly drawn characters from the alphabet of size \([1...62]\) of the characters \([0-9a-zA-Z]\). The strings can be found at https://github.com/fjerlv/thesis/tree/master/src/input/alphabet.

5.2.6  *chr1*

A real DNA string of length 10,000,000 obtained from a previous course where I removed all n-characters and ended up with a file of length 9,796,492 with alphabet size \( |\Sigma| = 4 \). This is the only real data for the following experiments, and serves to argue whether the algorithms are usable on real data or not.

5.3  *naïve*

I want to test if the simple suffix tree construction algorithm, as described in [Sec. 3.1], is running in quadratic time \( O(n^2) \). In the naïve algorithm, suffix tree construction algorithm is dominated by slowscan and the worst-case for slowscan is that we have to scan al-
most all the way down before discovering the location for each suffix where we need to split and insert the leaf $$. The first experiment I would like to conduct is using dataset alphabet-a [Sec. 5.2.1]. When using this dataset we are creating suffix trees with increasingly longer strings of consecutive a’s, and doing so I expect the running time to increase quadratically with respect to the length of the string, which connects with our worst-case running time analysis from [Sec. 3.1].

In [Fig. 12] we clearly see that it is growing by the power of two as expected. If you aren’t that easily persuaded, let us divide the y-axis, time, with $n^2$, then our measurements should converge against some constant and thus a slope $\approx 0$. The result can be seen in [Fig. 13] and it verifies our expectation. Our naïve algorithm runs in expected time $O(n^2)$ in the worst-case scenario.

Next, I want to test the algorithm using a more realistic alphabet of size 4 similar to DNA, hence I picked the alphabet-acgt. Comparing the two datasets, I expect alphabet-acgt to outperform alphabet-a, as there should be scenarios where slowscan doesn’t have to scan all the way down, due to the fact that the alphabet size is bigger. Let’s have a look at [Fig. 14]. It clearly shows that alphabet-acgt is much faster than alphabet-a. This connects with our theory that a larger alphabet size, with evenly distributed data, improves slowscan performance.
26 experiments

Figure 13: Naïve on alphabet-a – Running time in seconds over input length \( n \) divided by \( n^2 \).

How about alphabet-ab, alphabet-acgt and alphabet-20? I have compared them all in [Fig. 15]. It suggests that the naïve algorithm has a correlation between alphabet size \(|\Sigma|\), and running time. We will explore this later in [Sec. 5.5].

We can confirm that the worst-case scenario is \( O(n^2) \). However, it appears that the algorithm performs better on randomly drawn data with a bigger alphabet size.

5.4 MCCREIGHT

I want to test if McCreight’s, as described in [Sec. 3.3] is running in time \( O(n) \) as expected. Figuring out a worst-case isn’t straightforward, so let us start out by testing McCreight on all the datasets. See [Fig. 16]. It appears that all the datasets run on McCreight in \( O(n) \) time, which would suggest that McCreight runs in worst-case \( O(n) \) time. However, it is evident that some datasets run faster than others. To zoom in on the action, let us divide the y-axis by \( n \) and expect the lines to be \( \approx 0 \) if they run in linear time. See [Fig 17]. Here we see that alphabet-a runs in time \( O(n) \), however all other datasets are struggling a bit, as seen by their slightly upward slopes.

Why is that? The first clue might be that alphabet-ab is the worst performing of the bunch. A guess is that it’s caused by one of the constant time operations struggling with memory locality. More specifically I suspect suffix link jumping taking a longer time than expected. If a suffix link is forced to make increasingly longer jumps in mem-
Figure 14: Naïve – Comparing running time in seconds on alphabet-a and alphabet-acgt over input length $n$.

Figure 15: Naïve on alphabet-ab, alphabet-acgt and alphabet-20 with running time in seconds over input length $n$. 
ory, this constant time operation could become slower. In the case of alphabet-ab, many of the suffix link jumps have to go to a whole other subtree due to the nature of suffix trees. To test this guess, further experimentation is required. I will talk about this further in [Sec. 6.1].

Does McCreight have a best-case? For the naïve algorithm we concluded that consecutive a’s were the worst-case because it has to scan all the way down. With McCreight however, this seems to be the best case. Why is that? The next iteration always starts looking for \texttt{head}((i + 1)) from the root, and the depth of fastscan always becomes 0, then finding \texttt{head}((i + 1)) is simply jumping directly down on the current edge, which is done in constant time, thereby giving us a favorable time. When comparing all the datasets, as seen in [Fig. 16], this theory is very convincing.

5.5 Alphabet Size

For the first experiment let us use dataset \textit{alphabets} where we hold \( n = 10,000 \) and vary the alphabet size \( |\Sigma| \).

I expect naïve to be very dependent on the alphabet size and to converge against \( \mathcal{O}(n \log |\Sigma|) \) as the alphabet size increases, where with a constant sized alphabet is just \( \mathcal{O}(n) \). This is because the average LCP decreases when the alphabet size increases on a dataset randomly drawn from an alphabet, resulting in less slowscanning.

McCreight should also, to a lesser extent, be affected by the alphabet size, but that is due to the number of node jumps fastscan has to
perform, thus the depth of the tree, that makes it faster and not due to the amount of characters it has to scan. I expect it to be much faster in the beginning and then the naïve should start catching up.

We know that the naïve achieves worst-case time $O(n^2)$ with $|\Sigma| = 1$. We have also argued that McCreight has a best-case with $|\Sigma| = 1$. As these are two extremes and not that interesting in the real world, I have decided to omit $|\Sigma| = 1$ from the following experiment. This allows us to visually compare McCreight and naïve with different and more realistic alphabet sizes.

I will now test the two algorithms against alphabets from $|\Sigma| = 2\ldots62$. See [Fig. 18]. With the naïve algorithm we see that the larger the alphabet becomes, the faster the algorithm becomes, and it seems to be converging towards $O(n)$. With McCreight we also see an increase in speed as the alphabet increases, but the effect is not breathtaking. The larger alphabet size results in a shallow tree and thus fewer jumps are required to find $\text{head}(i+1)$. The McCreight implementation is mostly dependent on the effectiveness of Python dictionaries and suffix links jumps in memory.

Figure 17: McCreight on all datasets – Running time in seconds, divided by $n$, over input length $n$ for all datasets.
30 experiments

Figure 18: Naïve and McCreight on dataset alphabets – Running time in seconds over alphabet size length $|\Sigma|$.

5.6 Comparison

I’ve now shown the performance of the two algorithms, but we have yet to see how they compare.

5.6.1 chr1

Let’s start off with comparing some real DNA data from dataset chr1. The alphabet size of this dataset is 4, and since we have seen that McCreight outperforms naïve greatly at $|\Sigma| = 4$, from our alphabet experiment on random artificial data, I expect it to be a lot faster. See [Fig. 19].

We see, as expected, that McCreight is much faster at dealing with real data with alphabet size 4. However, the naïve is clearly deviating from the trend between 40,000 – 50,000 where it struggles to keep up. Let us dive into the dataset and figure out why. My best guess is that it is a result of very long LCPs causing a lot of slowscanning. To test my guess, I accumulated the number of slowscan comparisons into the variable self.comparisons, in the Naive-class. This variable will be accessible with a method called get_comparisons() and is used for plotting.

Let’s see if the number of comparisons mimics the running time of the naïve algorithm on dataset chr1 in [Fig. 19]. See [Fig. 20] for the results.
Figure 19: Naïve and McCreight on dataset \textit{chr1} – Running time in seconds over input length \(n\).

Figure 20: Naïve on dataset \textit{chr1} showing time on the first \(y\)-axis and number of comparisons in slowscan on the second \(y\)-axis.
The number of comparisons do mimic the running time nicely and there is a clear correlation. We can conclude that the deviation is just a sign of a lot of repetition, those being large LCPs causing a lot of slow scanning. However, we also see an increase larger than the number of comparisons, which we again can guess has something to do with memory locality as discussed earlier.
Figure 21: Naïve and McCreight on alphabet-20 – Running time in seconds over input length $n$.

5.6.2 Alphabet-20

Let us look at alphabet-20. Here I expect the naïve algorithm to be much more competitive as the alphabet size is large and the data is random. See [Fig. 21]. As expected, naïve is very competitive on this dataset.
CONCLUSION

In [Ch. 3] I talked about what suffix trees are, how to build them, the naïve algorithm and the McCreight algorithm. I have talked about why McCreight is expected to be faster, and made a case for its theoretical running time of $O(n)$.

In [Ch. 4] I have shown how I implemented the two algorithms in Python 3, showing caveats like recursion and splitting. I also discussed their correctness.

In [Ch. 5] I have concluded that the naïve algorithm runs in worst-case $O(n^2)$ but seems to converge against $O(n)$ when the alphabet size increases, on randomly generated datasets. I have also shown that the naïve algorithm seems decent on datasets with $|\Sigma| > 1$, however it performs poorly when the strings contain a lot of repetition.

I have also concluded that the McCreight algorithm almost runs in worst-case $O(n)$, and similarly favours a large alphabet but is not dependent on it for good performance. It is almost linear, however there are hints that memory locality plays a role in implementation.

6.1 FUTURE WORK

This thesis was not about the space allocation or memory locality. However, it seems this would be an obvious path to visit in future work.

In our experimentation with McCreight it was apparent that the running time was not linear, and I suspect memory locality to be the evildoer. If I had more time I would use a Python profiler\(^1\) to profile the constant time operation suffix link jumping, to see if it is a constant in practice.

If the problem is in fact memory locality, then it would be interesting to experiment with memory layouts for the suffix tree.

Furthermore, Python is not an ideal programming language for suffix tree construction due to the lack of low-level memory manipulation. A programming language like C++\(^2\) would be more appropriate.

---

1. https://docs.python.org/2/library/profile.html


IMPLEMENTATIONS

A.1 NAIVE

This is my implementation of the naive algorithm [Sec. 3.1] as discussed in [Sec. 4.1]. The source code can be found at https://github.com/fjerlv/thesis/blob/master/src/suffixtree/naive.py.

Listing 12: naive.py

class Node:
    def __init__(self, start_index, end_index):
        self.start_index = start_index
        self.end_index = end_index
        self.edges = {}

    def add(self, start_index, end_index, first_character):
        new_node = Node(start_index, end_index)
        new_node.parent = self
        self.edges[first_character] = new_node
        return new_node

    def split(self, split_location, first_character):
        new_child = Node(split_location, self.end_index)
        new_child.edges = self.edges
        self.edges = {first_character: new_child}
        self.end_index = split_location

    def __len__(self):
        return self.end_index - self.start_index

class Naive:
    def slowscan(self, node, start_index):
        while True:
            if start_index == len(self.string):
                return node, node
            if self.string[start_index] in node.edges:
                edge = node.edges[self.string[start_index]]
                cursor = 0
                while start_index + cursor < len(self.string) and
                     cursor < len(edge) and
                     self.string[edge.start_index + cursor] == \
                     self.string[start_index + cursor]:
                    cursor += 1
                if len(edge) == cursor:
                    node = edge
                    start_index = start_index + cursor
                    continue
                edge.split(edge.start_index + cursor, \
                            self.string[edge.start_index + cursor])
                edge.add(start_index + cursor, \
                         len(self.string), \
                         self.string[start_index + cursor])
            else:
                node.add(start_index, \
                         len(self.string), \
                         start_index + cursor
                         continue
        return
49
    return
50
51  def __init__(self, string):
52      self.string = list(string) + ['$']
53      self.root = Node(0, 0)
54      self.root.add(0, len(self.string), self.string[0])
55      for i in range(1, len(self.string)):
56          self.slowscan(self.root, i)
57
58  def get_root(self):
59      return self.root

A.2 MCCREIGHT

This is my implementation of McCreights algorithm [Sec. 3.3] as discussed in [Sec. 4.2]. The source code can be found at https://github.com/fjerlv/thesis/blob/master/src/suffixtree/mccreight.py.

Listing 13: mccreight.py

class Node:
    def __init__(self, start_index, end_index):
        self.start_index = start_index
        self.end_index = end_index
        self.edges = {}
        self.parent = None
        self.suffix_link = None

    def add(self, start_index, end_index, first_character):
        new_node = Node(start_index, end_index)
        new_node.parent = self
        self.edges[first_character] = new_node
        return new_node

    def reverse_split(self, split_location, first_character, new_parent_character):
        new_parent = Node(self.start_index, split_location)
        new_parent.edges = {first_character: self}
        new_parent.parent = self.parent
        self.parent = new_parent
        new_parent.parent.edges[new_parent_character] = new_parent
        self.start_index = split_location
        return new_parent, self

class McCreight:
    def slowscan(self, node, start_index):
        while True:
            if start_index == len(self.string):
                return node, node
            if self.string[start_index] in node.edges:
                edge = node.edges[self.string[start_index]]
                cursor = 0
                while start_index + cursor < len(self.string) and 
                    cursor < edge.end_index - edge.start_index and 
                    self.string[edge.start_index + cursor] == 
                        self.string[start_index + cursor]:
                    cursor += 1
                if edge.end_index - edge.start_index == cursor:
                    node = edge
                    start_index = start_index + cursor
```python
            split_head, split_tail = \n            edge.reverse_split(edge.start_index + cursor,
                self.string[edge.start_index + cursor],
                self.string[edge.start_index])
            new_tail = split_head.add(start_index + cursor,
                                        len(self.string),
                                        self.string[start_index + cursor])
            return split_head, new_tail
        else:
            new_tail = node.add(start_index,
                                len(self.string),
                                self.string[start_index])
        return node, new_tail

    def fastscan(self, node, v_start_index, v_end_index):
        while True:
            if v_end_index - v_start_index == 0:
                return node, node.end_index
            edge = node.edges[self.string[v_start_index]]
            if v_end_index - v_start_index >= edge.end_index - edge.start_index:
                node = edge
                v_start_index = v_start_index + edge.end_index - edge.start_index
                continue
            else:
                return edge, edge.start_index + v_end_index - v_start_index

    def __init__(self, string):
        self.string = list(string) + ['$ ']
        self.root = Node(0, 0)
        self.root.suffix_link = self.root
        tail = self.root.add(0, len(self.string), self.string[0])
        head = self.root
        for i in range(1, len(self.string)):
            if head.end_index - head.start_index == 0:
                head, tail = self.slowscan(self.root, tail.start_index + 1)
                continue
            u = head.parent
            if u.end_index - u.start_index > 0:
                w, w_cursor = self.fastscan(u.suffix_link, head.start_index, head.end_index)
            else:
                w, w_cursor = self.fastscan(self.root, head.start_index + 1, head.end_index)
            if w_cursor < w.end_index:
                split_head, split_tail = \n                    w.reverse_split(w_cursor,
                                    self.string[w_cursor],
                                    self.string[w.start_index])
                w = split_head
                tail = w.add(tail.start_index,
                              tail.end_index,
                              self.string[tail.start_index])
                head.suffix_link = w
                next_head = w
            else:
                next_head, tail = self.slowscan(w, tail.start_index)
                head.suffix_link = w
                head = next_head
        def get_root(self):
            return self.root
```