Pattern Matching using suffix trays, arrays and trees

Jens Olaf Svanholm Fogh, 20094924

Thesis

October 2014

Advisor: Christian Storm Pedersen

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Exact pattern matching is used in a wide range of applications in multiple fields of science. Until now the two most widely used data structures for exact pattern matching have been suffix trees and suffix arrays. This thesis examines a newly proposed data structure called *suffix tray*. The suffix tray has been proposed as an alternative to already known structures build in $O(n \cdot |\Sigma|)$ time using $O(n)$ space, where n denotes the size of the text and $|\Sigma|$ denotes the alphabet size.

When answering exact pattern matching queries, the suffix tray has a worst case query time of $O(m + \log(|\Sigma|))$ where m denotes the length of the query string. This is an improvement of the worst case complexities for suffix trees and suffix array, which under the restriction of $O(n \cdot |\Sigma|)$ time for preprocessing and $O(n)$ space have a worst case complexity of respectively $O(m \cdot \log(|\Sigma|))$ and $O(m + \log(n))$. The suffix tray achieves the improved worst case complexity by combining the ideas from suffix trees and suffix array.

In this thesis information about suffix trays, suffix arrays and suffix trees are provided, in a depth enabling a reader with a background within computer science to implement the three structures.

The three structures have been implemented in Java, and experiments have been performed to see in which extend the theoretical worst case complexities can be seen in practice. The results from these experiments showed that the worst case complexities of the implementations where consistent with the theory. The constants in the asymptotic running times were hereafter approximated for the implementations.

The practical relevance of the new suffix tray structure was investigated through comparison with suffix trees and suffix arrays in experiments designed to simulate real life scenarios. These experiments showed that the suffix tray not only are of theoretical relevance but also practical relevance, as the suffix tray was the preferable structure for small query sizes.

Contents

Chapter 1

Introduction

The exact pattern matching problem is a well known problem in computer science, and is found in a wide range of different applications such as text editing, data compression, cryptology and molecular biology. Let Q denote a query string of length m and T denote a text of length n, both over the same alphabet Σ of size $|\Sigma|$. The problem to be solved in exact pattern matching, is then queries on the form *Is Q in T?* . The main focus for this thesis is the newly proposed data structure from [1] called *suffix tray*. Suffix trays supports exact pattern matching and are proposed as an alternative to already known structures build in $O(n \cdot |\Sigma|)$ time using $O(n)$ space. Until know the two most widely used data structures for exact pattern matching have been suffix trees and suffix arrays. Common for suffix trees and suffix arrays is that both structures can be implemented in $O(n \cdot |\Sigma|)$ time using $O(n)$ space. Under these restrictions, the best worst case complexity when searching for a substring in a text becomes $O(m \cdot \log(|\Sigma|))$ for the suffix tree[4] and $O(m + \log(n))$ for the suffix array[3]. By combining the ideas from suffix trees and suffix arrays, the suffix tray achieves an improved complexity of $O(m + \log(|\Sigma|))$. Often when analyzing structures for exact pattern matching, the size of the used alphabet is seen as constant, which would make the complexity of the suffix tree and the suffix tray the same. The alphabet size is therefore important in this thesis, as it is in the alphabet size the nuance between the structures are found. The fact that it is relevant to look at the alphabet size as a varying size, is clear from the wide range of applications where the exact pattern matching problems is found. I.e. the alphabet size of genes are often seen as 4, as the alphabet here consists of A, C, G, T while the alphabet size in a typical English text is at least 27.

The goals of this thesis are to:

- Provide enough knowledge about suffix trays, suffix arrays and suffix trees to enable a reader with a background within computer science to implement the structures, and use them for exact pattern matching.
- Make implementations of the described suffix trays, arrays and trees.
- Test in which extend the worst case complexity of the search algorithms in the implemented suffix trays, arrays and trees corresponds to their theoretical worst case complexities.
- Test how the search algorithms in the implemented suffix trays, arrays and trees performs in a more realistic setup, to investigate whether suffix trays have a practical relevance.

1.1 Outline of Thesis

Chapter 2: In this chapter the theory and principles behind the three data structures are described in depth.

Chapter 3: In this chapter a link can be found to all code files, raw data files, plots and report files. Further more a short overview of the relevant files can be read as well as how the used implementations have been tested for correctness. If the used implementations deviates from the descriptions in chapter 2 it will be noted in this chapter.

Chapter 4: In this chapter experiments are performed to test whether the worst case complexity of the implemented search algorithms complies with their respective theoretical worst case complexity, as well as how the search algorithms performs compared to each other when used in realistic setups.

Chapter 5: In this chapter the conclusions from previous chapters are gathered and ideas for further work is presented.

1.2 Notations

The following notations have been used throughout the thesis.

T - Denotes the text in the exact pattern matching problem.

n - Denotes the length of T.

S_i - Denotes the suffix starting at index i in T.

Q - Denotes the query string in the exact pattern matching problem.

m - Denotes the length of Q.

$T[i]$ - Denotes the i'th character in the string, in this case in T.

$T[j..i]$ - Denotes the substring from index j to i, both endpoints included.

Σ - Denotes the alphabet.

$|\Sigma|$ - Denotes the size of the alphabet.

$\log(x)$ - Denotes the logarithmic function with base 2.

$\log^x(a)$ - The logarithmic function applied x times to a. I.e. $\log^2(a) = \log(\log(a))$

$A < B$ - Let A, B be strings, this then denotes that B is lexicographical larger than A according to the used alphabet.

Chapter 2

Data Structures

2.1 Suffix Tree

2.1.1 Outline of the suffix tree data structure

A suffix tree is a fast and space efficient data structure which supports exact pattern matching in $O(m \cdot \log(|\Sigma|))$ time. In addition to this, it is also a fairly simple data structure to implement, making it one of the most popular data structures for exact pattern matching. The main idea behind suffix trees is to utilize the fact that all substrings of T is a prefix of a suffix in T. Let S_i for $i = 1, 2, \ldots, n$ be the suffix starting at position i in T. An example of this property would be the text *Mississippi* and the substring "sip" which would be the prefix of $S_7 =$ sippi. To exploit this property the suffix tree is build as a trie containing the strings $S_1\$, S_2\$, \ldots, S_n\$$. \$ represents a character which is not contained in the alphabet and ensures that each string has a unique root-to-leaf path in the trie, a property that makes implementation easier without changing the space complexity. To make the data structure space efficient the trie is made compact. A trie is made compact by contracting all nodes with out degree one. This ensures a maximum of $n - 1$ inner nodes and n leafs giving a total of at most $2n - 1$ nodes. As each node requires constant space, the nodes can be stored within $O(n)$ space. In addition to the nodes, the trie contains at most $2n - 2$ edges as it is a tree structure. To be able to store an edge in constant space, it is exploited that each label is a substring of the original text. This makes it possible to store the labels as a pair of pointers into the original text, making the space consumption for a single edge constant, resulting in a total space usage of $O(n)$. The children of a note can be stored in different ways, resulting in different time complexities for the search algorithm. This part will be handled in the following subsection.

2.1.2 Search and construction algorithms

Searching in a suffix tree is a fairly simple matter. Starting from the root down the query string is matched a single character at a time. The query string is matched from index 0 and onwards until either the whole query string is matched or a match can not be found. Comparing two characters takes constant time, ultimately making the task to find the path down the suffix tree the challenging part.

This is done by repeatedly finding a child with a matching label on its edge, where after the query string and the label is matched one character at a time. The matching edge can be found by looking at the first character of the labels, as the labels of all edges going out from a node has a unique starting character. In worst case the labels on the followed path only have one character each, resulting in m child look-ups. This gives a worst-case complexity for the search algorithm of $O(m \cdot (\text{Time to find matching child edge}))$. Different techniques are used to index the children of a node. By using an array of size $|\Sigma|$ at each node, it is possible to find a potential matching child in constant time, giving a search complexity of $O(m)$. The space complexity for this solution is $O(n \cdot |\Sigma|)$. As the suffix tray in [1] is proposed as an improved solution to a data structure using $O(n)$ space, and the alphabet size is seen as a varying variable, this would not be a fair comparison. Instead [1] compare it with a suffix tree implementation that uses $O(n)$ space. The best search complexity reached with the demand of a space complexity of $O(n)$ is $O(n \cdot \log(\Sigma))$. This is reached by saving the children of a node in a way that enables binary search among the children and their respective labels. An example of such a structure is a balanced binary search tree.

It is possible to return the indices of all occurrences at an extra cost of $O(\#occ)$. This can be done using a simple tree traversal from the place where the last character is matched. As the subtree has $\#occ$ leafs and all inner nodes have an out degree of at least 2, there can be no more than $O(\#occ)$ nodes in the subtree.

To enable binary search it is necessary to perform a sorting of the children in a node. The construction complexity of a suffix tree supporting binary search among children is thus bounded by the complexity of sorting. This is due to the fact that we can use a suffix tree for sorting. By representing the items we wish to sort by a character, and thereafter build a suffix tree for the string containing all characters once, the resulting suffix tree will contain a root node with $|\Sigma|$ sorted children.
The naive approach when constructing a suffix tree is to add one suffix at a time, simply by comparing one character at a time from the root. The time complexity for the naive approach is $O(n^2 \cdot \log(|\Sigma|))$, as the children still has to be sorted. It is though feasible to reach a construction complexity of $O(n \cdot \log(|\Sigma|))$ as *Ukkonen's* algorithm and *McCreight's* algorithm are two practical algorithms which constructs a suffix tree in $O(n \cdot \log(|\Sigma|))$ time. All suffix trees in this thesis are constructed using Ukkonen's algorithm.

Ukkonen's algorithm

The description of Ukkonen's algorithm in this section, is based on the description of the algorithm in [4]. Ukkonen's algorithm builds a suffix tree in an iterative manner. After each iteration $i = 1, 2, \ldots, n+1$ a compressed trie containing $T\$[j..i]$ for all $j \leq i$ exists. The compressed trie at iteration i thus consists of all suffixes of the prefix $T\$[1..i]$. This implies that the trie after iteration $n+1$ contains all suffixes of $T\$$, and thereby is the suffix tree of T. In each iteration it is necessary to update the trie by appending $T[i+1]$ to $T[j..i]$ for all $0 \leq j \leq i$ as well as adding the new string $T[i+1]$ to the trie. A sketch for Ukkonen's algorithm can be written as follows:

Algorithm 1 Ukkonen's algorithm - Pseudo-code

```
for i = 1 to n do
  for j = 1 to i do
    Find T[j..i] and update it to T[j..i + 1]
  end for
  Add new string [T[i + 1]
end for
```

If the pseudo code is implemented naively the complexity becomes $O(n^3 \cdot \log(|\Sigma|))$. Adding $T[i+1]$ takes $O(\log(|\Sigma|))$, while the two for-loops and searching for $T[j..i]$ will have a complexity of $O(n^3 \cdot \log(|\Sigma|))$ as searching for $T[j..i]$ takes $O(n \cdot \log(|\Sigma|))$ time, and the appending of $T[i+1]$ can be done in constant time when $T[j..i]$ is found. It is therefore necessary to realize two things to get a complexity of $O(n \cdot \log(|\Sigma|))$.

(1) Only in $O(n)$ of the $O(n^2)$ cases in the two for-loops, is it necessary to find $T[j..i]$ and update it.

(2) Finding $T[j..i]$ can be done in $O(\log(|\Sigma|))$ time, when you are in possession of a pointer to $T[j-1..i]$.

When $T[j..i]$ is found there are three different cases. $T[j..i]$ can either end in a leaf, on an edge or in an inner node. The last two cases can then again be divided into two cases, namely the cases where $T[j..i+1]$ already resides in the trie and those where it does not. It is clear that in the cases where $T[j..i+1]$ already resides in the trie no updates are needed. By using a small trick the cases in which $T[j..i]$ ends in a leaf does not need to be updated either. This is achieved by setting all labels to a leaf to $[k..\infty]$ and define ∞ to represent $i+1$ at any time. In the cases where an update is performed, an inner node representing $T[j..i]$ is inserted as well as a leaf representing $T[j..i+1]$. It is clear that once a j ends in a leaf, this will be the case for the rest of the algorithm. As each update makes a new j end in a leaf (1) follows directly.

The challenging part is thus to know when an update is needed and only look at these cases. This can be achieved by utilizing 2.1.2 from [4], which here will be used without further proofs.

Lemma 1. *[4, lemma 5.2.4, p. 123]*

(A) *Let $j > 1$. If $T[j..i]$ ends in a leaf, then so does $T[j-1..i]$.*

(B) *Let $j < i$. If $T[j..i+1]$ already resides in the trie, then so does $T[j+1..i+1]$.*

Let j' be the largest j such that $T[j..i+1]$ ends in a leaf. It is then easy to keep track of j' as it only changes when a j-index is updated or if $T[i+1]$ is inserted as a leaf. Knowing j' it is possible to exploit (A) by letting the inner loop start from $j'+1$ in each iteration. By doing this the algorithm will never use time on an index once it ends in a leaf. Let j'' be the smallest j such that $T[j..i+1]$ already resides in the trie. According to (B) the inner loop can then break when it reaches j''. This means that for each iteration in the outer loop, at most one j-index will be found without being updated. Having $j'+1$ and j'' as boundaries for the inner loop, a total of $O(n)$ iterations in the two for-loops is achieved.

The second step is how to find $T[j..i]$ in $O(\log(|\Sigma|))$ time, when a pointer to $T[j-1..i]$ exists. Keeping a pointer to $T[j-1..i]$ is simple as $T[j-1..i]$ is found just prior to the search for $T[j..i]$. To find $T[j..i]$ it is utilized that $T[j..i]$ already resides in the trie. This enables the algorithm to jump from node to node, just by matching the first character on the edge in addition to the length of the edges label. Starting from the root each time, this gives a complexity of $O(n \cdot d \cdot \log(|\Sigma|))$, where d indicates the depth of the trie. This gives a complexity of $O(n^2 \cdot \log(|\Sigma|))$ as the depth of the trie is $O(n)$. It is therefore required to take some shortcuts which is why a pointer to $T[j-1..i]$ is needed. In addition to the pointer it is necessary to use suffix links. A suffix link is defined as follows:

Definition 1. *Let $s(T[j..i])$ be the suffix link of $T[j..i]$, then $s([T[j..i]]) := T[j+1..i]$, with the special case $s(T[i..i]) := root$.*

To make the suffix link effective, all inner nodes must have a suffix link at the end of an iteration. This can easily be achieved, as every time $T[j..i]$ is updated the algorithm proceeds to find $T[j+1..i]$, making it possible to update $s(T[j..i])$ to $T[j+1..i]$ within the same iteration. The special case where $j = i$ is inserted is not a problem, as $s(T[i..i])$ per definition is the root node. Having this invariant it is possible to find $T[j+1..i]$ from $T[j..i]$ by following the suffix link of $T[j..i]$'s parent. The parents suffix link is guaranteed to exist due to the invariant. Following the edge to the parent node clearly decreases the depth by one. Furthermore the difference in depth between an inner node, and the node pointed to by its suffix link can be at most one. This is the case as the depth of $T[j..i]$ is increased by one every time there exists a triplet (k, i', i'') such that $k < j$, $i', i'' < i$ and $T[j..i'] = T[k..i'']$. For every such triplet it also applies that $T[j+1..i'] = T[k+1..i'']$,

except for the special case where $k = i''$ hence the possible difference of one. The total reduction in depth is thus constant when following a suffix link, and since suffix links only are followed when a new j-index is updated this gives a total reduction in depth of $O(n)$. As the depth of the trie is $O(n)$ this implies a total number of $O(n)$ jumps between nodes. Each jump uses $O(|\Sigma|)$ time to find the matching edge resulting in a total work load of $O(n \cdot |\Sigma|)$. In addition to this $O(n)$ nodes are inserted, each at a cost of $O(|\Sigma|)$ giving a total work load of $O(n \cdot |\Sigma|)$ inside the two for-loops.

As the number of iterations in the two for-loop are decreased to $O(n)$ and the total work load inside the two for-loops are decreased to $O(n \cdot |\Sigma|)$ a time complexity of $O(n \cdot |\Sigma|)$ is reached.

2.2 Suffix Array

2.2.1 Outline of the suffix array data structure

A suffix array (SA) representing a text T, is an array containing pointers to $S_1, S_2, \ldots, S_n$. These pointers are sorted in lexicographical order according to the suffixes they represent. Let v be a node in a suffix tree with sorted children, it then applies that the subtree of v is equivalent to a continuous part of the suffix array. The suffix array structure can be extended to answer the same queries as suffix trees, by enhancing it with utility arrays such as a longest common prefix (lcp) array. The construction time for suffix arrays matches the construction time for suffix trees with a complexity of $O(n \cdot \log(|\Sigma|))$, while the search complexity on the other hands differ, as the complexity for the suffix array here is $O(m + \log(n))$.

2.2.2 Search and construction algorithms

Searching in a suffix array is done in a binary fashion. The naive approach is a binary search, where the query string is matched from index 0 each time. Using this approach the search algorithm gets a complexity of $O(m \cdot \log(n))$.

This can be improved by using an enhanced suffix array. In [3] Manber and Myers shows how to obtain a search complexity of $(m + \log(n))$. Let (L, M, R) be respectively the index of the left endpoint, the midpoint and the right endpoint of the current interval in each iteration of the binary search. For a specific T and M the pair (L_M, R_M) will always be the same, as the binary search always divides the interval in the same manner. With n possible indices as midpoint this results in n possible (L_M, M, R_M) triplets, all with a unique M. In [3] two auxiliary arrays are created containing information about these triplets. The two arrays are called Lcp_M and Rcp_M and are both of size n. The arrays contains respectively $|lcp(S_{SA[L_M]}, S_{SA[M]})|$ and $|lcp(S_{SA[M]}, S_{SA[R_M]})|$ at index M.

Let l, r denote $|lcp|$ between Q and the suffix at the leftmost/rightmost indices in the current interval, where Q denotes the query string in the exact pattern matching problem.

The algorithm is then as follows. At the start of the algorithm l is $lcp(Q, S_{SA[0]})$ and r is $lcp(Q, S_{SA[n-1]})$. The algorithm then checks if $r \geq l$. The two branches are symmetric, thus only the branch for $r \geq l$ will be explained here. In the implementation used in this thesis the check performed is $l \geq r$, which is how it is explained in [3]. Exactly how the $r < l$ branch of the algorithm is can thus be seen in [3], as it does not change the content of the branch whether $r < l$ or $r \leq l$. Assuming $r \geq l$ the next step of the algorithm is to compare r with $Rcp_M[M]$ with three possible outcomes:

(1) $r < Rcp_M[M]$: In this case $|lcp(S_{SA[M]}, S_{SA[R_M]})| > |lcp(Q, S_{SA[R_M]})|$. All suffixes right for the midpoint, including the midpoint, is therefore guarantied to have a longer prefix in common with $S_{SA[R_M]}$ than Q. It is therefore safe to assume that Q will reside in the left half of the interval, if it is a substring of T. A new iteration is started with $R = M$ and $M = \frac{M+L_M}{2}$ while r, l and L remains unchanged. r remains unchanged as $|lcp(Q, S_{SA[M]})| = |lcp(Q, S_{SA[R_M]})|$.

(2) $r = Rcp_M[M]$: In this case $S_{SA[M]}$ and Q shares at least the first r characters. A character by character comparison is therefore performed, starting from $T[SA[M]+r]$ and $Q[r]$ until a mismatch is found or the query string is fully matched. Let $p = |lcp(S_{SA[M]}, Q)|$, there are then two possible cases:

- $Q[p] > T[SA[M] + p]$: A possible match will reside in the right half. The next iteration is started with $L = M$, $M = \frac{M+R_M}{2}$ and $l = p$, while r and R remains unchanged.
- $Q[p] < T[SA[M] + p]$: A possible match will reside in the left half. The next iteration is started with $R = M$, $M = \frac{M+L_M}{2}$ and $r = p$, while l and L remains unchanged.

(3) $r > Rcp_M[M]$: In this case $|lcp(S_{SA[M]}, S_{SA[R_M]})| < |lcp(Q, S_{SA[R_M]})|$. This implies that Q has a shorter prefix in common with $S_{SA[M]}$ than with $S_{SA[R_M]}$. Being the opposite of case (1), this ensures that Q resides in the right half of the interval, if it is a substring of T. A new iteration is started with $L = M$, $M = \frac{M+R_M}{2}$ and $l = Rcp_M[M]$, while r and R remains unchanged.

The key observations are:
- The algorithm terminates when $max(l, r) = |Q|$.
- In none of the three cases are $max(l, r)$ decreased.
- The only case that does not take constant time is case (2) where single character matching is performed.
- In each iteration a maximum of one single character mismatch is performed.
- $max(l, r)$ will be increased by x when x characters are matched.

These observations combined, gives an upper bound of $m + \log(n)$ single character comparisons throughout the algorithm. As the single character comparisons are the only

work during an iteration that are not constant and the number of iterations is bounded by $O(\log(n))$ this implies a complexity of $O(m + \log(n))$.

When only one occurrence is needed, the index is returned the first time the search string has been matched. If the indices of all occurrences shall be returned, this can be done within an extra cost of $O(\#occ)$. The algorithm in [3] does not terminate when the first occurrence is found. Instead the next iteration proceeds with $R = M$ and the algorithm terminates when $R - L < 2$. When the algorithm terminates it is guaranteed that R will be the smallest index i where $Q \leq S_{SA[i]}$, and thus the index of the left most occurrence if any occurrences exists. With a symmetric algorithm the right most occurrence in the suffix array can be found. Having found the left most and the right most occurrences in $O(m + \log(n))$ time, it is straight forward to return the indices in $O(\#occ)$ time.

The pseudo code for finding the left most occurrence of the query string is provided in [3, p. 6] [1]. The pseudo code for finding the right most occurrence is symmetric, and is for the sake of completeness provided in algorithm 2, with the notations used in this thesis. It is easy to turn the pseudo code into the simple search algorithm as the only difference is, that the algorithm should return the first time an occurrence is found.

The algorithm are using three different arrays to reach its complexity, the suffix array and the two auxiliary arrays. The suffix array can be created in $O(n \cdot \log(\Sigma))$ time in multiple ways. Having a suffix tree with sorted children, the simplest solution is a depth first traversal of the suffix tree, where the children is visited in lexicographical order. This is an $O(n \cdot \log(|\Sigma|)$ time solution as a depth first traversal in lexicographical order clearly can be done in $O(n \cdot \log(|\Sigma|))$ time and section 2.1 shows how to build a suffix tree in $O(n \cdot \log(|\Sigma|))$ time.

Creating the two auxiliary arrays with a better or equally good complexity as the suffix array is on the other hand not straight forward. Finding lcp(S_i,S_j) can be reduced to finding the nearest common ancestor (NCA) of the two leafs representing S_i and S_j in the suffix tree. Harel et al. [2] shows how NCA(S_i, S_j) can be found in constant time, at the cost of $O(n)$ preprocessing. The time complexity for creating the auxiliary arrays thus becomes $O(n)$, while the complexity for the entire enhanced suffix array becomes $O(n \cdot \log(|\Sigma|))$. The data structures used to solve the NCA problems, can be created within $O(n)$ space, giving the enhanced suffix array a space complexity of $O(n)$ like it was the case for the suffix tree. In the following subsections it will be explained how Harel et al. [2] solves the NCA problem in constant time.

Solving the NCA problem

To solve the NCA problem, it is separated into two subproblems called *the nca depth problem* and *the depth problem*.

[1] If the pseudo code from [3, p. 6] is followed, the user should notice that there is an error on line 5 as line 5 should have been $w_r > a_{Pos[N-1]+r}$ instead of $w_r \leq a_{Pos[N-1]+r}$.

Algorithm 2 Finding the right most occurrence in the suffix array: $O(m + \log(n))$

1: $l \leftarrow lcp(Q, S_{SA[0]})$
2: $r \leftarrow lcp(Q, S_{SA[n-1]})$
3: **if** $r == m$ **then**
4: Return $n - 1$
5: **else if** $r < m$ **and** $T[SA[n-1] + r] < Q[r]$ **then**
6: Return no match found
7: **else if** $l < m$ **and** $T[SA[0] + l] > Q[l]$ **then**
8: Return no match found
9: **else**
10: (L,R) $\leftarrow$ (0,n-1)
11: **while** R-L > 1 **do**
12: M $\leftarrow$ (R+L)/2
13: **if** $r \geq l$ **then**
14: **if** $r > Rcp_M[M]$ **then**
15: $L \leftarrow M$
16: $l \leftarrow Rcp_M[M]$
17: **else if** $r == Rcp_M[M]$ **then**
18: $p \leftarrow |lcp(S_{SA[M]}, Q)|$
19: **if** $p == m$ **or** $Q[p] > T[SA[M] + p]$ **then**
20: $L \leftarrow M$
21: $l \leftarrow p$
22: **else**
23: $R \leftarrow M$
24: $r \leftarrow p$
25: **end if**
26: **else**
27: $R \leftarrow M$
28: **end if**
29: **else**
30: **if** $l > Lcp_M[M]$ **then**
31: $R \leftarrow M$
32: $r \leftarrow Lcp_M[M]$
33: **else if** $l == Lcp_M[M]$ **then**
34: $p \leftarrow |lcp(S_{SA[M]}, Q)|$
35: **if** $p == m$ **or** $Q[p] > T[SA[M] + p]$ **then**
36: $L \leftarrow M$
37: $l \leftarrow p$
38: **else**
39: $R \leftarrow M$
40: $r \leftarrow p$
41: **end if**
42: **else**
43: $L \leftarrow M$
44: **end if**
45: **end if**
46: **end while**
47: Return L (L will be the largest index i, where $S_{SA[i]} \leq Q$)
48: **end if**

Definition 2. *Let v, w be two nodes in a tree.*

nca depth problem: Find the depth d_a of the nearest common ancestor to v and w.

depth problem: Find v's ancestor of depth d_a.

It is then utilized that the two subproblems can be solved in constant time on complete binary trees [2, p. 342]. To solve the nca depth problem, let the nodes in the complete binary tree be numbered according to an in-order traversal of the tree. The two subproblems can then due to the properties of in-order numbering, be solved via calculations involving the in-order numbering combined with the hight of the two nodes and the depth d of the tree. To solve the nca depth problem on complete binary trees in constant time, three different cases have to handled. Let $sym(v)$ and $sym^{-1}(i)$ be respectively the in-order numbering of v and the node given the in-order number i. The three cases and how they are solved is then described in algorithm 3.

Algorithm 3 Solving the nca depth problem

v and w are unrelated: return $d - \lfloor \log(sym(v) \oplus sym(w)) \rfloor$.

v is an ancestor of w: return $d - h(v)$.

w is an ancestor of v: return $d - h(w)$.

By definition of the in-order numbering, v is an ancestor of w if $sym(w) \in [sym(v) - 2^{h(v)} + 1, sym(v) + 2^{h(v)-1}]$ and vice versa. This enables the algorithm to decide which calculation to use in constant time, and thereby solve the nca depth problem in constant time.
To solve the depth problem on complete binary trees in constant time the following calculation can be used:

Algorithm 4 Solving the depth problem

$h = d - d_a$: return $sym^{-1}(2^{h+1}\lfloor \frac{sym(v)}{2^{h+1}} \rfloor + 2^h)$

To reduce the NCA problem on arbitrary trees to the NCA problem on complete binary trees, two different auxiliary trees are created. The NCA problem on an arbitrary tree is reduced to the NCA problem on a compressed tree, which then again is reduced to the NCA problem on a complete binary tree. To illustrate how the tree transformations are done, the running example from [2] is used.

Reduction from arbitrary trees to compressed trees

In [2] the transformation from an arbitrary tree A to a compressed tree C is done by making a distinction between *heavy* and *light* edges in A. Let v be a node in A, $size_A(v)$ is then defined as the number of descendants to v in A, including v itself. An edge $v \to p_A(v)$ is defined to be light if $2 \cdot size_A(v) \leq size_A(p_A(v))$. If an edge is not light, it is defined to be heavy. As the size of $p_A(v)$ includes itself, at most one of its children can be connected via a heavy edge. A is therefore divided into a collection of heavy paths. The node with the smallest depth in a heavy path is denoted as the *apex* of the heavy path. If a node is not connected to any other nodes via a heavy path, the node becomes the apex of a self contained heavy path.

With the definitions in place, a transformation from A to C can be made.

Definition 3. *Compressed tree C of arbitrary tree A [2, p. 343]*

The set of nodes in A and C is the same.

The set of edges in C is defined as $\{v \to apex(p_A(v)) | v$ is a node in A other than the root$\}$

In [2] Harel et al. shows how to find $NCA_A(v, w)$ in constant time, by using algorithm 5 under the assumption that $NCA_C(v, w)$ can be found in constant time. How [2] solves the NCA problem on C in constant time will be described in the next subsection, for now it will just be assumed that it is possible. In [2] algorithm 5 is presented without proof of correctness. Proves of correctness for the four steps are therefore provided here.

Algorithm 5 Finding $NCA_A(v, w)$

Let $u = NCA_C(v, w)$

(1) Either $u = v$, $u = w$ or u is the apex of the heavy path containing $NCA_A(v, w)$.
If $u = v$ or $u = w$: return u
Else: Look up $d_C(u)$

(2) Compute the ancestor v' of v in C whose depth is $d_C(u) + 1$.
If $apex(v') = u$ (v' and u are on the same heavy path): Let $v'' = v'$
Else: Let $v'' = p_A(v')$

(3) Compute the ancestor w' of w in C whose depth is $d_C(u) + 1$.
If $apex(w') = u$ (w' and u are on the same heavy path): Let $w'' = w'$
Else: Let $w'' = p_A(w')$

(4) Return as $NCA_A(v, w)$ whichever of v'' and w'' that has the smallest depth in A.

Proof of correctness (algorithm 5).
Correctness of (1): Let v be a node in A, all ascendants to v in C are then also ascendants to v in A by construction of C. This directly gives the correctness of the return statement. Assuming $u \neq v$ and $u \neq w$, then u has to be the apex of the heavy path containing $NCA_A(v, w)$. Per definition of C, all nodes except leafs, are guarantied to be the apex of a heavy path in A. Furthermore the definition gives that the path $v \to root$ from C is $v \to apex(p_A(v)) \to apex(p_A(apex(p_A(v)))) \to \cdots \to root$. This implies that the path from C contains all nodes that are the apex of a heavy path and resides on the path $v \to root$ in A.

Assume $NCA_A(v, w)$ resides on a heavy path where the apex has a higher depth than u. Let a be the apex of this path. a will then reside on the path $v \to root$ and the path $w \to root$ in C. a is thus a common ancestor for v and w in C. This contradicts the fact that u is $NCA_C(v, w)$, as a resides deeper than u in A, which implies that a resides deeper than u in C. Hence u must be the apex of the heavy path containing $NCA_A(v, w)$.

Correctness of (2) and (3): Let hp_u denote the heavy path containing u. To find $NCA_A(v, w)$ the next step becomes to find the nodes, from where the paths $v \to root$ and $w \to root$ intersects with hp_u. If $apex(v') = u$ it is known that v' is on hp_u and thus is not an apex itself. In this case v' has to be a leaf in C, which together with the fact that v' is on the path $v \to root$ implies that $v' = v$. This clearly gives that v' has to be the first node on $v \to root$ that intersect with hp_u.

If $apex(v') \neq u$ it is known that v' is not in hp_u which implies that none of v' descendants in A are in hp_u. Since the edge $v' \to u$ exists in C it is known that $p_A(v')$ is located in hp_u by the definition of the edges in C. Hence $p_A(v')$ in this case has to be the first node where $v \to root$ intersects with hp_u.

It is hereby shown that the first ascendant of v to reside in hp_u has to be either v' or $p_A(v')$. The proof for w is symmetric to the one of v.

Correctness of (4): The correctness of this step follows directly from (2) and (3). □

The only part of algorithm 5 which is not straight forward to solve in constant time is the depth problem in case (2) and (3). In [2] the depth problem on C is solved by dividing C into three plies. The nodes are divided into the three plies according to their sizes in C. The size of a node v in C depends on whether v is an apex or not. If v is an apex $size_C(v) = size_A(v)$ as all descendants to v in A will be a descendant to v in C. On the other hand if v is not an apex, it has to be a leaf in C implying $size_C(v) = 1$. By definition of heavy paths v can only be an apex if $2 \cdot size_A(v) \leq size_A(p_A(v))$. Since v is a descendant to $p_C(v)$ in A and $p_C(v)$ per definition is an apex, it is known that $2 \cdot size_C(v) \leq size_C(p_C(v))$. Let $rank(v) = \lfloor \log(size_C(v)) \rfloor$, then $rank(v) < rank(p_C(v))$ implying that no path in C contains two nodes of the same rank. As the size of the root is n this also gives that C has a depth of at most $\lfloor \log(n) \rfloor$.

Using the definition of rank, the nodes in C are divided into three plies in the following way [2, p. 344]:

Ply 3: Nodes where $rank \geq \lfloor \log(\log(n)) \rfloor$

Ply 2: Nodes where $\lfloor \log(\log(\log(n))) \rfloor \leq rank \leq \lfloor \log(\log(n)) - 1 \rfloor$

Ply 1: Nodes where $rank < \lfloor \log(\log(\log(n))) \rfloor$

The idea behind dividing C into plies, is that it is possible to solve the depth problem in constant time, if the node and the ancestor searched for resides inside the same ply.

To solve the depth problem within a ply it is utilized that C has a depth of at most $\lfloor \log(n) \rfloor$. In addition to this Lemma 2 from [2] is used.

Lemma 2. *[2, p. 345]*

1. *Ply three contains at most* $O(\frac{n}{\log(n)}$ *nodes.*
2. *Ply two contains at most* $O(\frac{n}{\log^2(n)})$ *nodes.*
3. *Each connected component of ply one is a subtree of* C *containing at most* $\log^2(n)$ *nodes.*

The depth problem in ply two and three are solved in constant time by constructing an array for each node. The array for a node contains pointers to all ancestors within the same ply, indexed by the depth of the ancestors. Since all nodes in C have at most $\log(n)$ ancestors, the total combined size of the arrays in ply three is $O(\frac{n}{\log(n)} \cdot \log(n)) = O(n)$. By construction of ply two there can be at most $\log^2(n)$ descendants with different ranks. This gives a total combined size of the arrays in ply two of $O(\frac{n}{\log^2(n)} \cdot \log^2(n) = O(n))$. As the largest part of C resides in ply one, ply one has to be represented in another fashion. This is in [2] solved by transforming the subtrees in ply one into binary trees. The transformation is done by letting a node v residing at depth i in a subtree be placed at depth $i \cdot \lceil \log^3(n) \rceil$ in the corresponding binary tree. Let $0 \leq i' \leq i$, then the binary tree is constructed such that the ancestor at depth i' in the subtree, is the ancestor of depth $i' \cdot \log^3(n)$ in the binary tree. This is possible as a node in ply one has a maximum of $\log^2(n)$ children, and the difference in depth for a parent and its children becomes $\log^3(n)$ in the binary tree.

By embedding the constructed binary trees in complete binary trees and provide them with an in-order numbering it is possible to solve the depth problem on the binary trees in constant time using algorithm 4. Even with the division of C into plies, the binary trees are still too large to represent explicitly within $O(n)$ space. It is however possible to represent the binary trees in a implicitly manner within $O(n)$ space, according to [2, p. 346].

It is now established that it is possible to solve the depth problem in constant time within a ply. For all nodes v in C, let $a(v)$ be the shallowest ancestor of v in the ply containing v. It is then possible to reduce a depth problem in C into a depth problem within a ply, by using algorithm 6.

Algorithm 6 Transforming the depth problem in C into a depth problem within a single ply. [2, p. 345]

Let v be a node in C, and d the depth of the ancestor searched for.
while $d \notin [d_C(a(v)), d_C(v)]$ **do**
  $v = p_C(a(v))$
end while
(v,d) now resides within the same ply and can be found in constant time.

The ply number is increased with at least one in each iteration of the while-loop, making the time for the transformation constant. It is know shown how to solve the depth problem on C in constant time using $O(n)$ space, and thus also how to solve the NCA problem on A in constant time using $O(n)$ space. This under the assumption that $NCA_C(v, w)$ can be found in constant time and $O(n)$ space.

Reduction from compressed trees to complete binary trees

In [3] the compressed tree C is transformed into a balanced binary tree B via a simple recursive procedure called *binarize*. Let v be a node in C and W be the set of children to v in C. Binarize is the defined as shown in algorithm 7.

Algorithm 7 Binarize(v,W) [2, p.347]

(1) Let $W = w_1, .., w_k$ and $s = \Sigma_{i=1}^{k} size_C(w_i)$. Let j be the minimum index such that $\Sigma_{i=1}^{j} size_C(w_i) \geq \frac{s}{2}$. If $j = k$, replace j by $k - 1$

(2) If $j = 1$, attach w_1 as the left child of v. Otherwise, let x_1 be a new red node. Attach x_1 as the left child of v and execute *binarize*(x_1, W_1), where $W_1 = w_1, \ldots, w_j$.

(3) If $j = k - 1$, attach w_k as the right child of v. Otherwise, let x_2 be a new red node. Attach x_2 as the right child of v and execute *binarize*(x_2, W_2), where $W_2 = w_j + 1, \ldots, w_k$.

The base case of the recursive algorithm is when $|W| \leq 2$. In this case the node and its children already forms a binary tree, and can therefore be inserted as it is. By calling the binarize procedure with the root in C as argument, the compressed tree is transformed into a balanced binary tree B. B contains two types of nodes. The green nodes are the nodes also contained in C, while the red nodes are the nodes added to transform C into a balanced binary tree. By construction it is clear that $NCA_C(v, w) = green(NCA_B(v, w))$, where *green(u)* is defined as the nearest green ancestor to u. By embedding B in a complete binary

tree B', and number the nodes in B according to an in-order traversal of B', it is possible to solve the nca depth problem and the depth problem in constant time. The numbering can be done using the procedure called *number* from [2]. By calling $Number(root, h, 2^h)$ all nodes will be assigned their in-order number.

Algorithm 8 Number(v,h,i) [2, p. 349]

v is a node in B, h is the height of v in B', and i is the in-order number of v in B'.

Step 1. Assign number i and height h to v

Step 2. If v has a left child w_1, execute $number(w_1, h-1, i-2^{h-1})$.

Step 3. If v has a right child w_2, execute $number(w_2, h-1, i+2^{h-1})$.

In [3] the nca depth problem on B is solved by using algorithm 3 and the in-order numbering from B', while the depth problem on B is solved using plies as it was the case for the compressed trees. It is however also possible to solve the depth problem on B by using algorithm 4 and the in-order numbering from B'. As *green(u)* can be found in constant time, by adding a pointer to each node in B, the NCA problem on C can be solved in constant time. Storing B within $O(n)$ space is not a problem since the depth of B is $O(\log(n))$ ([3, lemma 11, p. 348]) and the number of nodes thereby is $O(n)$.

2.3 Suffix Tray

2.3.1 Outline of the suffix tray data structure

The following subsections are based on the suffix tray data structure presented by Cole et al. [1]. The suffix tray data structure builds upon the ideas from the suffix tree presented in section 2.1 and the enhanced suffix array presented in section 2.2, hence the name suffix tray. By transforming a suffix tree into a new tree structure combined with an enhanced suffix array, a search complexity of $O(m+\log(|\Sigma|))$ is achieved. Suffix trays are constructed within an $O(n \cdot \log(|\Sigma|)$ time complexity and an $O(n)$ space complexity, making it a real alternative to suffix trees and suffix arrays.

2.3.2 The construction algorithm

To build a suffix tray over a text, it is necessary to divide the corresponding suffix tree into different node types. The different node types are defined in definition 4.

Definition 4. *[1, def. 2]*

σ-node: A σ-node is a node in a suffix tree, which subtree contains at least $|\Sigma|$ leafs.

Branching σ-node: A σ-node is called a branching σ-node if two or more of its children is a σ-node.

Non branching σ-node: A σ-node is called a non branching σ-node if exactly one of its children is a σ-node.

σ-leaf: A σ-node is called a σ-leaf if none of its children is a σ-node.

As a backbone the new tree structure contains all σ-nodes from the corresponding suffix tree. In addition to the σ-nodes, all children to branching or non branching σ-nodes, which are *not* a σ-node, are made into *suffix interval*-nodes and inserted as children in the new tree structure. A suffix interval-node is a node representing the leafs in the subtree of the given node. The interval that a suffix interval-node represents is thereby equivalent to a continuous part of the corresponding suffix array.

Representation of the data structure

Cole et al. [1] does not explain how the labels of the different edges should be defined. This is though easily defined as all edges between two nodes, each corresponding to a single node in the suffix tree, gets the same label as the corresponding edge in the suffix tree. The only case where a node in the suffix tray does not correspond to a single node in the suffix tree, is when multiple suffix intervals are represented as a single suffix interval. In this case the edge to the suffix interval does not contain any label.

A suffix interval-node contains the start and the end index of the interval represented. In some cases multiple suffix interval-nodes may be represented by a single suffix interval-node.

A σ-leaf is represented as a suffix interval node. An important difference between σ-leafs and suffix interval-nodes is that a σ-leaf never gets merged with other nodes.

A non branching σ-node contains three pointers. One pointer is to the child that is a σ-node. Let c be the first character on the path to the child that is a σ-node, also called the *separating character*. Then all children with an edge starting with a character $c_1 < c$ will be a suffix interval-node and the intervals represented will reside right after each other in the suffix array. It is therefore possible to represent these with one suffix interval-node. The same will be the case for all children where $c < c_1$. The two extra pointers in the non branching σ-node is pointers to these two suffix interval-nodes. If no edges has a starting character $c_1 < c$ or $c_1 > c$ the respective pointer becomes a null pointer.

A branching σ-node contains an array of size $|\Sigma|$, consisting of pointers to its children. The pointers are placed in the array according to the first character on the edge to the specific child. This is done by using the characters position in the used alphabet as index. For characters in the alphabet with no matching edge, a null pointer is inserted into the array.

The representation of the branching σ-nodes is slightly different than how it is done in [1]. When it is possible Cole et al. merges multiple suffix interval-nodes into one suffix

interval-node, and makes the relevant pointers point to the new node. This is a trade off between space usage and time used when searching for a substring. The complexities for both solutions are though the same. It is clear that a branching σ-node in both cases consumes $O(|\Sigma|)$ space. Since the number of extra suffix interval-nodes per branching σ-node is at most $O(|\Sigma|)$, and suffix interval-nodes are using constant space, the absence of contraction does not change the space complexity. The more specific division of suffix intervals and the possibility of labels on the edges gives a practical speed up on the search algorithm. Due to the possible size of suffix intervals created from σ-leafs the complexity of the search algorithm will however remain the same. As the focus for this thesis is to compare the search functions in the presented structures, the version preserving the suffix intervals is used in the experiment section.

To fully exploit the advantage gained by using arrays to store children pointers in branching σ-nodes, it is necessary to be able to convert a character into an index fast. Cole et al. [1] assumes this can be done in constant time, but in practice we have to make some assumptions to be able to do this. As a characters index has to be between 0 and $|\Sigma|$ it is necessary to reassign the indices every time an alphabet is used. To be able to convert a character into an index in constant time two assumptions are made.

(1) Only ASCII characters are allowed in the alphabets.

(2) The length of the text is equal or greater than the number of ASCII characters.

It is fair to assume that suffix trays are used in cases where (2) is met, since exact pattern techniques without preprocessing typically are used for small text sizes. In all cases used in this thesis the texts contains only ASCII characters making (1) a reasonable assumption. It is however possible to use another set of characters in (1) if (2) is changed slightly. How the dependency is between (2) and the set of characters in (1) will be clear after reading how indices are assigned to characters.

An *alphabet array* of size |ASCII| is created. It is then utilized that ASCII characters in programming languages also are represented as integers and that these integers are located in a continuous sequence. This is done by scanning through the text, and for each character use the number representation to index into the alphabet array. During the scan a counter is used to decide which number the character is in the alphabet. If the entry in the alphabet array is empty for the current character, the value of the counter is assigned to the entry, and thereafter increased. By using the alphabet array it is possible to convert a character to its index in constant time. The construction of the alphabet array is not affecting the construction complexity of the suffix tray as it is done in $O(n)$ time, and by (2) it is secured that it does not conflict with the space complexity either.

When converting a suffix tree into a suffix tray the order of the alphabet in the two structures should be the same, as the underlying suffix array otherwise would differ. A clean way to ensure this is to order the alphabets such that they obey classical lexicographical

order which also ease the life of the programmer. This can easily be done, by using the aforementioned technique. Every time a new character is found it is inserted into a list. When the full text has been scanned, the list is sorted and the entries of the alphabet array is updated. This takes an extra time of $O(|\Sigma| \cdot \log(|\Sigma|))$ which in practice is insignificant as $|T| >> |Sigma|$.

Another thing to be aware of, is that the NCA data structures used when building the enhanced suffix array in section 2.2, in many cases would be deleted to save space after the two auxiliary arrays have been established. When building the suffix tray the suffix array is divided into multiple smaller intervals. This leads to binary searches on smaller arrays, making new (L_M, M, R_M) triplets necessary. It is required to have the NCA data structures to recompute these new triplets. The midpoint of the triplets will however still be unique since the suffix intervals in the suffix tray are disjoint, making it possible to save the triplets in two auxiliary arrays like in previous cases.

Space and time complexity

In section 2.1 and section 2.2 it was shown how to build suffix trees and suffix arrays within a space complexity of $O(n)$. The nodes in the suffix tray are a subset of the nodes in the corresponding suffix tree, before some of the suffix interval-nodes may get merged. The number of nodes in the suffix tray is thereby bounded by the number of nodes in the corresponding suffix tree. Only the branching σ-nodes is not of constant size. The branching σ-nodes consist of an array of size $O(|\Sigma|)$, allowing the suffix tray to contain no more than $O(\frac{n}{|\Sigma|})$ branching σ-nodes, if a space complexity of $O(n)$ shall be reached. That this requirement is met and the space complexity for the suffix tray thereby is $O(n)$, can be realized through the following steps:

(1) The subtree of a σ-leaf in a suffix tree contains at least $|\Sigma|$ leafs.

(2) By definition no descendants of a σ-leaf can be a σ-node. This means the σ-leafs in the suffix tree have disjoint subtrees.

(3) (1) and (2) implies that a tree build from the σ-nodes has no more than $\frac{n}{|\Sigma|}$ σ-leafs.

(4) Collapsing edges through non branching σ-nodes in the tree from (3), creates a tree consisting of all branching σ-nodes and σ-leafs. As all branching σ-nodes have at least two children, there can be at most $\frac{n}{|\Sigma|} - 1$ branching σ-nodes.

Focusing on the time complexity, it is clear that the tree structure can be build within $O(n)$ time via a depth first traversal of the suffix tree. Likewise the two auxiliary arrays Rcp_M and Lcp_M in the enhanced suffix array can be updated in $O(n)$ time using the NCA-structures. Lastly using the alphabet array, the children pointers in the branching σ-nodes can be assigned to the right index in constant time. Thus the time complexity for the entire suffix tray structure given a suffix tree and a suffix array becomes $O(n)$.

2.3.3 The search algorithm

Searching in a suffix tray is done character by character. Standing in a branching σ-node, a possible matching edge can be found in constant time using the alphabet array. For non branching σ-nodes the matching edge can be found in constant time by a single character comparison with the separating character. Hence the worst case complexity is $O(m)$ if the search is finished before reaching a suffix interval-node.

If a suffix interval-node is reached the worst case time becomes $O(m + \log(|\Sigma|))$. The search through the suffix interval is done in the same manner as it was for the suffix arrays in section 2.2. The search algorithm in section 2.2 has a complexity of $O(m + \log(n))$ as a binary search in an array of size n is performed. A suffix interval in the suffix tray can be created in three different ways:

(1) From a node that is not a σ-node. In this case the node has less than $|\Sigma|$ leafs in its subtree implying that the size of the interval is less than $|\Sigma|$.

(2) A σ-leaf that is turned into a suffix interval. As a node can have at most $|\Sigma|$ children and a σ-leaf has no σ-nodes among its children, the total number of leafs in the subtree is less than $|\Sigma|^2$.

(3) When representing a non branching σ-node and its children, a maximum of $|\Sigma| - 1$ suffix intervals-nodes are merged into one suffix interval-node. None of the $|\Sigma| - 1$ suffix interval-nodes are created from a σ-node, hence they have to be created through (1). The total size of the composite interval is therefore less than $|\Sigma|^2$.

The size of a suffix interval in a suffix tray is thus $O(|\Sigma|^2)$, resulting in a time complexity of $O(m + \log(|\Sigma|^2) \Rightarrow O(m + \log(|\Sigma|)$ for the binary searches within a suffix interval. The time complexity for the entire search algorithm is thereby $O(m + \log(|\Sigma|)$. Let p be the part of the search string that is already matched when a suffix interval is reached. It is then known that all suffixes in the suffix interval has p as a prefix. A small speed up can thus be gained by starting the character matching from $p + 1$ when performing the binary search. The small speed up is however purely a practical speed up, and does not change the time complexity of the search algorithm.

With an extra cost of $O(\#occ)$ the indices of all occurrences can be returned. This is done by using the technique from the suffix array when the search ends within a suffix interval. Otherwise the traversal technique from the suffix tree can be used.

Chapter 3

Implementations

The implementations, raw data files, plots and the files used to create the report can be found at:

`https://bitbucket.org/jfogh/thesis/src`

All implementations are made in Java and compiled using Javac version 1.7.0_17.

The following command can be used from the **code/Thesis/src** directory to compile all relevant classes. The compiled files will reside in the bin folder.

javac CorrectnessModule.java SuffixTreeUkkonen/*.java SuffixTreeNodes/*.java SuffixTray/*.java SuffixArray/*.java SimpleScan/*.java Experiments/*.java -d ../bin

The following is a short overview of the relevant java classes and data folders.

code/Thesis/src/CorrectnessModule.java: Class containing methods to test the correctness of the suffix tree, array and tray structures. By calling the main method without parameters, the tests are performed on the files registered with their respective filenames in the file *texts/fileNames.txt*,

code/Thesis/src/impleScan/SimpleScan.java: Implementation of a simple search function that scans through a text from index 0 to n. This algorithm is only used in the correctness check of the suffix tray, array and tree. The signature of the constructor is *SimpleScan(String text)*. The signature of the search function is *int search(String searchString)*. The int returned is the index of the found occurrence, and -1 if no occurrence is found.

code/Thesis/src/SuffixTreeUkkonen/SuffixTree.java: Implementation of the suf-

fix tree from subsection 2.1. Ukkonen's algorithm is used to construct the suffix tree. The signature of the constructor is *SuffixTree(String text, ChildrenStructures childrenStructure)*, where ChildrenStructures is an enum in the class *NodeTypes.java* deciding how the children of a node is saved within the node. In all experiments in this thesis the enum *BINARY* is used, indicating that the used suffix tree nodes are of the type *SuffixTreeNodeBinary*. The signature of the search function is *int search(String searchString)*. The int returned is the index of the found occurrence, and -1 if no occurrence is found.

code/Thesis/src/SuffixTreeNodes/SuffixTreeNodeBinary.java: Is a suffix tree node where the pointers to the children are saved in a *TreeMap*.

code/Thesis/src/SuffixArray/SuffixArray.java: Implementation of a suffix array. The implementation is as the described suffix array in subsection 2.2, with the difference that the lcp problem is solved via a simple character by character comparison instead of a reduction to the NCA problem, as it was the case in subsection 2.2. This solution has been chosen as the actual construction time is not the focus of this thesis. The constructor does not take any parameters, but the actual suffix array is first build, when the method with the signature *construct_SuffixArray_from_SuffixTree(SuffixTree suffix_tree)* is called with the corresponding suffix tree. The signature of the search function is *int search(String searchString)*. The int returned is the index of the found occurrence, and -1 if no occurrence is found.

code/Thesis/src/SuffixTray/SuffixTray.java: Implementation of the suffix tray from subsection 2.3. The signature of the search function is *int search(String searchString)*. The int returned is the index of the found occurrence, and -1 if no occurrence is found. The signature of the constructor is *SuffixTray(SuffixTree suffix_tree, SuffixArray suffix_array))*, where the parameters are the corresponding suffix tree and suffix array.

code/Thesis/src/SuffixTray/SuffixTrayBranchingNode.java: Branching σ-node implemented as described in subsection 2.3.

code/Thesis/src/SuffixTray/SuffixTrayNoneBranchingNode.java: Non Branching σ-node implemented as described in subsection 2.3.

code/Thesis/src/SuffixTray/SuffixTrayInterval.java: Suffix interval-node implemented as described in subsection 2.3.

code/Thesis/src/Experiments/: The classes *WorstCaseSuffixTreeExperiments.java*, *WorstCaseSuffixTrayExperiments.java*, *WorstCaseSuffixArrayExperiments.java* and *CompareStructuresExperiments.java* contains the experiments performed in this thesis. All four classes have a main method which run the experiments from the thesis, if no parameters are parsed. In addition to the experiment classes, the folder contains different utility classes used in the post processing of the data.

report: Contains all latex files to create the report.

texts: Contains the texts used in subsection 4.3 as well as the texts used in the correctness module.
The texts *AV1611Bible_2pow21.txt* and *AV1611Bible_2pow21_lowerCase.txt* corresponds to Bib and BibLow in subsection 4.3, while *genome1.fa* corresponds to Sgen.

data: Contains all data used in the thesis as well as the plots and the gnuplot files to create the plots.
Data files with a name containing *Sorted* are sorted after the varying variable.
Data files with a name containing *Exp12Removed* are files where the two first runs have been removed.
Data files with a name containing *avgANDstdDev* are files containing average values and relative standard deviations.

3.0.4 Correctness

The correctness of the implementations was tested using the CorrectnessModule before any experiments were performed. The structures were here tested on multiple texts, using both random substrings from the texts as well as strings not residing in the texts. Furthermore it was tested that the structures complied with different invariants. The invariants tested were:

Suffix tree: No more than $n - 1$ inner nodes.

Suffix tree: Exactly n leaves.

Suffix array: All suffixes have to be in lexicographical order.

Suffix tray: No suffix intervals can be larger than $|\Sigma|^2$.

Suffix tray: The tree structure must not contain more than $\frac{n}{|\Sigma|}$ branching σ-nodes.

In addition to the random tests, corner cases and their respective results have been controlled manually. Last but not least, it was tested that the results of the search functions were as expected during all experiments. No tests showed bugs in the implementations.

Chapter 4

Experiments

4.1 Experimental setup

All experiments in this thesis are performed using a computer with an Intel i7-3630QM CPU. The CPU has the following specifications:

4 cores running at 2.4GHz, with a max turbo frequency of 3.4 GHz.

6MB L3 cache, shared between all four cores.

1MB L2 cache, 256KB dedicated to each core.

256KB L1 cache, 64KB dedicated to each core divided into a 32KB data cache and a 32KB instruction cache.

The size of the main memory is 8GB, while the used operating system is *Windows 8*. All experiments are run using the runtime flag - *Xmx1024m* to increase the heap and thereby allow larger structures. Furthermore the runtime flag - *Xss90m* is used to increase the stack size, in all experiments involving suffix trays. This is necessary as the implemented suffix tray among other things is build through a recursive depth first traversal of the corresponding suffix tree. Gnuplot version 4.6 are used to generate all graphs in the thesis, and in a single case the build-in *fit*[1] function is used in the post-processing of the data.

All experiments are measured using *System.currentTimeMillis()*, implying that all running times are measured as wall-time. In the specific setups for the individual experiments an iteration number is defined. This number tells how many queries each running time is consisting of. This is done, as it is necessary to perform multiple queries to reach a measurable running time. It should here be emphasized that all work not part of a query

[1]Documentation for the fit function can be found on page 63 at `http://www.gnuplot.info/docs_4.6/gnuplot.pdf`

is done off the clock. This also applies possible interleaving queries made to influence the cache patterns.

Each experiment is performed seven times, where the first two runs are performed to warm-up the cache. The values of the two first runs where usually only slightly higher that the values of the last five runs, but were still omitted for the sake of good practice. The running times used in the data analysis is the average values of the last five runs. This is done to even out possible external influences. To be sure that the average was a good representation of the five measured values, the relative standard deviations were monitored.

Common for all experiments is that different combinations of m and the number of iterations have been tested, to find combinations where the running times becomes high enough to make the relative standard deviations for each individual point $0 - 5\%$. This is achieved with the used values, without the experiments takes infeasible long time. By doing this it is secured that the average values are good representatives.

4.2 Experiments to Determine the Worst Case Complexities

When performing pattern matching the worst case complexity of the search function can depend on multiple variables. In this section the following four variables will be tested.

The index of the first occurrence (i).

The length of the query string (m).

The size of the input text (n).

The size of the alphabet ($|\Sigma|$).

The experiments are performed to establish in which degree the implementations used in this thesis are consistent with the theory. The primary goal for the experiments is to investigated whether the worst case running times for the implementations used in this thesis are coherent with the theoretical worst case complexities. The secondary goal for the experiments is to approximate the implementation specific constants in the worst case complexities.

4.2.1 Suffix Tree

In this section it is tested whether the worst case running time of the search function in the implemented suffix tree follows the theory. This is done by testing which of the four variable the worst case running time depends on.

Common for all experiments is that the worst case running time depends on the content of the text. All texts in this section is thus designed to create a path of length m where all

nodes have the maximum number of children. This path is designed to consist purely of the first character in the applied alphabet. In the implemented suffix tree the children of a node are saved in a binary search tree. Thus to construct a worst case query for the suffix tree, the traversal of the binary trees should go all the way to the leafs. If the binary trees are perfectly balanced, the smallest element will reside in a leaf in the lower left corner of the tree. It is with this reasoning that the path is constructed to consist purely of the first character in the applied alphabet. The implementation in this thesis however uses the build in data structure *TreeMap* from the *java.util* package to contain the children pointers. The TreeMap is an implementation of a Red-Black search tree which is a balanced binary tree that guarantees $O(\log(n))$ operations. It is though not guaranteed to be fully balanced, which implies that the smallest element in some cases may reside slightly higher than the leafs in the tree.

Experiment 1 [Dependency of n]:
In this experiment it is tested whether the worst case running time of the search function in the implemented suffix tree, depends on the size of n. This is done by making n the only varying variable. According to the theory the worst case running time should be independent of n.

Setup
#Iterations: 75.00.000.
Query: String of length $m = 100$ consisting of the character A.
Alphabet size ($|\Sigma|$): 2 (A-B).
Index of first occurrence: 0.
Text: The text is a string of size $n = 10.000, 20.000, \ldots, 200.000$. The content of the string is $A^m B^{n-m}$.

The first $m+1$ characters in the text string, creates the needed path of m fully exploited nodes. The rest of the content does not have an effect on the running time, and is therefore chosen to be pure B's.

Results
In figure 4.1a the running times for the different sizes of n has been plotted. The graph illustrates running times that are close to constant, with no real outliers. It is therefore fair to conclude that the running times are consistent with the theory, and the search function in the implemented suffix tree is independent of n.

Experiment 2 [Dependency of $Q's$ position in T]:
In this experiment it is tested whether the worst case running time of the search function in the implemented suffix tree, depends on the index i of the first occurrence of Q in T. This is done by making i the only varying variable. According to the theory the worst case

running time should be independent of i. The only difference in the setup compared to experiment 1 is how the text string is created.

Setup
#Iterations: 75.00.000.
Query: String of length $m = 100$ consisting of the character A.
Alphabet size ($|\Sigma|$*)*: 2 (A-B).
Index of first occurrence: The first occurrence is at index $i = 0, 10.000, \ldots, 190.000$.
Text: The text is a string with a fixed size of $n = 200.000$. The content of the string is $B^i A^m B^{n-m-i}$.

The characters at position i to $i + m$ in the text, creates the needed path of m fully exploited nodes. The rest of the content does not have an effect on the running time, and is therefore chosen to be pure B's.

Results
The results of this experiment is presented in figure 4.1b. The graph is close to constant with only a few outliers. In general the relative standard deviations are between $0 - 1\%$ for the individual points, making the average value a good representation. A closer look into the deviating points however shows that the relative standard deviations here goes from $2 - 9\%$. From the data it is clear that the high deviations are caused by single measurements that deviates significantly from the rest, and thereby have a relatively large impact on the average values. This indicates that the deviations can be due to external influences. To further investigate this, a re-run was made for the points deviating the most. The results of the re-run confirms that the deviations most likely are caused by external influences, making it fair to conclude that the worst case complexity of the implemented search function is independent of i.

Experiment 3 [Dependency of $|\Sigma|$ and m]:
In experiment 3 multiple test series are performed, to determine how the search function depends on m and $|\Sigma|$. It is expected that the worst case running time will depend on both $|\Sigma|$ and m, as the theoretical complexity is $O(m \cdot \log(|\Sigma|))$. It has already been established that the implemented search function follows the theoretical complexity, by being independent of both the index of the first occurrence and the length of the text. It is therefore not necessary to make attention to these two variables in experiment 3. The following setup has been run with alphabet sizes of $|\Sigma| = 2, 4, 8, 16, 32, 64$ (0-o in ASCII table).

Setup
#Iterations: 2.500.
Query: String of length $m = 5.000, 10.000, \ldots, 95.000$ consisting of the character 0.
Alphabet size ($|\Sigma|$*)*: Constant (2,4,8,16,32,64 (0-o in ASCII table)).

Text:The text consists of two parts. Given an alphabet A, B, C, D the structure of part one is $A^{95.000}BA^{95.000}CA^{95.000}D$, while the second part is 100.000 random characters from $\Sigma \setminus A$. Part two is appended to part one.

The idea behind the first part of the text, is to create a path of length 95000 where all nodes on the path have the maximum number of children. The path for A^{95000} is such a path, as all strings $A^i c$, where $c \in \Sigma$ and $0 \leq i \leq 95000$ is a substring of the constructed text. Between each search for the query string, five random searches are performed. The idea behind the random searches is to shuffle the cache, to make a possible constant for the worst case complexity more practical relevant. The random searches are made within part two as close to all substrings in part one has a long prefix in common with the query string.

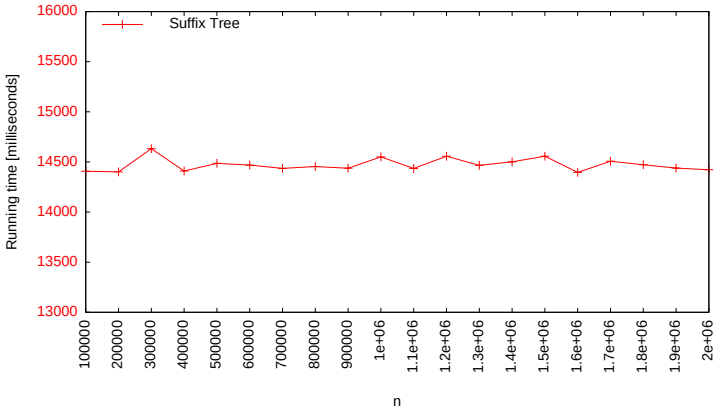
Results

The results of the experiments are plotted in figure 4.1c. The fact that all the graphs are ascending indicates that the worst case running time depends on the size of m. At the same time there is a clear tendency towards higher running times for growing sizes of $|\Sigma|$ implying a dependency of $|\Sigma|$. The only exception of this is the running times for $|\Sigma| = 8$ that are higher than those of $|\Sigma| = 16$ for small sizes of m. The running times for small sizes of m is however also the most volatile as these are very small, making external influences more significant. When calculating the average times for each point, this showed with slightly higher relative standard deviations for small sizes of m.

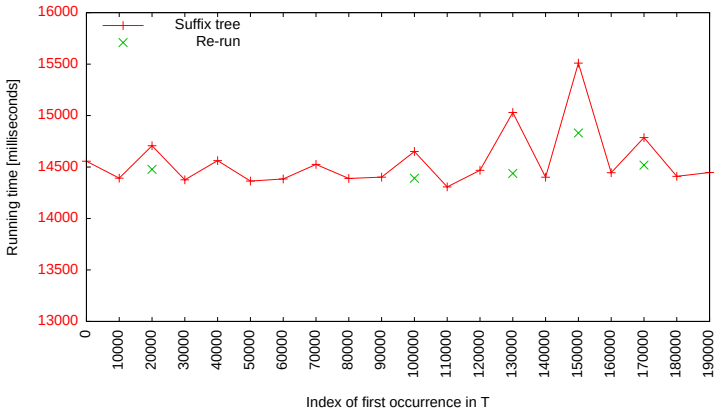
To test whether the dependencies are consistent with the theoretical complexity of $O(m \cdot \log(|\Sigma|))$ the running times have been divided by $m \cdot \log(|\Sigma|)$. The results are plotted in figure 4.1d. As expected the graphs becomes close to constant as m grows, and the external influences becomes insignificant. In addition to this the graphs are close to the same constant for all sizes of Σ confirming the complexity of $O(m \cdot \log(|\Sigma|))$. With this complexity it is expected that the worst case running time for $|\Sigma| > 2$ are a factor of $\log(|\Sigma|)$ larger than those for $|\Sigma| = 2$. This is illustrated in figure 4.1e and 4.1f where the running times has been divided by the running times for $|\Sigma| = 2$. Common for the five sizes of Σ is that the points resides close to the expected constants. For some sizes of $|\Sigma|$ the points resides slightly higher, while the points reside on the line or a bit under the line for other sizes of $|\Sigma|$. This most probably is an effect of the uncertainty about the depth of the smallest element in the TreeMap, which as described earlier is not guaranteed to be exactly $\log(|\Sigma|)$.

Having established that the practical worst case complexity corresponds to the theory, the next step becomes to decide the constant for the specific implementation. To reach a constant that approximates the worst case running time, an average constant is read from figure 4.1d. From the point where the graphs becomes close to constant a fair approximation of the average is $0,120$. It is here important to remember that each plotted time represents 2500 iterations, which means it is necessary to divide the constant with 2.500. This gives a

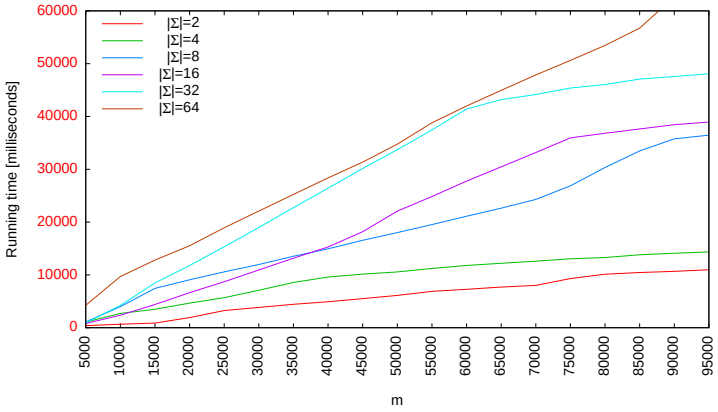
constant of $0,12/2.500 = 4,8 \cdot 10^{-5}$ and a worst case running time of $4,8 \cdot 10^{-5} \cdot m \cdot \log(|\Sigma|)$.



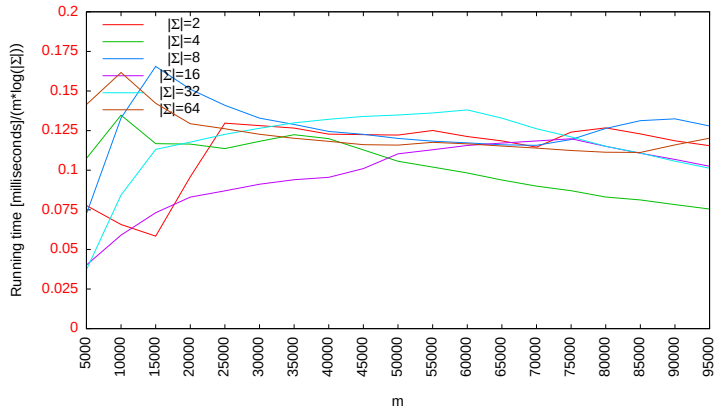
(a) Experiment 1: Running times for varying n.



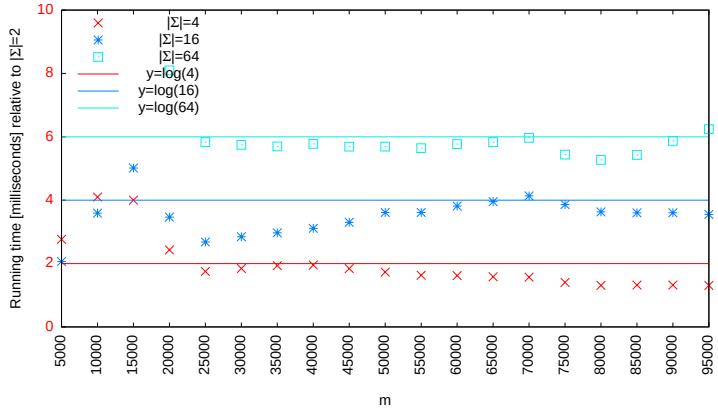
(b) Experiment 2: Running times for varying positions of the first occurrence in T.



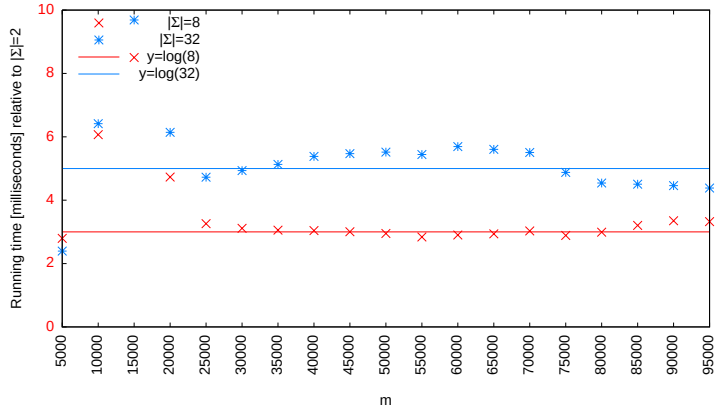
(c) Experiment 3: Running times for different sizes of $|\Sigma|$ and m.



(d) Experiment 3: Running times divided by $m \cdot \log(|\Sigma|)$.



(e) Experiment 3: Running times for $|\Sigma| = 4, 16, 64$ divided by the running time for $|\Sigma| = 2$.



(f) Experiment 3: Running times for $|\Sigma| = 8, 32$ divided by the running time for $|\Sigma| = 2$.

Figure 4.1: Results of experiments performed to determine the worst case complexity of the search function in the implemented suffix tree.

4.2.2 Suffix Array

In this section it is tested whether the worst case running time of the search function in the implemented suffix array follows the theory.

Common for all experiments is that the worst case running time depends on the content of the text. To reach a worst case running time, it is necessary to force the binary search to proceed all the way down to intervals of length one. All texts in this section is thus designed such that it contains the used query strings exactly once, and this in a way such that the suffix that has the query string as a prefix, resides at index one in the suffix array. This is achieved by making the query string consist only of the first character in the used alphabet. The text then contains the query string once, and characters different from the first character on the remaining positions. As the suffix array implementation is build from the implemented suffix tree, where \$ is appended to the text, the form of the text and query ensures a worst case running time.

Experiment 4 [Dependency of $|\Sigma|$]:
In this experiment it is tested whether the worst case running time of the search function in the implemented suffix arrays, depends on the size of $|\Sigma|$. This is done by making $|\Sigma|$ the only varying variable. According to the theory the worst case running time should be independent of $|\Sigma|$.

Setup
#Iterations: 75.000.
Query: String of length $m = 20.000$ consisting of the character 0.
Alphabet size ($|\Sigma|$): $|\Sigma| = 2, 4, 6, .., 64$ (0-o).
Index of first occurrence: 0.
Text: The text is a string with a fixed size of 200.000. The first m characters are 0 succeeded by $|\Sigma| - 1$ characters, where all characters in the alphabet but 0 is represented. The rest of the text is random chosen characters from $\Sigma \setminus 0$.

Results
The results for the experiment is plotted in figure 4.2a. As the graph is very close to constant, it is fair to conclude that the search function in the implemented suffix array follows the theory by being independent of $|Sigma|$.

Experiment 5 [Dependency of $Q's$ position in T]:
In this experiment it is tested whether the worst case running time of the search function depends on the index i of the first occurrence of Q in T. This is done by making i the only varying variable. According to the theory the worst case running time should be independent of i.

Setup
#Iterations: 75.000.
Query: String of length $m = 20.000$ consisting of the character A.
Alphabet size ($|\Sigma|$*)*: 2 (A-B).
Index of first occurrence: The first occurrence is at index $i = 0, 10000, \ldots, 170000$.
Text: The text is a string with a fixed size of $n = 200000$. The text string is on the form $B^i A^m B^{n-m-i}$.

The largest value of i is 170000, as it is necessary to have the character B after the query string, to ensure that the suffix that have the query string as prefix, resides at index one in the suffix array.

Results
The results of the experiment is plotted in figure 4.2b. As the graph is very close to constant, it is fair to conclude that the search function in the implemented suffix array follows the theory by being independent of i.

Experiment 6 [Dependency of m and n:
The results of experiment 4 and 5 has been as the theory suggests. What is left to show is how the worst case running time is depending on m and n. To test this, the running times for different combinations of m and n have been measured. The expected result is the theoretical worst case complexity of $O(m + \log(n))$ for suffix arrays. The following setup has been run with $n = 2^8, 2^9, \ldots, 2^{21}$.

Setup
#Iterations: 10000000.
Query: String of length $m = 5, 10, \ldots, 200$ consisting of the character A.
Alphabet size ($|\Sigma|$*)*: 4 (A-D).
Text:The text is a string of constant size (n), on the form $A^m B^{n-m-2} CD$.

The size of n has been chosen to be powers of 2, as it from the theory is expected that the worst case running time has a logarithmic dependency of n. The size of m has been chosen such that the contribution from the binary search is big enough to be distinguish from random fluctuations, and can be seen when plotting all runs in a graph. The text is build to test the worst case complexity, and to approximate the constants. The C and D at the end of the text, enables a search for C between every search for the query string. C will reside at the second to last index in the suffix array, ensuring that new parts of the suffix array and text will be loaded into the cache, and thereby give a higher and more practical constant to the worst case complexity. In addition to this the text string is designed to create as many single character comparisons as possible. This is done by

forcing as many mismatches as possible. A mismatch is performed each time case (2) in the search algorithm is performed, without the full query string is matched. Hence it is needed to proof that case (2) is used in all iterations. L will always be index 0, as the search for index 1 always proceeds in the left half of the interval. Index 0 contains the string \$ which is a unique character not used in the alphabet. This gives that $Lcp_M[M] = 0 = l$ in all iterations. In iterations where $l >= r$ the algorithm compares l with $Lcp_M[M]$, and thus it ends in case (2). If $r > l$ the algorithm compares r with $Rcp_M[M]$. As $r > 0$ it is known that $S_{SA[R]}$ starts with an A. In the text string the last A is followed by an B, making the suffixes containing A's sorted in the suffix array from index 1 to m where the suffix at position $1 \leq k \leq m$ is $A^{m-(k-1)}B^{n-m-2}CD\$$. Hence for all $1 \leq p, q < k$ it applies that $|lcp(S_{SA[p]}, S_{SA[k]}) = m - (k-1) = lcp(S_{SA[q]}, S_{SA[k]})|$, and more specifically that $|lcp(S_{SA[1]}, S_{SA[M]})| = m - (M-1) = |lcp(S_{SA[R]}, S_{SA[M]})|$ implying that case (2) is used.

Results

The results from the experiments with different sizes of n is plotted in 4.2c. It is hard to differentiate the graphs from each other but common for the graphs are, that they all seems to be increasing with the size of m. In addition to this they all seems to be linear with a gradient close to each other. If this is the case, the worst case running time will have an linear dependency of m that is independent of the size of n. To get a clear view of this the graphs have been fitted to a linear function on the form $a \cdot x + b$ using the *fit* function in gnuplot. The fit function uses a nonlinear least-squares algorithm to fit the functions to the measured points. At the end of a fit, the algorithm returns the standard deviation of the fit, also called the root-mean-square deviation [2]. For all functions the standard deviation were between $35 - 70$ with the exception of $n = 2^{21}$ where the standard deviation was 123. As all the times measured are between $1200 - 10000$ this indicates that the found functions are good fits.

The fact that it is possible to fit linear functions to the experiments, implies that the worst case running time has a linear dependency of m. The fitted function are plotted in figure 4.2d. Even though it is still difficult to distinguish the graphs, it is now possible to see that the graphs does not cross each other, and that the constants in the linear functions is increasing with the size of n. To investigate the gradients they are plotted for increasing sizes of n in figure 4.2e. From the graph it is clear that the gradients are very close to constant implying that the worst case running times dependency of m is independent of n. The point deviating the most is $n = 2^{21}$, but it was also here the largest standard deviation was seen, when fitting the data to linear functions.

To inspect the dependency of n the constants from the fitted functions are used. In figure 4.2f the constants are plotted as a function of $\log(n)$. Before plotting the constants,

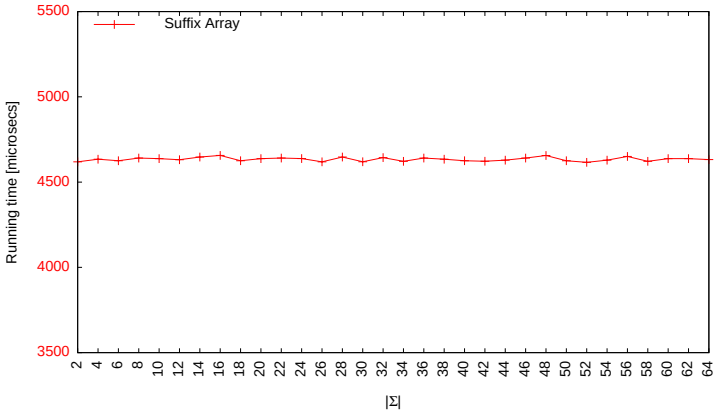
[2] A short explanation of root-mean-square deviation can be found at `http://en.wikipedia.org/wiki/Root-mean-square_deviation`

they have been divided by $\log(n)$ as the theory shows a linear dependency between the worst case running time and $\log(n)$. This means the values in the graph are the average time used for each iteration in the binary search. The graph is slightly descending until approximately $n = 2^{14}$ from where it is slightly ascending. It is expected that the graph is descending for low sizes of $\log(n)$, as the algorithm performs some constant work before the binary search is started. The algorithm compares the query string with the suffix on position 0 and $n - 1$ to establish l, r. These comparisons are due to the way the text is constructed independent of m, and thus only affecting the constants and not the gradients of the functions. This also explains way the slope is decreasing, as the number of binary iterations to "pay" for the constant work increases. The slightly ascending part can be explained by the sizes of the suffix arrays. When the suffix arrays becomes to large to reside in the cache, more work naturally have to be done in the binary search. A possibility to investigate this further is to use larger sizes of n than 2^{21}. This is however not possible with the used implementation, as this creates structures that are too large to have in memory. Instead some quick test runs were made, where the search for C between query searches was skipped. Skipping the search for C had the effect that the right half of the suffix array was not loaded into the cache. This made the constants increase at a later stage, supporting the cache theory.

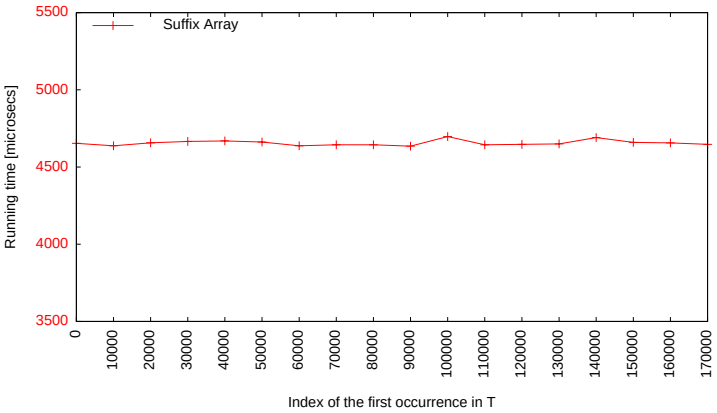
With reasonable explanations for the slopes in the graph, it is fair to conclude that the worst case running time is linear dependent of $\log(n)$. It is clear that the linear dependency of $\log(n)$ is independent of m, as it has already been shown that the dependency of m was independent of n.

The theoretical worst case complexity of $O(m+\log(n))$ has thereby now been confirmed in practical use. The constants in the worst case complexity for the search function in the used implementation can be found from figure 4.2e and 4.2f. According to the theory figure 4.2e should be constant, meaning that deviations from the average value, according to the theory, are random fluctuations caused by external influences. It is therefore reasonable to use the average from figure 4.2e as the constant of m. As the constant of $\log(n)$ the value of $\log(n) = 8$ in figure 4.2f is used $(142, 7)$. This is done with the reasoning that this is the highest value, where the caches are not a limiting factor. Constants for $\log(n) < 8$ would be expected to be larger, but as the experiment goes up to $m = 200$ the text size can not be a lower power of 2 than 8. It is possible to make tests with lower powers of 2, by making the max length of m smaller. This however at the cost of less possible m values, and thereby less data to fit a function to. With this in mind, an the fact that suffix arrays primarily are used for large sizes of n further experiments will be omitted.

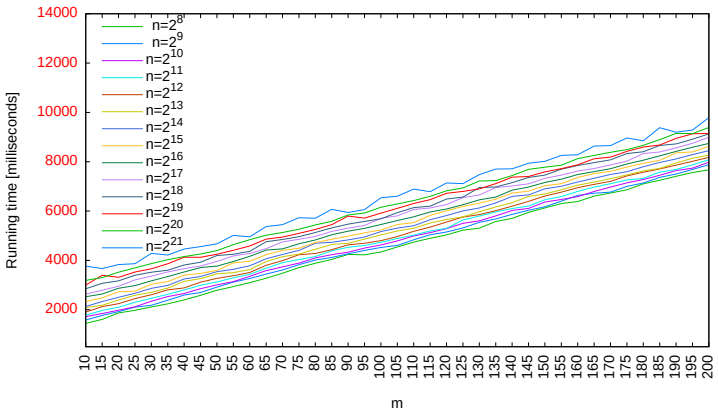
As it was the case for the suffix array, it is important to remember that each point represent multiple searches for the same string. In these experiments 10.000.000 iterations have been performed, which means the constants read from the graphs have to be divided by 10.000.000. The worst case running time thus becomes $3,28 \cdot 10^{-6} \cdot m + 1,43 \cdot 10^{-5} \cdot \log(n)$.



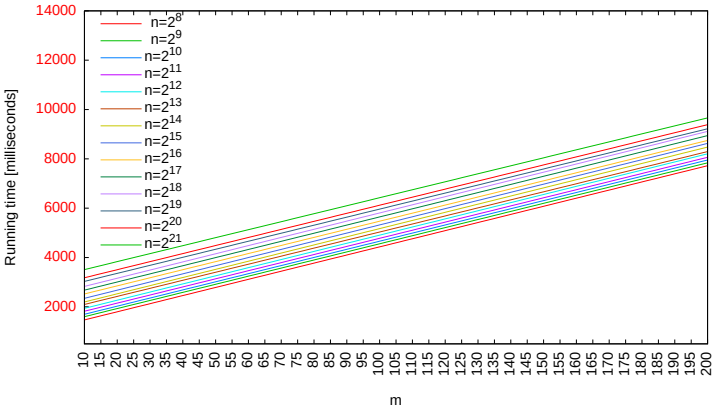
(a) Experiment 4: Running times for varying sizes of $|\Sigma|$.



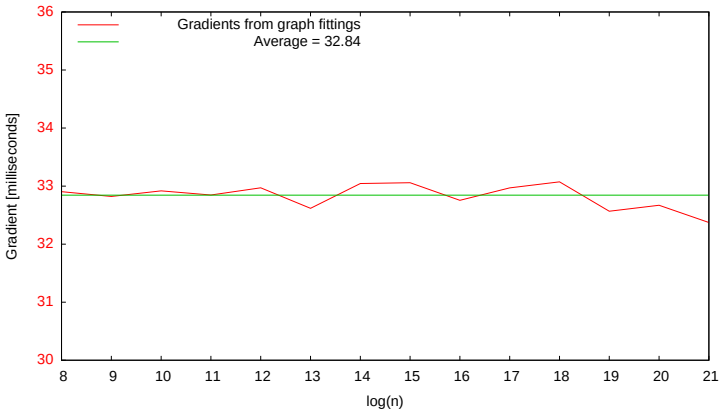
(b) Experiment 5: Running times for varying positions of the first occurrence in T.



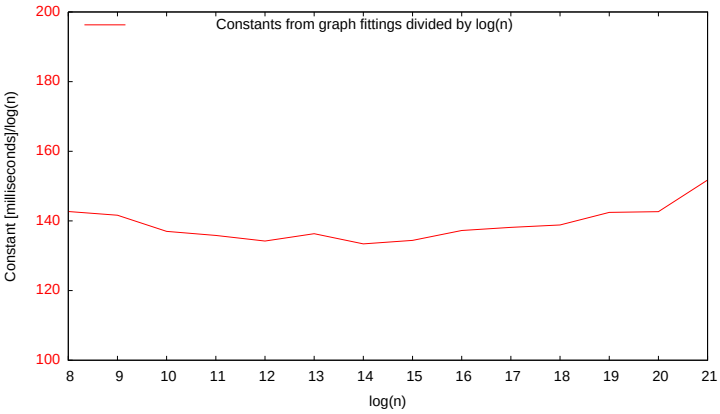
(c) Experiment 6: Running times for varying n and m.



(d) Experiment 6: Running times for varying n and m fitted to functions on the form $a \cdot x + b$ using the *fit* function in gnuplot.



(e) Experiment 6: Graph over the gradients from the linear functions in figure 4.2d.



(f) Experiment 6: Graph over the constants from the linear functions in figure 4.2d.

Figure 4.2: Results of experiments performed to determine the worst complexity of the search function in the implemented suffix array.

4.2.3 Suffix Tray

Looking at the suffix tray data structure, it consists of three different types of nodes. The branching σ-nodes, the non branching σ-nodes and the suffix interval-nodes. To find the worst case complexity for the full suffix tray structure it is therefore necessary to find the dependencies for all three kinds of nodes. As the code for the search function used in suffix intervals is identical to the code used for search in suffix arrays, the complexity of $O(m + \log(n))$ where n is the array size applies. Since the maximum size for an suffix interval is $|\Sigma|^2$ this gives an complexity of $O(m+\log(|\Sigma|^2)) \Rightarrow O(m \cdot (\log(|\Sigma|)+\log(|\Sigma|)) \Rightarrow O(m + \log(|\Sigma|))$. The missing part to establish that the implemented suffix tray follows the theory, is to show that a search purely in branching and non branching σ-nodes takes $O(m)$ time.

The following experiments are designed to test queries within respectively a string of branching σ-nodes and a string of non branching σ-nodes. The only difference in the experiments performed on branching σ-nodes and non branching σ-nodes is how the texts are constructed. In the experiment setups (B) will denote the text string used in experiments performed on branching σ-nodes, while (NB) will denote the text strings used in experiments performed on non branching σ-nodes. The default setup that is used within this section unless else is specified, is as follows:

Default Setup
#Iterations: 20.000.000.
Query: String of length $m = 50$, with the first character in the used alphabet on all positions.
Alphabet size ($|\Sigma|$): 2 (A-B).
Index of first occurrence: 0.
Text (B): The text is a string with a fixed size of $n = 200.000$. The initial part of the text is given an alphabet $0, 1, 2, 3$ on the form $((0^m10)(0^m11)(0^m12)(0^m13))$. The rest of the string is padded with $n - |\Sigma| \cdot (m + 2)$ random characters chosen from Σ.
Text (NB): The text is a string with a fixed size of $n = 200.000$. The initial part of the text is given an alphabet $0, 1, 2, 3$ on the form $0^(m + |\Sigma|)$, while the rest of the text is on the form $01230^{n-m-2\cdot|\Sigma|}$).

To be able to make a query purely within branching σ-nodes, it is necessary to have a path from the root, where the first m nodes are branching σ-nodes. The initial part of the text guarantees that the m nodes visited when searching for the query string are branching nodes. The rest of the text can be padded with random characters, as the running time does not depend on the content of the text when searching within branching σ-nodes.

To be able to search purely within non branching σ-nodes the text has to be created in another way. Here the first $m + |\Sigma|$ characters guarantees a path of m non branching σ-nodes, when searching for the query string. The running time is also unaffected by the

content of the text, when searching within non branching σ-nodes. It is however not possible to just insert random characters on the remaining positions, as this may cause non σ-branching nodes to turn into branching σ-nodes. The text is therefore always extended in the specified manner, as this ensures that all characters in the alphabet are used, without the risk of turning non branching σ-nodes into branching nσ-odes.

Experiment 7 [Dependency of $|\Sigma|$]:
In this experiment it is tested whether the worst case running time of the search function in the implemented suffix tray, depends on the size of $|\Sigma|$. This is done by making $|\Sigma|$ the only varying variable. According to the theory the worst case running time should be independent of $|\Sigma|$. The default setup has been used, with the following changes.

Experiment specific setup
Alphabet size ($|\Sigma|$*)*: $|\Sigma| = 2, 4, 6, .., 64$ (0-o).

Results
The results from the experiments are plotted in figure 4.3a. In general the results are as expected, as there are no clear dependency between the running time and the size of Σ. The graph for the branching σ-node experiment goes from $9696,0 - 10085,2$. This means the graph deviates with $\pm 194,6$ from $9890,6$, which is just $\pm 2,0\%$. It is fair to conclude, that these deviations are random fluctuations, as there are a relative standard deviation of $0-2,5\%$ on the individual points. The same calculations can be made for the graph for the non branching σ-node experiment which deviates with $\pm 1,5\%$. As the relative standard deviations on the individual points here are $0-3\%$ the same conclusions can be made. The worst case running time for the two types of structures are thus independent of $|\Sigma|$.

Experiment 8 [Dependency of $Q's$ position in T]:
In this experiment it is tested whether the worst case running time of the search function, in the implemented suffix tray, depends on the index i of the first occurrence of Q in T. This is done by making i the only varying variable. According to the theory the worst case running time should be independent of i. The default setup has been used, with the following changes.

Experiment specific setup
Index of first occurrence: The first occurrence is at index $i = 0, 10.000, \ldots, 190.000$.
Text: The texts are on the same form as the default text strings, though with the change that a character on position p is moved to position $(p + i) \mod n$.

Results
The results from the experiments are plotted in figure 4.3b. Like it was the case in exper-

iment 8, the fluctuations in the graphs are small enough to be explained by the deviations in the individual points. It can therefore as expected be concluded that the worst case running time is independent of i

Experiment 9 [Dependency of n]:
In this experiment it is tested whether the worst case running time of the search function in the implemented suffix tray, depends on the size of n. This is done by making n the only varying variable. According to the theory the worst case running time should be independent of n. The default setup has been used, with the following changes.

Experiment specific setup
Text (B): The text is a string of size $n = 100.000, 200.000, \ldots, 2.000.000$, with the default structure.
Text (NB): The text is a string of size $n = 10.000, 20.000, \ldots, 200.000$, with the default structure.

In this experiment there is a factor of 10 in difference between the sizes of n in the two texts. This is due to the way the suffix trays are implemented in this thesis. When building the tree structure in a suffix tray from a suffix tree, the suffix tree is traversed via a recursive method. By the construction of the text used to test non branching σ-structures, the depth of the suffix tree becomes close to n. Even though the experiments have been run with the flag *-Xss90m*, the largest possible size for n without getting a stack overflow error has been $n = 200000$. It is possible to implement the construction of the suffix tray in an iterative manner, but it has been assessed that the advantages gained from rewriting the algorithm are insignificant for the purposes in this thesis. The same problem does not occur for the text used to test branching σ-node structures, as the text here is padded with random characters resulting in a smaller depth of the suffix tree. The maximum size here is thus 2000000 as this is the limit without getting a heap error.

Results
The results from the experiment using the branching σ-node structure is plotted in figure 4.3c. Again a close to constant graph can be seen, indicating that the implementation follows the theory. Only the point for $n = 1500000$ is notable higher than the rest of the graph. This point however also have a slightly higher relative standard deviation that most of the other points, as the deviation here is 3.1%. In addition to this, there was no clear signs of higher running times for $n = 1500000$ than for the surrounding values, when the experiment was repeated a couple of times with the values $n = 1300000 - 1700000$. The results from the experiment using the structure with non branching node is plotted in figure 4.3d. Here there are no real notable difference in the running times, and it can thus be conclude that the implementation follows the theory, as the worst case running time theoretically is independent of n.

Experiment 10 [Dependency of m]:
In this experiment it is tested whether the worst case running time of the search function in the suffix trays, depends on the size of m. This is done by making m the only varying variable. According to the theory there should be a linear dependency between m and the worst case running time. The default setup has been used, with the following changes.

Experiment specific setup
#Iterations: 4.000.000.
Query string: String of length $m = 50, 100, \ldots, 1.500$, with the first character in the used alphabet on all positions.
Text strings: The same texts are used for all sizes of m. The used texts are build as if $m = 1.00$.

The number of iterations are decreased to 4.000.000 which is fifth of the default amount. This means the relative standard deviations for small sizes of m becomes slightly higher, but ensures the time used for running the entire experiment becomes feasible. The content of the texts can be kept constant, as the text build from $m = 1.500$ creates a path from the root of length 1.500 containing only branching / non branching σ-nodes, making it possible to perform all queries. The advantage gained by keeping the content constants, is that the experiment becomes faster to run, as the data structures only have to be build once for the entire experiment. Between each search for the query string, five random searches are performed. The idea behind the random searches, is to shuffle up the cache, to make a possible constant for the worst case complexity more practical relevant.

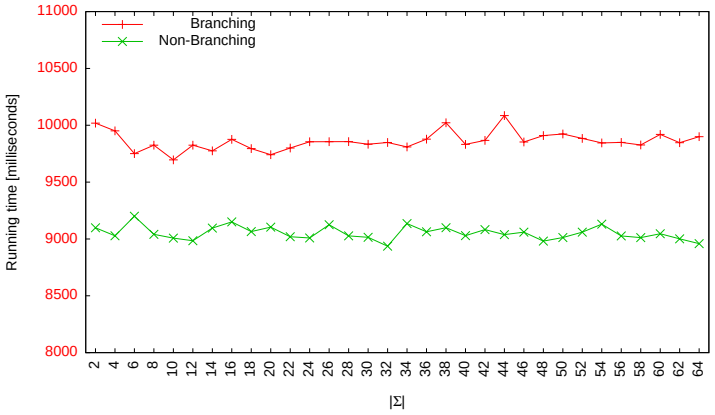
Results
The results from the experiments are plotted in figure 4.3e. The graphs looks linear, with only few small fluctuations and no real outliers. This indicates that the behavior of the running time for growing sizes of m is as expected. In figure 4.3f the actual running time is plotted relative to the worst case running time. The normalized values are slightly higher for low sizes of m where the measured times are more volatile, and the relative standard deviations for the points are higher. It is however clear that the normalized graphs becomes constant, as the size of m increases and the measured times becomes more stable. This verifies the expected linear dependency between m and the worst case running time, and thus the worst case running time for the implemented suffix tray follows the theory.

It has now been shown that the complexity of the search function in the implemented suffix tray, when searching within branching or non branching nodes, is $O(m)$. This can be combined with the already known fact, that the complexity for searching within suffix intervals in suffix trays is $O(m \cdot \log(|\Sigma|))$. By combining this knowledge it is known that the search function in the implemented suffix tray follows the theory, by having a worst case complexity of $O(m + \log(|\Sigma|))$. To find a good approximation of the constants in the

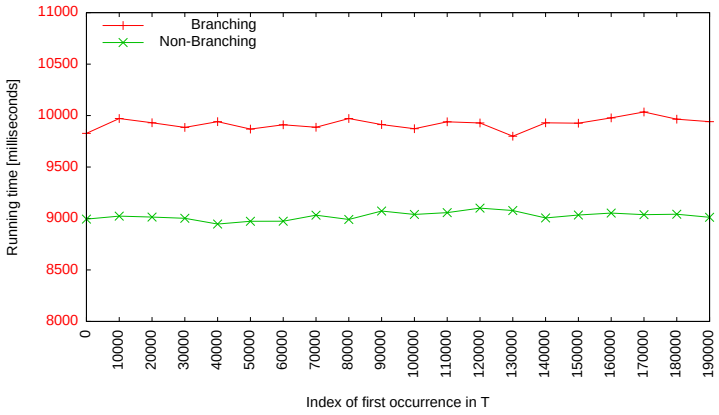
worst case complexity of the used implementation, it is necessary to know when it is most expensive to increase the size of m.

The constants when searching within branching or non branching σ-nodes can be approximated from figure 4.3f. A good approximation of the constants are the average values from the graphs, as the graphs are expected to be constant, and the deviations are considered to be random fluctuations. These values should be divided by the number of iterations, 4.000.000, and thus becomes $1,06 \cdot 10^{-5} \cdot m$ for the branching σ-node structure and $8,94 \cdot 10^{-6} \cdot m$ for the non σ-branching node structure. When looking at the constants for the suffix interval-node it is important to remember, that the complexity comes from $O(m \cdot \log(|\Sigma|^2)) \Rightarrow O(m + (\log(|\Sigma|) + \log(|\Sigma|)) \Rightarrow O(m + \log(|\Sigma|))$. The constant in the $\log(|\Sigma|)$ dependency should therefore be twice the size of the constant found in 4.2.2. As the constants found in 4.2.2 where $3,28 \cdot 10^{-6} \cdot m + 1,43 \cdot 10^{-5} \cdot \log(n)$, this gives the constants $3,28 \cdot 10^{-6} \cdot m + 2,86 \cdot 10^{-5} \cdot \log(|\Sigma|)$ for the suffix tray solution.

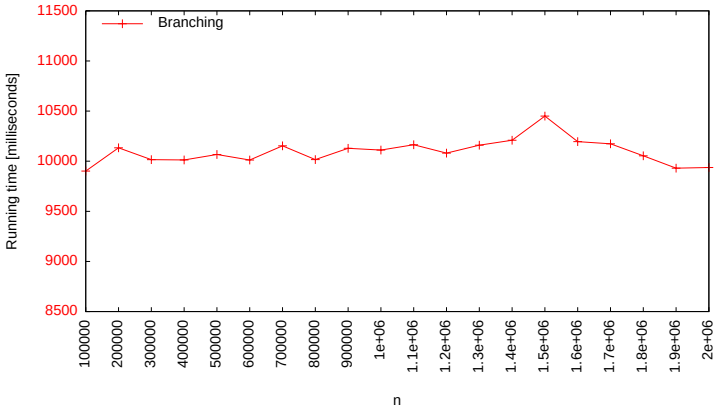
Having the constants for the different types of structures in the implemented suffix tray, it is possible to find the constants for the whole structure. It is clear that the branching σ-node structure has the highest worst case time for comparing characters. To get a good approximation of the upper bound for the entire suffix tray structure, a combination of the constants in the suffix interval-nodes and the branching σ-node structure is thus used. By combining these an upper bound of $1,06 \cdot 10^{-5} \cdot m + 2,86 \cdot 10^{-5} \cdot \log(|\Sigma|)$ is reached.



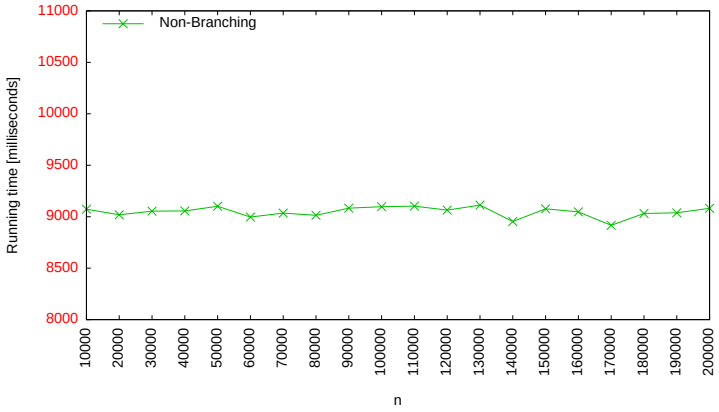
(a) Experiment 7: Running times for varying sizes of $|\Sigma|$.



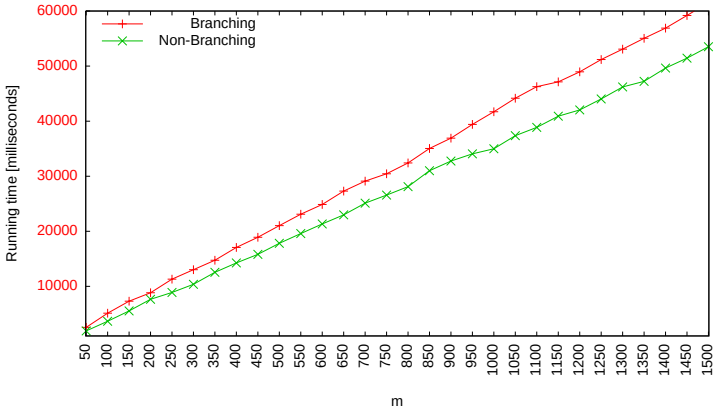
(b) Experiment 8: Running times for varying positions of the first occurrence in T.



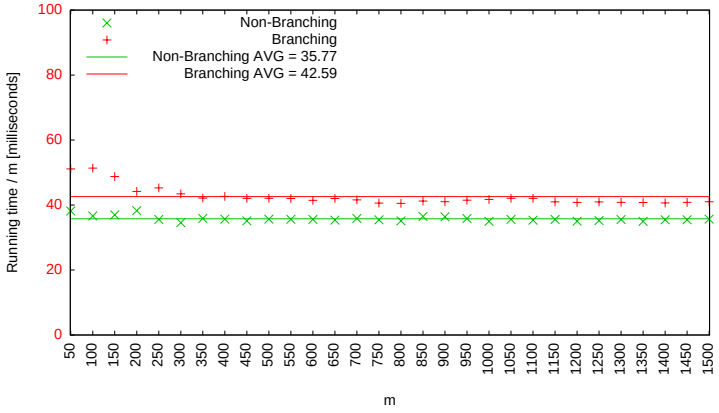
(c) Experiment 9: Running times for varying sizes of n (Branching σ-nodes).



(d) Experiment 9: Running times for varying sizes of n (Non Branching σ-nodes).



(e) Experiment 10: Running times for varying sizes of m.



(f) Experiment 10: Running times for varying sizes of m divided by m.

Figure 4.3: Results of experiments performed to determine the worst case complexity of the search function in the implemented suffix tray.

4.3 Experiments to test the practical performance

One thing is to decide the worst case complexities for the three data structures, as well as approximating the constants in the asymptotic times for the implemented versions, a different thing is how the data structures perform in practical use. To investigate this the three structures have been used for exact pattern matching on three different types of texts, all representing practical scenarios.

(1) A Staphylococcus genome (*Streptococcus pyogenes M1 GAS*) with an alphabet size of $|\Sigma| = 4$ (ACGT), and a length of $n = 1852441 \approx 2^{20.8}$. From here on denoted *Sgen*.

(2) The first $n = 2097152 = 2^{21}$ characters of the King James Bible with an alphabet size of $|\Sigma| = 80$. From here on denoted *Bib*.

(3) The text from (2), but with all letters transformed into lower case, resulting in an alphabet size of $|\Sigma| = 53$. From here on denoted *BibLow*.

Sgen is used to simulate the use of pattern matching within the field of bioinformatics, while Bib and BibLow simulates case sensitive and case insensitive searches in English texts.

Nine different texts are used to investigate how the search functions are affected of m,n and $|\Sigma|$ in practical use. The used texts are *Sgen*, *Sgen16*, *Sgen18*, *Bib*, *Bib16*, *Bib18*, *BibLow*, *BibLow16* and *BibLow18*, where 16 and 18 denotes that it is the first 2^{16} or 2^{18} characters from the original text. To test the search functions on the nine texts, two experiments are designed. The running times from these experiments will be denoted as the *actual* running times

The first experiment is designed to see how the search functions performs when the query string resides in the text. These queries will be denoted as *positive* queries. The experiment is run with all three data structures.

Experiment 11-a [Small sizes of m]:
#Iterations: 500000.
Query: Randomly chosen substrings from the text of length $m = 5, 10, \ldots, 100$.

Experiment 11-b [Small sizes of m]:
#Iterations: 250000.
Query: Randomly chosen substrings from the text of length $m = 100, 200, \ldots, 2000$.

Experiment 11-c [Large sizes of m]:
#Iterations: 62500.
Query: Randomly chosen substrings from the text of length $m = 4000, 8000, \ldots, 60000$.

The second experiment is designed to see how the search functions performs when the query string does not reside in the text. These queries will be denoted as *negative* queries. The experiment is only run using the search functions from the suffix tray and the suffix array The search function for the suffix tree is omitted, as the only difference between an query of length m residing in the text, and a query of length m where the first $m-1$ characters resides in the text from a algorithmic point of view is the return statement.

Experiment 12-a [Small sizes of m]:
#Iterations: 500000.
Query: Randomly chosen substrings from the text of length $m = 5, 10, \ldots, 100$, where the last character has been replaced with a random chosen character, such that the string no longer is a substring of the text.

Experiment 12-b [Small sizes of m]:
#Iterations: 250000.
Query: Randomly chosen substrings from the text of length $m = 100, 200, \ldots, 2000$, where the last character has been replaced with a random chosen character, such that the string no longer is a substring of the text.

Experiment 12-c [Large sizes of m]:
#Iterations: 62500.
Query: Randomly chosen substrings from the text of length $m = 4000, 8000, \ldots, 60000$, where the last character has been replaced with a random chosen character, such that the string no longer is a substring of the text.

Common for the two experiments is that they are divided into three parts. This is done to get a more fine grained graph for the small sizes of m, while still being able to get information about the larger sizes of m, without the experiment takes infeasible long time. To be able to make a fair comparison of the three structures, it is important that it is the same random chosen queries that are used for all three structures. At the same time it would not be fair to search for the same string three times in a row using the three different structures. The problem here is that the first used structure would risk to pay a higher price for loading the different parts of the text into the cache. To avoid this an array is used to save the start and end indices of each query for the current size of m. For experiment 12, the random chosen substitution characters are saved as well. By using the array it is possible to run the entire sequence of queries on all three structures independent of each other.

It is worth to notice that the relative standard deviations for the individual points in general are slightly higher for these experiments, than for those testing the worst case complexities. This is natural as there are a degree of randomness included in the running

times. The relative standard deviations are however still $0 - 5\%$ in close to all cases.

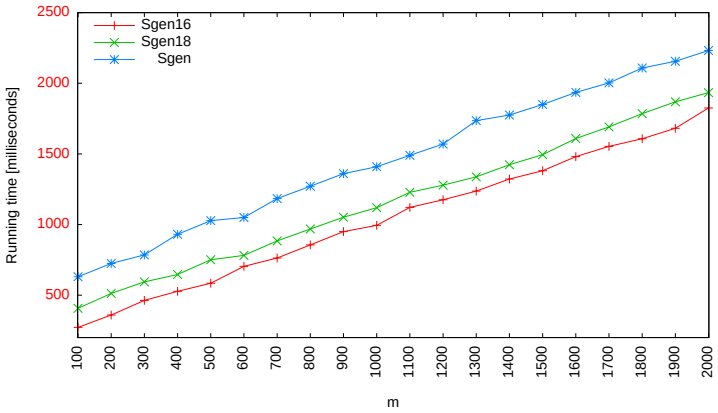
4.3.1 Suffix Tree

In subsection 4.2.1 it was demonstrated that the worst case complexity for the suffix tree is $O(m \cdot \log(|\Sigma|))$ and thus is independent of n. The results from Experiment 11-b however shows that this is not the case for the actual running time. The running times for Sgen, Sgen16 and Sgen18 are plotted in figure 4.4a. Here it is clear that the actual running time is increasing with the size of n. The results from the other types of texts and different sizes of m, shows the exact same pattern. When looking at the search algorithm for the suffix tree, this may not be so surprising. A larger text size gives a higher probability of suffixes with common prefixes, resulting in a higher number of nodes with many children, as well as shorter labels on the edges. The higher number of children makes the binary search for the matching edge longer, while the shorter labels makes the algorithm traverse a higher number of nodes, before the entire query string is matched. Hence both things affects the running time in a negative manner, explaining why the actual running time for the search function in suffix trees, are influenced by the text size. The difference in running time however looks quite stable, indicating that the penalty in running time already is applied during the search for the first 100 characters in the query string. Again this is not so surprising as the probability of multiple suffixes with common prefixes, decrease exponentially with the length of the needed common prefix.

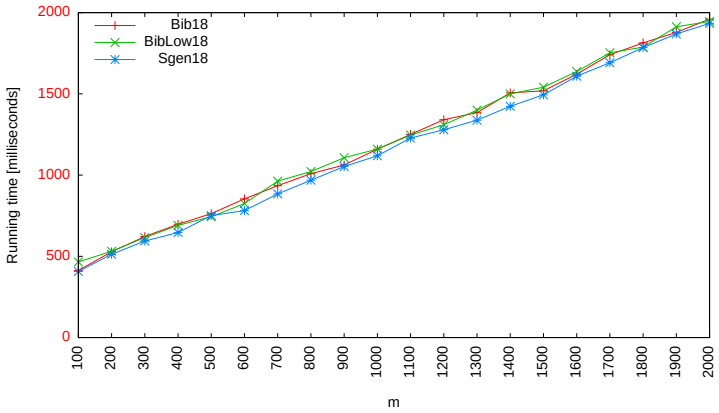
Comparing the results for Sgen18, Bib18 and BibLow18 it becomes clear that the size of $|\Sigma|$ has little influence on the running time in practical use. The running times from experiment 11-b are plotted in figure 4.4b. Even though the running times for Sgen18 is slightly lower than the once for Bib18 and BibLow18 it is not anywhere near the differences that was seen for the worst case running times. It should be noticed that the same picture was seen in the results for the two other text lengths and different sizes of m. The fact that the possibility of a higher number of children does not influence the running time, suggests that the increased running time for growing sizes of n primely is due to a higher number of nodes on the visited path resulting in shorter labels, and secondarily caused by a higher number of children to the nodes on the path of the query string. This is because the chance for multiple suffixes with a common prefix deviating at the exact same index is very low.

To get an idea of how the practical running time is relative to the approximated worst case running time found in subsection 4.2.1, the worst case running time has been plotted in figure 4.4c together with the running time for Sgen from experiment 11-b. Sgen has been chosen as this is the text with the highest actual running time, compared to its approximated worst case complexity. It is clear that as m increases, the worst case running time becomes a bad tool to describe the performance in practice. In figure 4.4d the practical running time has been plotted as a percentage of the worst case running time. Here the graph converge towards 5% for growing sizes of m, again indicating that the actual running

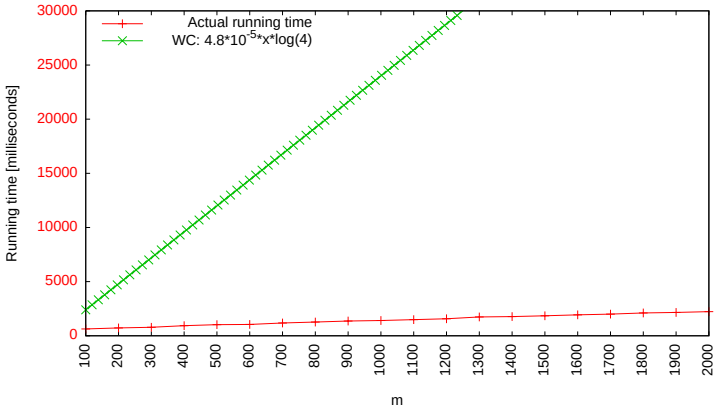
time will be dominated by the character comparisons on the edges, instead of the binary searches inside nodes to find the matching edges.



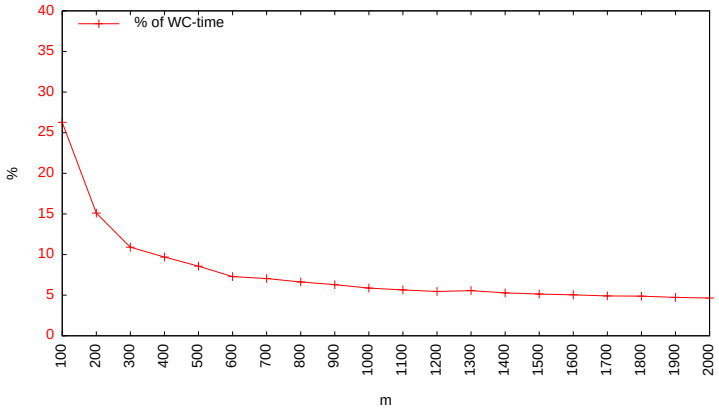
(a) Experiment 11-b: Running times for different sizes of n. (suffix trees)



(b) Experiment 11-b: Running times for different sizes of $|\Sigma|$. (suffix trees)



(c) Experiment 11-a: Running times for Sgen compared with the approximated worst case running time for suffix trees (subsection 4.2.1).



(d) Experiment 11-b: Running times for Sgen as percentage of the approximated worst case running time for suffix trees (subsection 4.2.1).

Figure 4.4: Graphs showing how the search algorithm in suffix trees performs in practical use.

4.3.2 Suffix Array

To see how the search function in the suffix array performs in practical use, it is necessary to look at both positive and negative queries. This is necessary because the search for a negative query always will result in a binary search all the way to an interval of size one, before it can be decided that it is a negative query. This though with the exception of the special case where the negative query string is either larger or smaller than all suffixes in the suffix array. A positive query on the other hand may be found in an earlier stage of the binary search. This means a positive query will have a faster average query time than a negative query in the cases where the length is the same, and the negative query is deviating at the last character.

The results for Sgen, Sgen16 and Sgen18 from experiment 11-b and 12-b are plotted in figure 4.5a and 4.5b to investigate whether this difference in running time is significant. Sgen,Sgen16 and Sgen18 are chosen, as the difference in running time are depended of n, since a larger n gives the possibility of a longer binary search. From the graphs it is clear that there are no significant difference between the running times for the positive and the negative queries regardless of the text size. The same picture was seen for the a and c parts of the experiments and for the two other types of texts.

The fact that the difference in the running times are insignificant are not so surprising, as previous results have shown that the majority of the running time is used on character comparisons. Furthermore the average depth of a binary search for a random positive query will be close to $\log(n)$ since the number of suffixes at each depth are increased with a factor of 2.

The analysis performed in the rest of this subsection will thus be on the data for the positive queries, but will apply to both negative and positive queries.

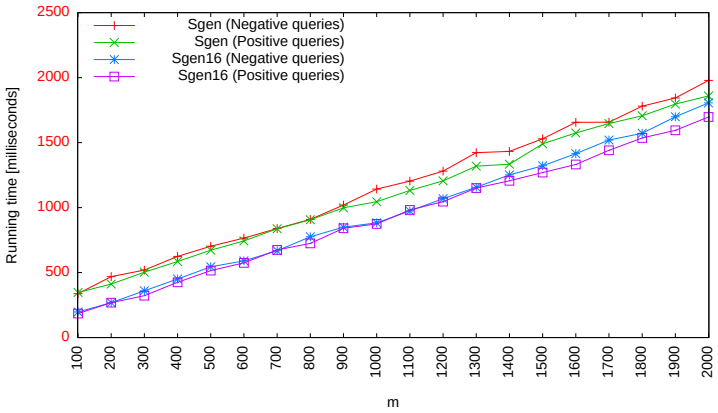
In subsection 4.2.2 it was demonstrated that the worst case running time for the suffix array is $O(m + \log(n))$. The results for Sgen, Sgen16 and Sgen18 from experiment 11-b are plotted in figure 4.5c Here it can be seen that the practical running time is dependent of n. This was expected as n has a direct influence on the average depth of the binary search performed for random positive queries. The graph also shows that the difference in the actual running time for the three text sizes is constant for growing sizes of m, like it would be for the worst case running time. It should be noted that the picture are the exact same for the a and c parts of the experiments, as well as for the two other text types.

The results for Sgen18, Bib18 and BibLow18 from experiment 11-b are plotted in figure 4.5d. From the graph it becomes clear that the actual running time is independent of $|\Sigma|$, as it is the case for the worst case running time. Again this is as expected since the alphabet size does not have any influence on the probability of a long binary search. Again it should be noticed that the same results are seen for the different sizes of m and the two other text sizes.

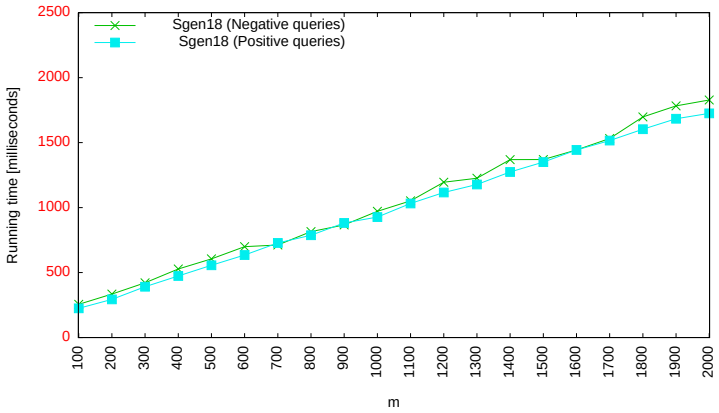
To get an idea of how the actual running time and the approximated worst case running time found in subsection 4.2.2 relate to each other, the results for Sgen from experiment

11-b and the found worst case running time are plotted in figure 4.5e. Quite surprisingly the actual running time is worse than the approximated worst case time for all sizes of m. It was expected that the practical running time would be close to the approximated worst case running time, since the number of character comparisons with a positive outcome always will be m for a positive queries, and the expected depth of the binary search for a random string is close to $\log(n)$. This does however not explain why the actual running time is worse than the approximated worst case running time. To find the explanation for this, we have to look at the way the worst case experiments are performed. To be able to get a measurable time it is necessary to perform the query multiple times. Theoretically the running times of the queries are independent of each other, but due to the the way a computer is build such as different levels of cache, this is not the reality. When performing the same query multiple times, things will be in the near cache already making the running time lower. In the worst case experiments queries where performed of the clock between the multiple worst case queries to simulate the cache behavior of random queries. Figure 4.5e indicates that the interleaving searches did not have the wished effect. The running times for Sgen and Sgen16 from experiment 11-b are plotted relative to the worst case running times for the two text sizes in figure 4.5f . Common for the two graphs is that they both are converging toward 100% when m increases. This together with the fact that the graph for Sgen16 starts lower, indicates that the extra work in the actual running time is dependent of the size of the text. This is consistent with the fact, that each time a jump in the binary search is performed, a random jump in the text is performed. These random jumps becomes expensive when parts of the array and text can not reside in the cache. It was most probably also the effect of this that showed in figure 4.2f from subsection 4.2.2, where the constant of the $\log(n)$ part of the worst case complexity was approximated. The algorithm is also forced to read in a larger part of the text when m increases, but the big difference is that this is read sequentially and thus is fast to load in. With this analysis in mind it becomes less surprisingly that the practical running time is slightly higher than the approximated worst case running time.

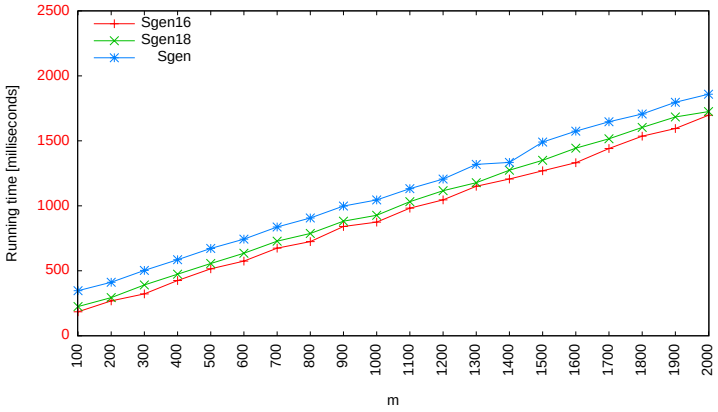
Most likely there is the same problem with the suffix trees approximated worst case running time, as the labels on the edges is represented by two pointers into the text, potentially resulting in random jumps when a new edge is followed. It is however not clear from figure 4.4d that these things affects the actual running time for the suffix tree, as the actual running time still is significantly lower than the approximated worst case running time.



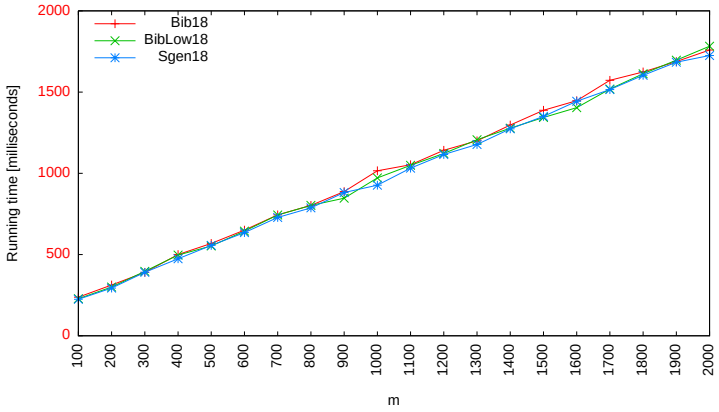
(a) Experiment 11-b and 12-b: Comparison of running times for Sgen and Sgen16 (suffix arrays).



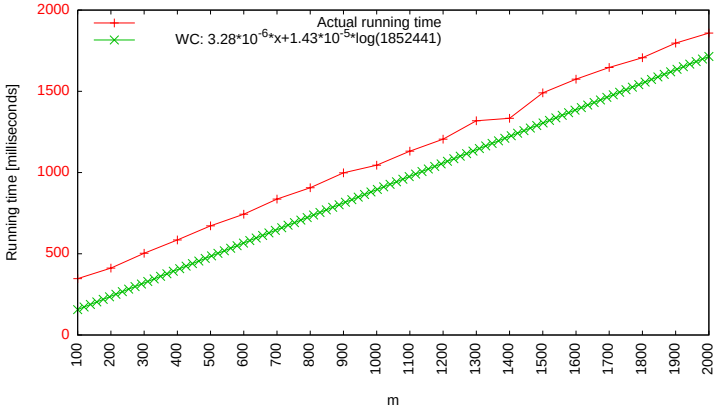
(b) Experiment 11b and 12-b: Comparison of running times for Sgen18 (suffix arrays).



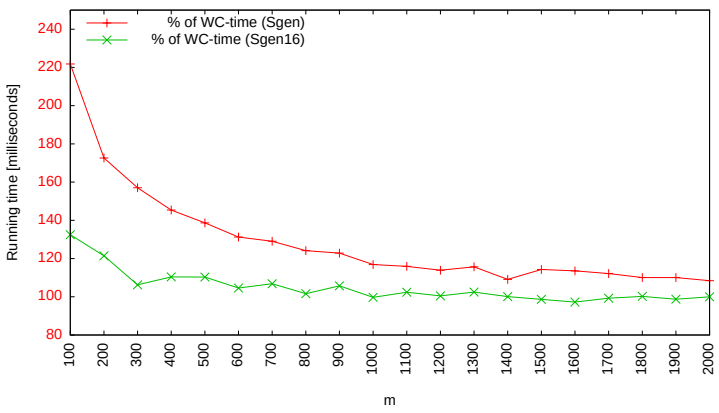
(c) Experiment 11-b: Running times for different sizes of n (suffix arrays).



(d) Experiment 11-b: Running times for different sizes of $|\Sigma|$ (suffix arrays).



(e) Experiment 11-b: Running times for Sgen compared with the approximated worst case running time for suffix arrays (subsection 4.2.2).



(f) Experiment 11-b: Running times for Sgen and Sgen16 as percentage of the approximated worst case running time for suffix arrays (subsection 4.2.2).

Figure 4.5: Graphs showing how the search algorithm in suffix arrays performs in practical use.

4.3.3 Suffix Tray

It is necessary to look at both positive and negative queries when evaluating the search function in suffix trays. This due to the same reasons that made it necessary for suffix arrays. The results for Sgen18, BibLow18 and Bib18 from experiment 11-b and 12-b are plotted in figure 4.6a-4.6c. This combination of texts are chosen as the size of the suffix intervals are depended of $|\Sigma|$, opposite the size of the suffix array that are dependent of n. The result is however the same as all three graphs shows no significant difference between the running times for the positive and the negative queries. To be sure this was the real picture, the data from the other sizes of m and n were controlled, showing the same results.

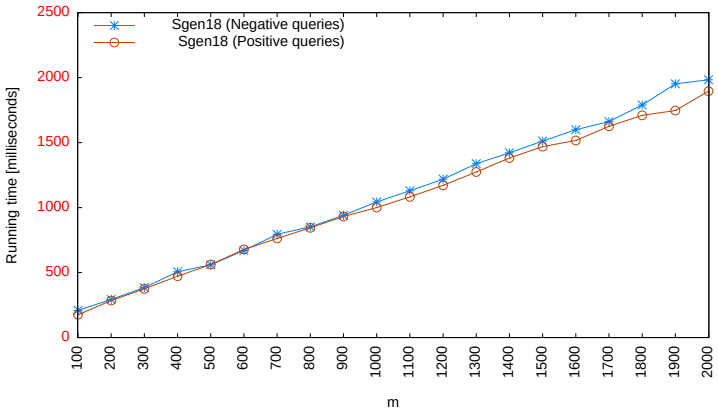
The analysis performed in the rest of this subsection will thus be on the data for the positive queries, but will apply to both negative and positive queries.

In subsection 4.2.1 it was demonstrated that the worst case running time for the suffix tray is $O(m + \log(|\Sigma|))$ and thus is independent of n. The results however shows that this is not the case for the practical running time. The results for Sgen, Sgen16 and Sgen18 from experiment 11-b are plotted in figure 4.7a. As it was the case for the suffix tree, the running time is increased for growing sizes of n. The reason behind is the same as it was for the suffix tree. The tree structure in the suffix tray is build from the σ-nodes in the corresponding suffix tree. The size of the corresponding suffix tree is increased when n is increased, implying that the subtrees of the different nodes becomes larger. As an effect of this, there are a higher number of branching and non branching σ-nodes in the suffix tree, and thereby also a higher number of nodes in the suffix tray. From the worst case experiment it was clear that the most expensive way to compare characters is through branching or non branching σ-nodes, hence the increased running time. Again all text sizes and values of m showed the same results.

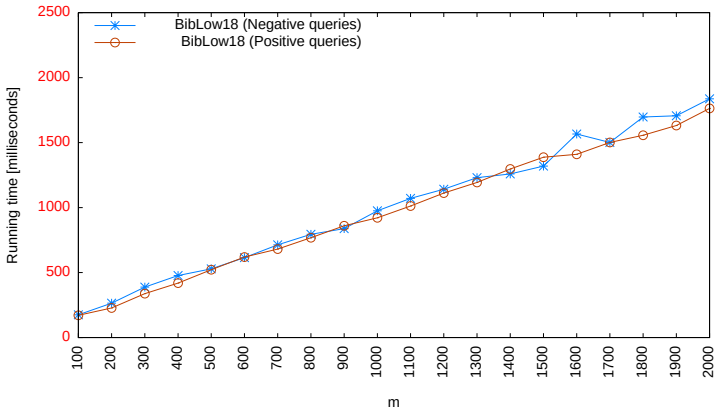
In figure 4.7b the running times for Sgen18, BibLow18 and Bib18 from experiment 11-b are plotted. Here it can be seen that there is a dependency between the running time and $|\Sigma|$. Opposite the worst case complexity, there is a tendency that the practical running time is decreasing for larger sizes of $|\Sigma|$. In addition to this there seems to be a relationship between the advantaged gained with larger alphabet sizes, and the size of m. That these tendencies also applies for larger sizes of m, can be seen in figure-**??** where the results from Experiment 11-c can be seen. Furthermore the result for the other sizes of n and m showed the same results. To understand why this happens, it is needed to look deeper into how alphabet sizes affects the suffix tray structure. With a larger alphabet, the suffix interval-nodes potentially gets larger, as these by construction have a maximum size of $|\Sigma|^2$. This is the effect that can be seen in the worst case complexity. What happens which can not be seen in the worst case complexity is though, that the number of nodes that are turned into branching or non branching σ-nodes, when the text size is increased is larger for small alphabets, due to the definition of σ-nodes. The number of extra nodes traversed is thus depended of both $|\Sigma|$ and n. This means figure 4.7b consequently shows that the time penalty suffered from traversing a potentially larger suffix interval, in practice

is outweighed by the time penalty suffered from traversing a potentially higher number of nodes.

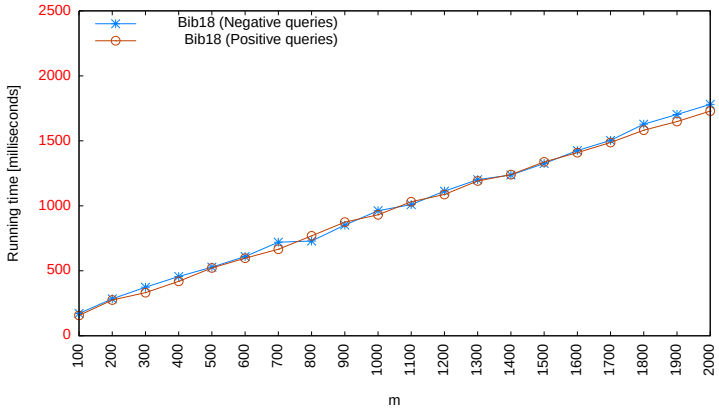
Again the approximated worst case running time is plotted with the results for Sgen from experiment 11-b. The graphs can be seen in figure 4.7d. The reason for plotting Sgen is that Sgen has the lowest worst case complexity of the nine texts, while Sgen on the other hand has the highest actual running time of the nine texts. This means Sgen is the text closest to its approximated worst case running time. Like it was the case for the suffix tree, the worst case running time seems to be a bad tool to describe the actual running time when m becomes large. To get a more clear view the relationship between the actual and the worst case running time is plotted in figure 4.7d. As a combination of a tree structure and a suffix array it is not surprising that the relative running time has things in common with both the results for the suffix trees and the results for the suffix array. For small sizes of m the actual running time are slightly higher than the worst case running time. With large probability the search ends in a suffix interval-node, as the subtree of the position from where the search ends in the corresponding suffix tree else should contain at least $|\Sigma|$ nodes. This implies that at least $|\Sigma|$ suffixes should share a prefix of size m to make the search in the suffix tray end in a σ-node. As the approximated worst case running time for the suffix array are used as the worst case running time for the suffix interval-nodes, the same problems applies. The practical running time is however only just over the worst case running time, and converge towards 40% in this case. This is due to the fact that the arrays in the suffix interval-nodes are smaller than the suffix array, resulting in a lower number of binary steps. In addition to this, part of the search is done within the tree structure where, as we saw for the suffix tree, the practical running time is significantly lower than the worst case running time. This primely due to a high number of characters on per label.



(a) Experiment 11-b and 12-b: Comparison of running times for Sgen18 (suffix trays).

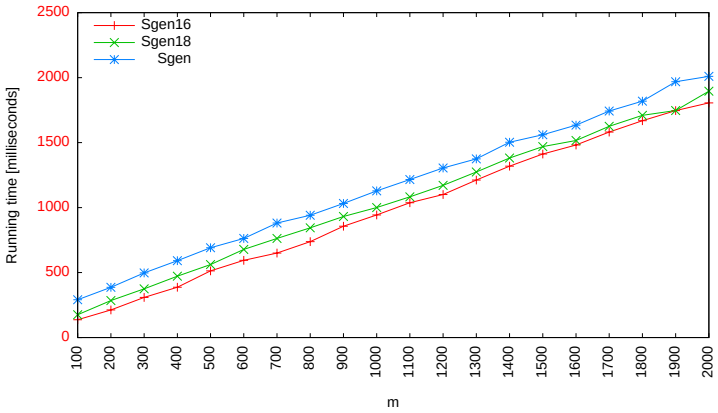


(b) Experiment 11-b and 12-b: Comparison of running times for BibLow18 (suffix trays).

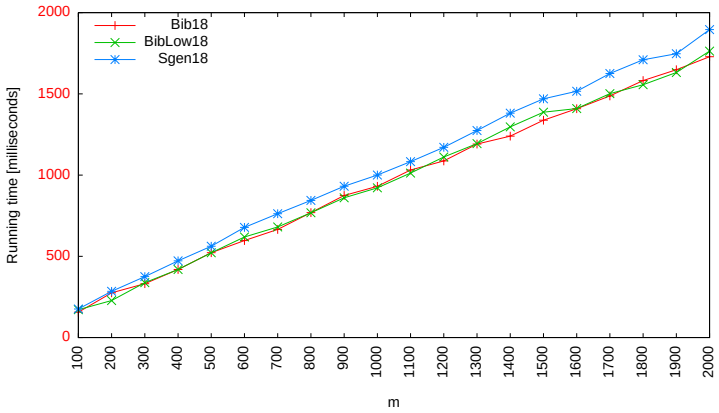


(c) Experiment 11-b and 12-b: Comparison of running times for Bib18 (suffix trays).

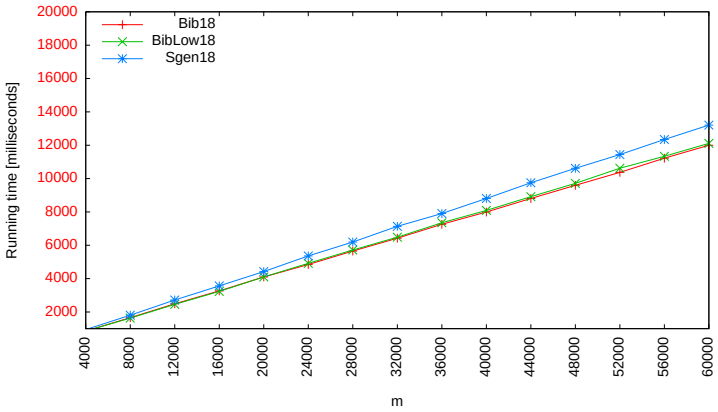
Figure 4.6: Graphs showing how the search algorithm in suffix trays performs on respectively positive and negative queries.



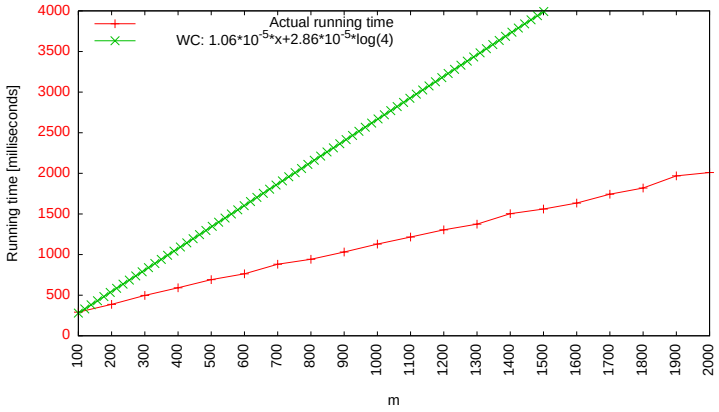
(a) Experiment 11-b: Running times for different sizes of n. (suffix trays)



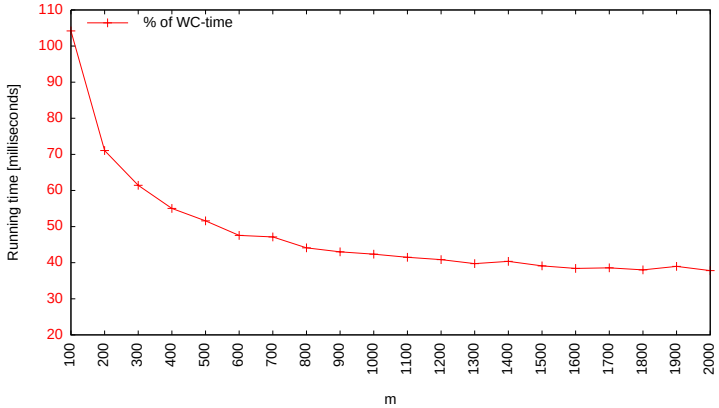
(b) Experiment 11-b: Running times for different sizes of $|\Sigma|$. (suffix trays)



(c) Experiment 11-c: Running times for different sizes of $|\Sigma|$. (suffix trays)



(d) Experiment 11-b: Running times for Sgen compared with the approximated worst case running time for suffix trays (subsection 4.2.3).



(e) Experiment 11-b: Running times for Sgen as percentage of the approximated worst case running time for suffix trays (subsection 4.2.3).

Figure 4.7: Graphs showing how the search algorithm in suffix trays performs in practical use.

4.4 Comparison of the structures

To get a clear view of how the different structures performs compared to each other, and when it is beneficial to use which structure, three tables are created. A table is created for each type of text, while a row in the tables represent texts of specific length. In the first three columns the actual running times are compared, while the approximated worst case running times are compared in the last three columns. In the columns of the different structures are written the sizes of m where the particular structure is the fastest. In some cases the running times for the different structures are so close to each other, that random deviations makes the graphs cross each other multiple times. In these cases the structure currently holding the status as the fastest structure keeps it status, until a clear pattern can be seen.

To decide which structure that are the favorable one in practice, the data from Experiment 11 are used. It should be emphasized that the tables purely shows which structure that are the fastest for a particular size of m. The graphs comparing the three structures performance on respectively the Staphylococcus texts, the lower case bible texts and the unchanged bible texts are plotted in the figures 6.1 - 6.5 and can be seen in the appendix. In these graphs knowledge about how much faster a structure is than the two others can be found.

The first table is showing the data for the Staphylococcus texts.

$\lvert\Sigma\rvert = 4$	Suffix Tree	Suffix Array	Suffix Tray	WC-Tree	WC-Array	WC-Tray
Sgen16 ($n = 2^{16}$)	$12.000 \rightarrow$	$800 - 12.000$	$0 - 800$	$-$	$24 \rightarrow$	$0 - 24$
Sgen18 ($n = 2^{18}$)	$16.000 \rightarrow$	$500 - 16.000$	$0 - 500$	$-$	$27 \rightarrow$	$0 - 27$
Sgen($n \approx 2^{20.8}$)	$24.000 \rightarrow$	$400 - 24.000$	$0 - 400$	$-$	$33 \rightarrow$	$0 - 33$

As the analysis in subsection 4.3 showed, the suffix tree and the suffix tray, both containing tree structures, suffered the most when the text sizes was increased. This also shines through here, where the interval where the suffix tray is the favorable algorithm increases with n. Looking at the worst case running times, the suffix array quickly becomes the favorable one, when m is increased. It was clear in the practical experiments that the approximated constant for the suffix arrays worst case complexity was a bit low, meaning that the real size of m, from where the suffix array is better than the suffix tray may be a bit higher.

The second table is showing the data for the bible texts where the capital letters is exchanged with lower case letters.

$\|\Sigma\| = 55$	Suffix Tree	Suffix Array	Suffix Tray	WC-Tree	WC-Array	WC-Tray
BibLow16 ($n = 2^{16}$)	$8.000 \rightarrow$	–	$0 - 8.000$	–	$9 \rightarrow$	$0 - 9$
BibLow18 ($n = 2^{18}$)	$8.000 \rightarrow$	$2.000 - 8.000$	$0 - 2.000$	–	$13 \rightarrow$	$0 - 13$
BibLow ($n = 2^{21}$)	$20.000 \rightarrow$	–	$0 - 2.0000$	–	$19 \rightarrow$	$0 - 19$

Here the suffix tray performs even better, as it overtakes some of the suffix arrays dominance. The analysis in subsection 4.3 showed that the practical running time of the suffix tray improves as the alphabet size is increased. The suffix tray and the suffix array however performs close to the same level, which is also why the suffix array is the fastest structure for a small interval in BibLow18. The picture drawn by the worst case running times practically unchanged.

The third table is showing the data for the unchanged bible texts.

$\|\Sigma\| = 80$	Suffix Tree	Suffix Array	Suffix Tray	WC-Tree	WC-Array	WC-Tray
Bib16 ($n = 2^{16}$)	$8.000 \rightarrow$	–	$0 - 8.000$	–	$7 \rightarrow$	$0 - 7$
Bib18 ($n = 2^{18}$)	$4.000 \rightarrow$	–	$0 - 4.000$	–	$11 \rightarrow$	$0 - 11$
Bib ($n = 2^{21}$)	$24.000 \rightarrow$	–	$0 - 24.000$	–	$16 \rightarrow$	$0 - 16$

Again we see how the actual running time for the suffix tray becomes relatively better, as the suffix tray has completely overtaken the parts where the suffix array previous was the fastest structure. Looking at all three tables, there are slightly changes in when the suffix tree becomes the favorable structure. This is however primary because of deviations in the measured times, as the analysis in subsection 4.3 showed that the practical running time for the suffix tree was untouched by the alphabet size. Lastly the worst case running time of the suffix array has taken yet another bite of the intervals that were dominated by the worst case running time for the suffix tray.

In general the result gathered in the tables shows, that if the focus is purely on the worst case running times for the implemented structures, the suffix array becomes the favorable structure. In a more practical view, the suffix tray performs quite well, and is the preferable structure in cases where the sizes of m is small. The suffix tree clearly has its force when m grows large, while the suffix array is a nice alternative for the in between sizes. For sizes of m between $0 - 100$ the suffix tree is notable worse that the suffix array and the suffix tray. For all sizes of m larger than 100, the relative difference in the actual running time, however newer becomes more than a factor of 2.

Chapter 5

Conclusion

In subsection 4.2 experiments were performed to decide the worst case complexities of the search functions in the implemented suffix tray, array and tree. The results showed that the worst case complexity of the implemented search functions followed the theory. The constants in the asymptotic running times where approximated for all three data structures, and were found to be:

Suffix Tree: $4,8 * 10^{-5} * m * log(|\Sigma|)$

Suffix Array: $3,28 * 10^{-6} * m + 1,43 * 10^{-5} * log(n)$

Suffix Tray: $1,06 * 10^{-5} * m + 2,86 * 10^{-5} * log(|\Sigma|)$

In subsection 4.3 random queries where performed on all three structures, to simulate how the structures performs in real life scenarios. The results from subsection 4.3 showed that the actual running time for the suffix trees were significantly lower than the estimated worst case running time. The actual running times for the suffix arrays were a factor of two higher than the estimated worst running times for small sizes of m, while it for larger sizes of m converged towards the estimated worst case running time. The actual running times for the suffix tray, showed that it is a combination of a tree structure and an array, by being only slightly higher than the estimated worst case time for small sizes of m, and significantly lower for large sizes of m. The problem that the actual running time for the suffix array and the suffix tray for some sizes of m were higher that the estimated worst case running time, was found to be due to the fact that it is necessary to perform multiple queries to get a measurable time. With things such as branch prediction and multiple levels of cache in modern computers, queries that should be independent of each other according to the theory becomes dependent. This makes it close to impossible to tell which query that will result in the highest constant for the worst case complexity at a specific moment.

Looking purely at the found worst case running times, the suffix array was the preferred structure for close to all sizes of m. The results from subsection 4.4 however showed that the

suffix tray is the favorable structure for small sizes of m, when looking at the actual running time, making suffix trays not only of theoretical relevance but also practical relevance. The suffix tray proved best for sizes of m up to 400/24000, and had its force in scenarios where the size of n and $|\Sigma|$ were large. The actual running times of the suffix array were close to those of the suffix tray making the two structures relatively even. The suffix tree on the other hand was slightly better for larger sizes of m but noticeably worse for small sizes of m.

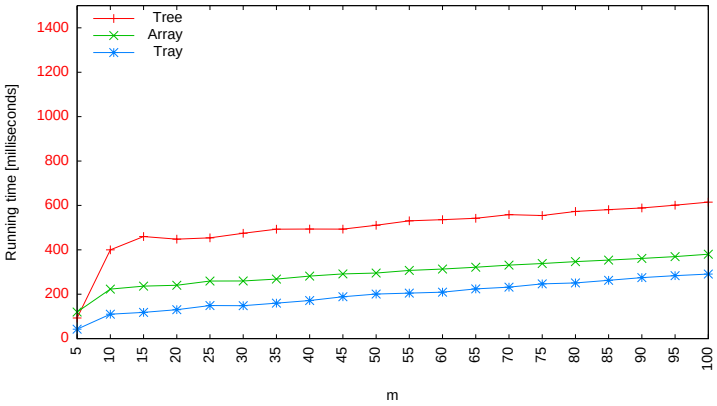
5.0.1 Future work

As future work it could be interesting to make statistics surrounding the use of exact pattern matching. Things as the length of the queries, how often the queries are negative and how early the negative queries deviates from the suffixes relative to the length of the queries, could all be used to decide which structure to use. I.e. the suffix tray may be better than the suffix tree for large sizes of m, if the queries most of the time are negative and deviates early.

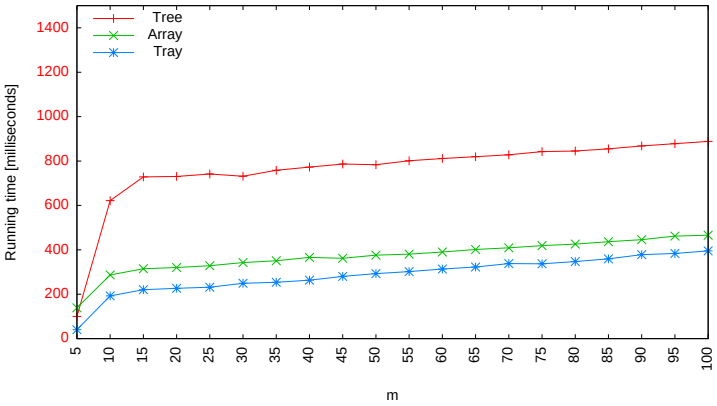
Another possibility for future work is to implement the described algorithms for finding all occurrences, and perform experiments to see how these performs compared to each other. An efficient implementations of the algorithms constructing the three data structures could also be a possibility. This to investigate the amount of extra time used on preprocessing for suffix trays relative to suffix trees and suffix arrays.

Chapter 6

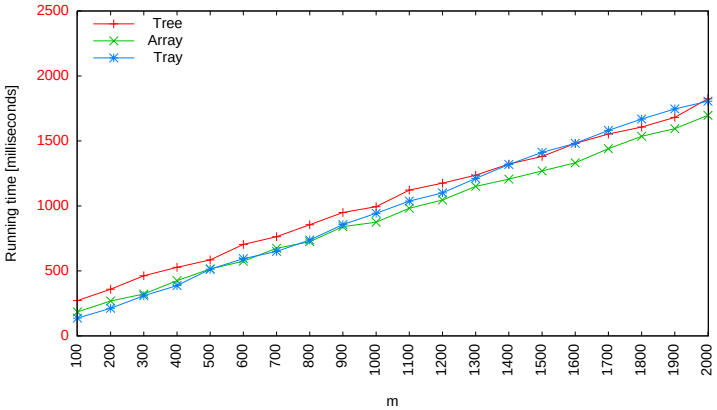
Appedix-Figures



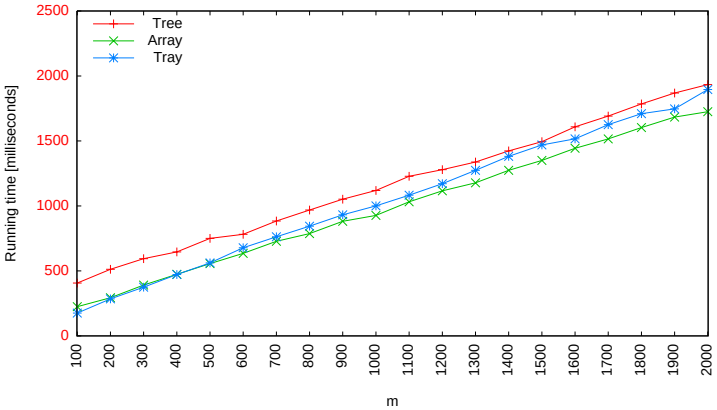
(a) Experiment 11-a: Sgen16.



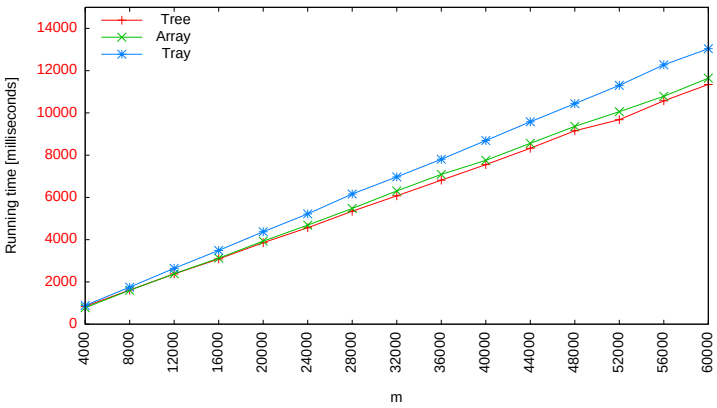
(b) Experiment 11-a: Sgen18.



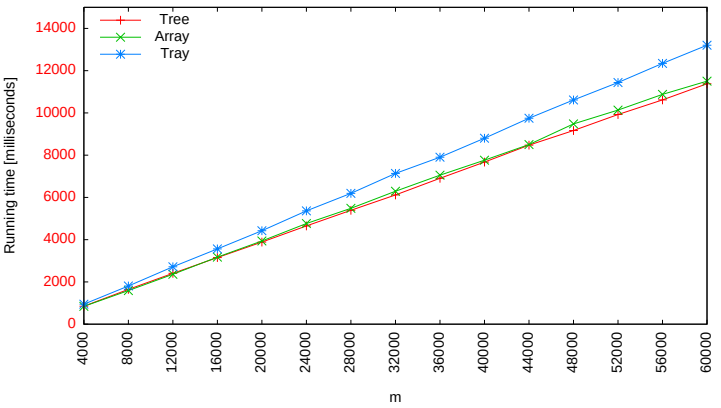
(c) Experiment 11-b: Sgen16.



(d) Experiment 11-b: Sgen18.

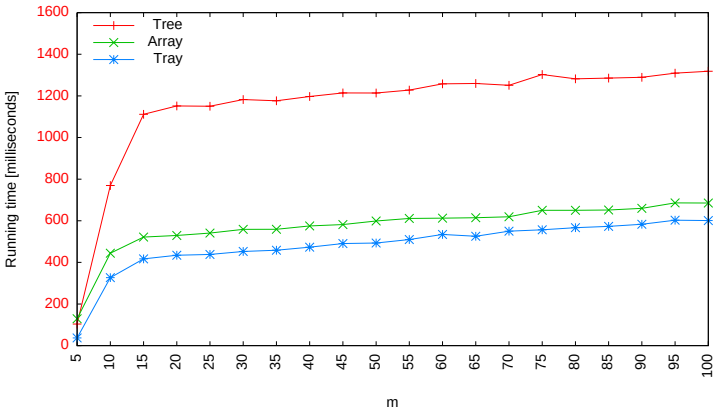


(e) Experiment 11-c: Sgen16.

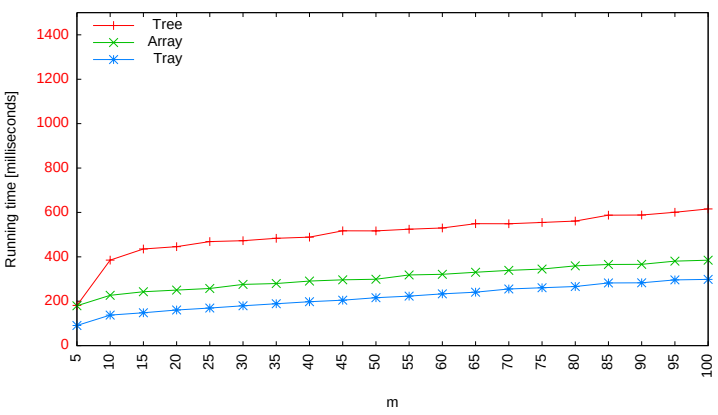


(f) Experiment 11-c: Sgen18.

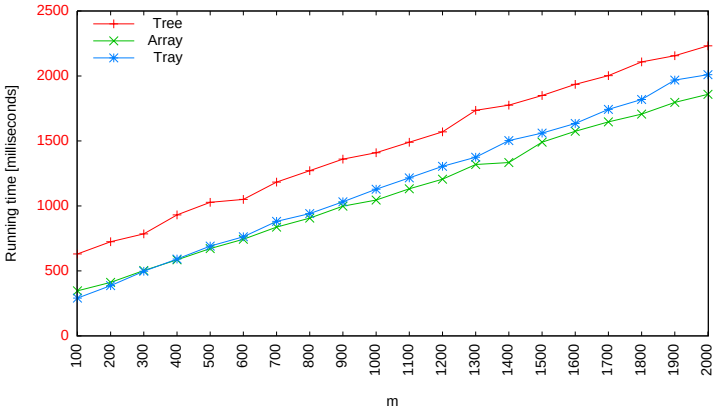
Figure 6.1: Graphs showing the results from experiment 11-a,11-b and 11-c, designed to compare the pracitcal performance of the three structures. (Sgen16 and Sgen18)



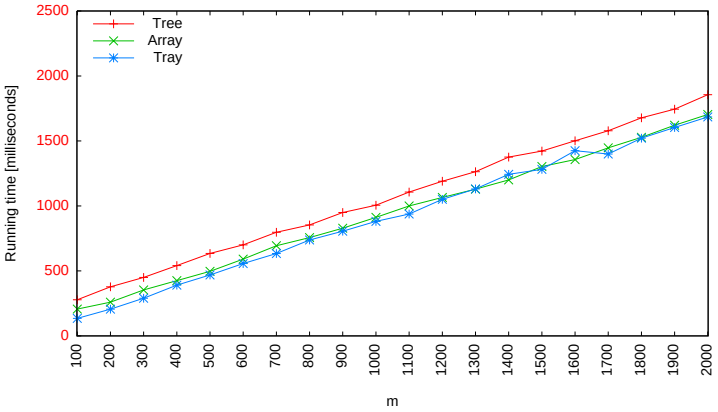
(a) Experiment 11-a: Sgen.



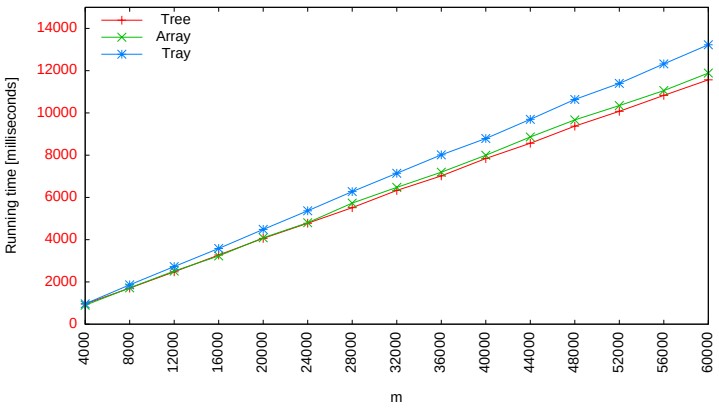
(b) Experiment 11-a: BibLow16.



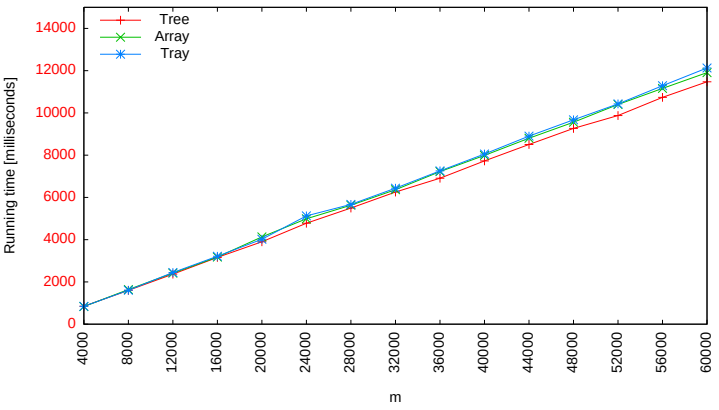
(c) Experiment 11-b: Sgen.



(d) Experiment 11-b: BibLow16.

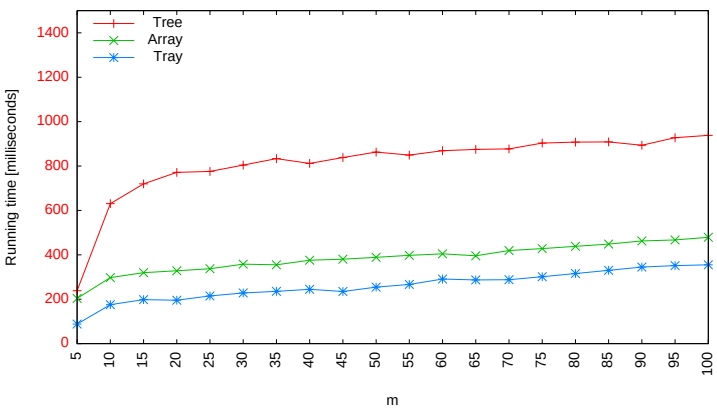


(e) Experiment 11-c: Sgen.

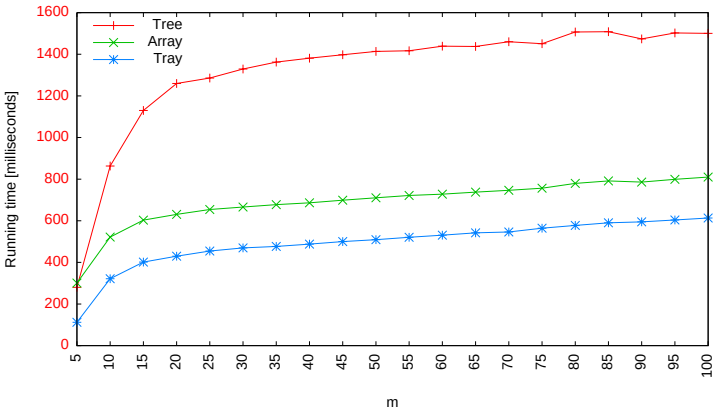


(f) Experiment 11-c: BibLow16.

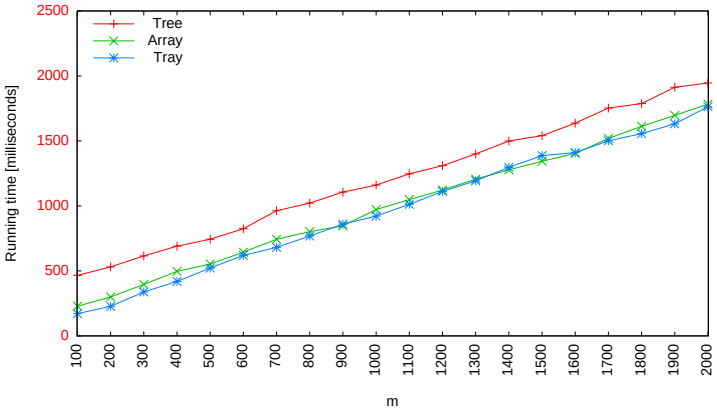
Figure 6.2: Graphs showing the results from experiment 11-a,11-b and 11-c, designed to compare the pracitcal performance of the three structures. (Sgen and BibLow16)



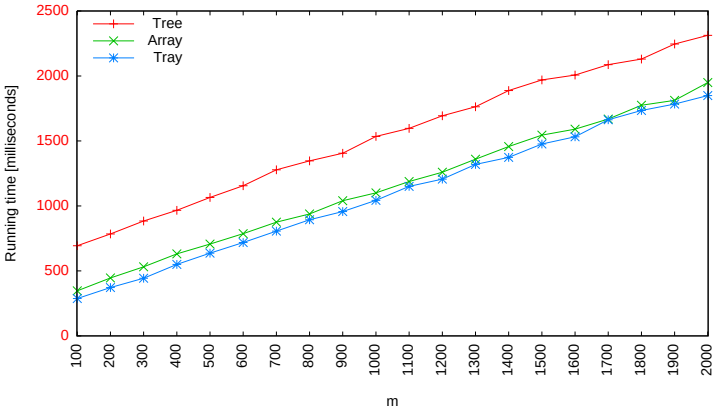
(a) Experiment 11-a: BibLow18.



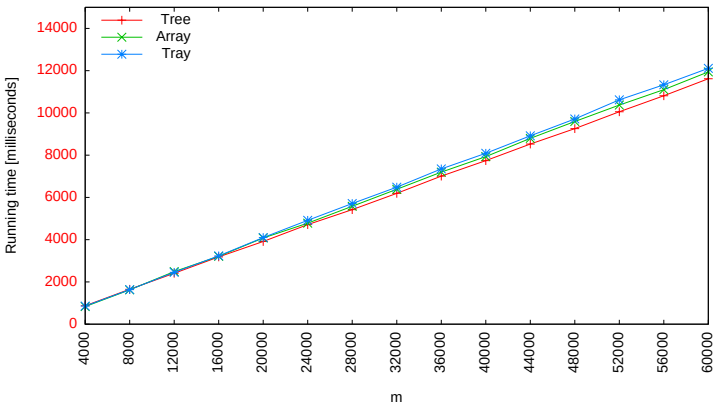
(b) Experiment 11-a: BibLow.



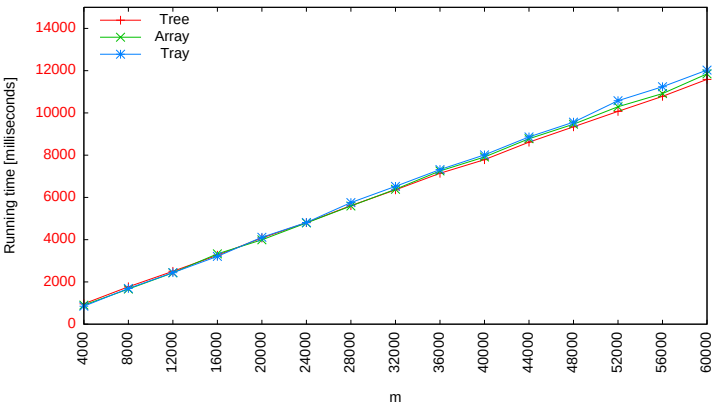
(c) Experiment 11-b: BibLow18.



(d) Experiment 11-b: BibLow.

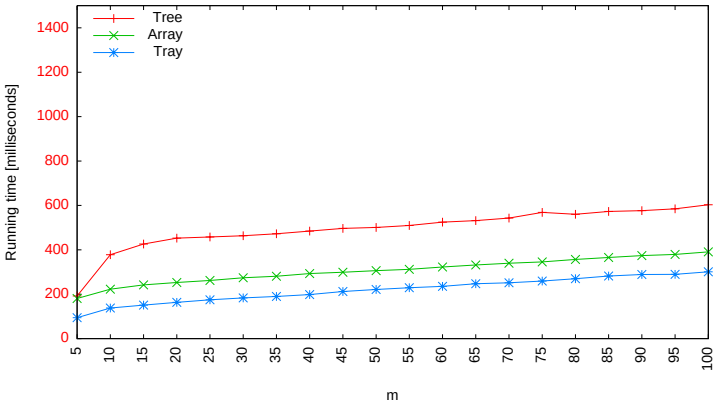


(e) Experiment 11-c: BibLow18.

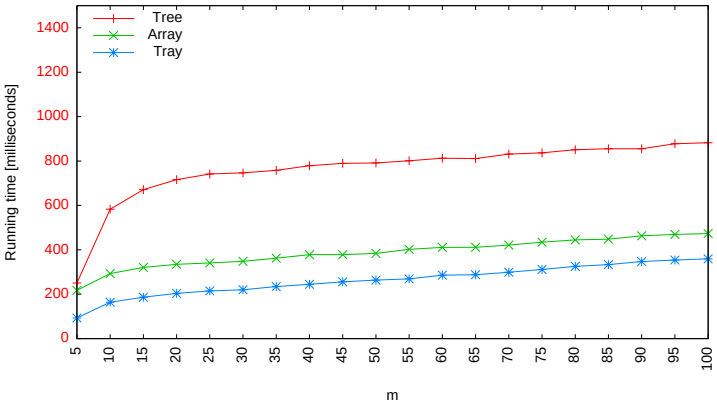


(f) Experiment 11-c: BibLow.

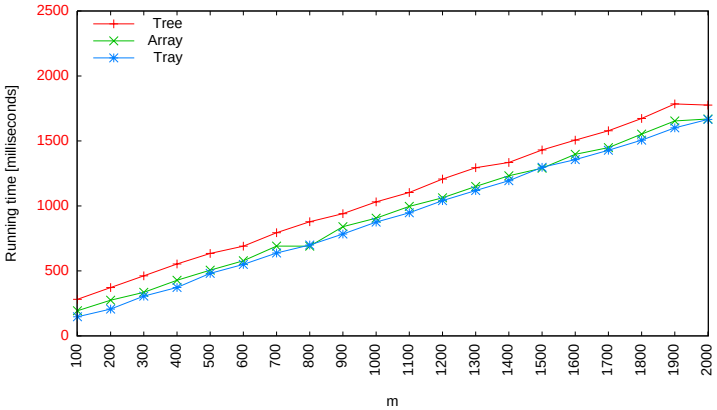
Figure 6.3: Graphs showing the results from experiment 11-a,11-b and 11-c, designed to compare the pracitcal performance of the three structures. (BibLow18 and BibLow)



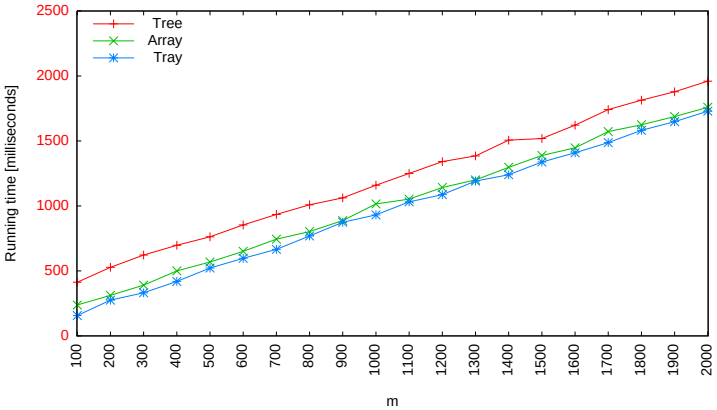
(a) Experiment 11-a: Bib16.



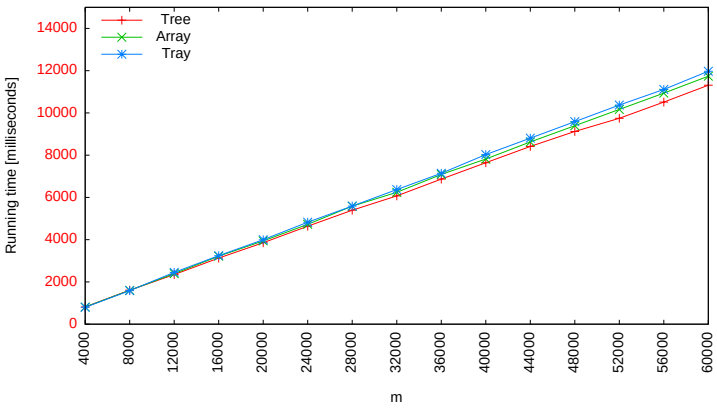
(b) Experiment 11-a: Bib18.



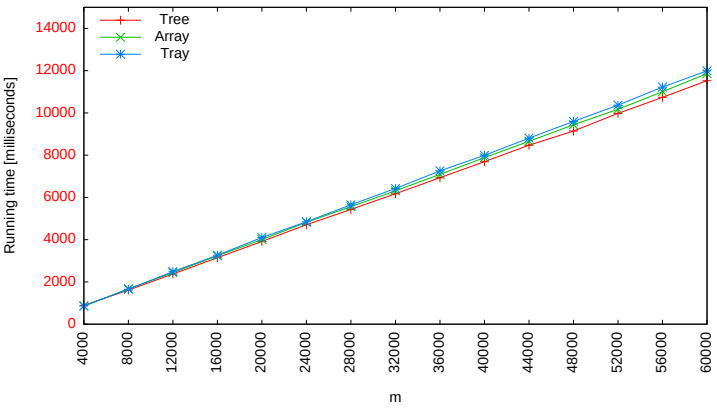
(c) Experiment 11-b: Bib16.



(d) Experiment 11-b: Bib18.

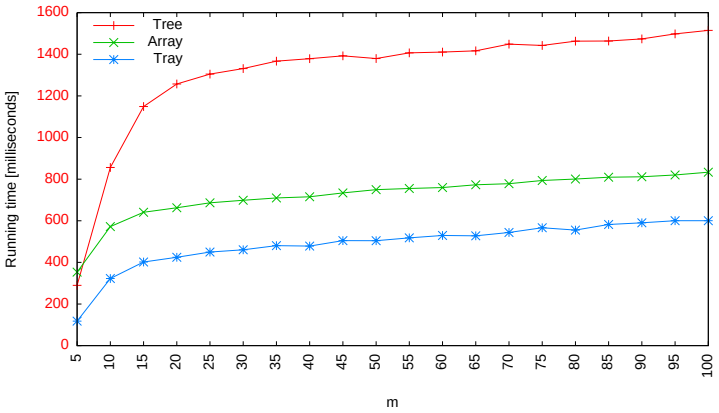


(e) Experiment 11-c: Bib16.

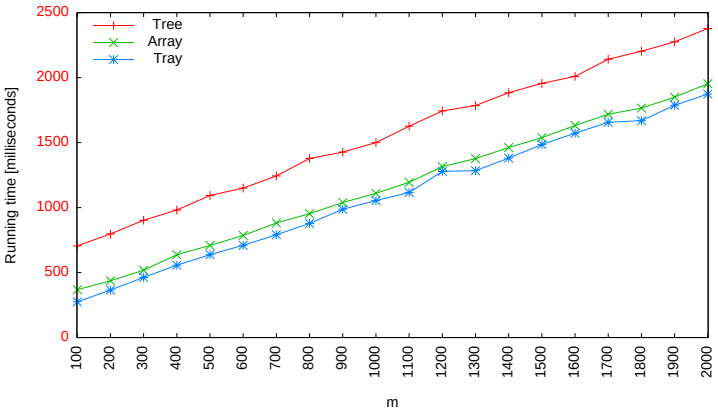


(f) Experiment 11-c: Bib18.

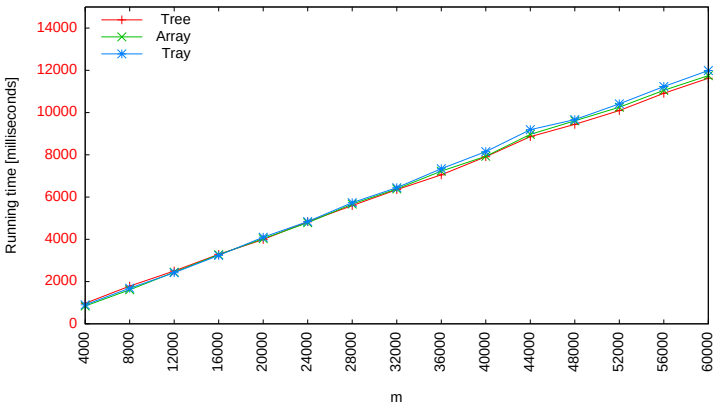
Figure 6.4: Graphs showing the results from experiment 11-a,11-b and 11-c, designed to compare the pracitcal performance of the three structures. (Bib16 and Bib18)



(a) Experiment 11-a: Bib.



(b) Experiment 11-b: Bib.



(c) Experiment 11-c: Bib.

Figure 6.5: Graphs showing the results from experiment 11-a,11-b and 11-c, designed to compare the pracitcal performance of the three structures. (Bib)

Bibliography

[1] Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix trays and suffix trists: Structures for faster text indexing. *Algorithmica*, pages 1–17, 2014.

[2] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.

[3] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[4] Bill Smyth. *Computing patterns in strings*. Pearson Education, 2003.