Abstract

The research papers about suffix arrays have grown many, and asymptotically better algorithms are being developed. There are, however, two areas that seem to have been a little forgotten – searching in external memory and document retrieval from a suffix array. We present and compare four different methods for document retrieval from an external suffix array. Our results show that only one yields adequate results in the presence of many documents, namely embedding document information into the suffix array. We also touch on the subject of searching external suffix arrays, presenting and discussing four techniques.

Contents

Abstract **3**

1 Introduction **7**

2 Background **8**
2.1 Strings . . . 8
2.2 String searching . . . 8
2.3 Document retrieval . . . 9
2.4 Suffix arrays . . . 9
2.5 Suffix trees . . . 10
2.6 Memory model . . . 11

3 Search algorithms **12**
3.1 Internal suffix array search . . . 12
3.1.1 Without LCP . . . 13
3.1.2 With LCP . . . 16
3.2 External suffix array search . . . 17
3.2.1 SPAT . . . 17
3.2.2 Interpolation search . . . 17
3.2.3 LCP . . . 18
3.2.4 Interleaved text . . . 18
3.3 Document retrieval . . . 19
3.3.1 Direct . . . 19
3.3.2 Lookup . . . 19
3.3.3 Embedded . . . 20
3.3.4 Search . . . 20
3.3.5 Optimal . . . 22

4 Construction algorithms **23**
4.1 Document retrieval . . . 23
4.1.1 Direct . . . 23
4.1.2 Lookup . . . 23

5 Implementation **24**
5.1 Memory management . . . 25
5.1.1 mlock . . . 25
5.1.2 Disk access . . . 26
5.1.3 new . . . 26
5.2 Result gathering . . . 27
5.2.1 Timing . . . 27
5.2.2 Scripts . . . 27
5.3 Disk test . . . 28
5.4 Suffix array construction . . . 28
5.4.1 DC3 . . . 28
5.4.2 Embedded . . . 28
5.5 Suffix array search . . . 29

5.6 Document information construction . . . 30
5.6.1 Direct . . . 30
5.6.2 Lookup . . . 30
5.7 Document retrieval . . . 30
5.7.1 Direct . . . 30
5.7.2 Lookup . . . 31
5.7.3 Removing duplicates . . . 31

6 Experiments **32**
6.1 Machine configuration . . . 32
6.1.1 Details . . . 32
6.2 Input . . . 33
6.2.1 Text . . . 33
6.2.2 Documents . . . 34
6.2.3 Queries . . . 34
6.3 Results . . . 35
6.3.1 Disk speed . . . 35
6.3.2 Suffix array construction . . . 36
6.3.3 Document information construction . . . 36
6.3.4 Suffix array search . . . 38
6.3.5 Duplicate removal . . . 38
6.3.6 Document listing . . . 39

7 Summary and conclusions **42**

Bibliography **44**

A Range Minimum Query and Nearest Common Ancestor **45**

List of Figures

3.1 Document retrieval tables D and C . . . 22

6.1 Disk performance . . . 35
6.2 Block burst . . . 36
6.3 Document information construction, varying document alignment . . . 37
6.4 Document information construction, varying number of documents . . . 37
6.5 Suffix array search, varying text length . . . 38
6.6 Duplication removal, time . . . 39
6.7 Duplication removal result . . . 39
6.8 Varying alignment size, time . . . 40
6.9 Varying alignment size, memory usage . . . 40
6.10 Method comparison, time . . . 40
6.11 Method comparison results . . . 40
6.12 Method comparison, time . . . 41

Chapter 1

Introduction

Suffix arrays are data structures that index some text to facilitate fast searching and retrieval of all occurences. The past years have seen a raging development in suffix array construction.

Traditionally suffix arrays have only been suitable for internal memory – it took too long to construct a suffix array in external memory. In 2003, however, this began to change with the introduction of three different suffix array construction algorithms that ran in linear time, [KS03], [KA03] and [KSPP03]. In 2005, feasible external memory suffix array construction was introduced by Dementiev et al. in [DKMS05], using the techniques from [KS03]. In last four years, at least fifteen articles regarding suffix arrays have been published. As much as twice that number of articles regarding closely related data structures have been published.

Yet, noone seems to have looked into the problem of multiple documents and suffix arrays, with the honest exception of Muthukrishnan [Mut02]. His solution requires much preprocessing and extra storage – it is not fit for external memory.

With this thesis I hope to cover some ground on the problem of retrieving documents from a suffix array in external memory. I will present various techniques for retrieving documents and compare them to each other.

Also, I will try the external suffix array construction presented in [DKMS05] and briefly look into various external memory suffix array search algorithms, such as that of Baeza-Yates et al. in [BYBZ96].

Chapter 2

Background

In this chapter we will introduce and define some basic concepts, problems and models. A loose sketch of a solution to a problem will be presented when appropriate.

After reading this chapter you will have a overview of the fields we touch in this thesis, as well as understanding of our notation. String processing articles suffers somewhat from a wide variety of notations, with respect to the letters used to denote various concepts and the actual syntax. We have tried to use a uniform naming scheme of variables, most notably capital letters for sets and lower case letters for sizes. Some places we have used $|S|$ notation to denote the size of set S, if a lower case variable is not readily available. Our goal has been to achieve mathematical clarity in the notation, while trying to keep it reasonably close to actual implemenation.

2.1 Strings

A *string* is a sequence of symbols, denoted $T = T[0]T[1]...T[n-1]$, $T[0]T[1]...T[n-1]$ being the concatenation of individual symbols in the string. The length of a string T is its number of characters $n = |T|$. When discussing string algorithms, T denotes the source text with length n and $P = P[0]P[1]...P[n-1]$ denotes a search pattern of length $m = |P|$.

A *substring* is a contiguous part of another string, denoted $T_{i..j} = T[i]T[i+1]...T[j-1]T[j]$.
A *prefix* of T is a substring starting at t_0. The ith prefix of T is thus $T[0...i-1] = T[0]T[1]...T[i-1]$.
A *suffix* of T is a substring ending at t_{n-1}. It is a particularly vital notion in this thesis. The ith suffix of T is denoted $T_{i-1} = T[i-1]T[i]...T[n-1]$.

Another important notation with respect to suffix arrays, which will be introduced shortly, is that of longest common prefix (lcp). The lcp between two strings A and B is the longest prefix of A that is also a prefix of B. If the length of the lcp between A and B is j, we have $A[0] = B[0]$, $A[1] = B[1]$, ..., $A[j-1] = B[j-1]$, $A[j] \neq B[j]$.

2.2 String searching

String searching is about finding one or more occurences of a pattern P in a text T. With respect to suffix arrays, it is most of the time about finding all the occurences of P in T. There are two important limitations that need to be made about the scope of pattern matching.

First, there are many forms of patterns. The most basic one is a simple pattern, where the matching is just about the plain text in the symbols. We will refer only to this kind of string searching – matching a simple pattern. Other kinds of patterns include don't care symbols and full fledged regular expressions; these will not be discussed, as

these are own fields of study. If $T_{i..i+w-1} = P$, we say that there is a *match* or an occurence of P in position i. There can be at most $n - w + 1$ matches, because of the length of the pattern.

Second, there are two fundamentally different groups of string searching algorithms, indexing and non-indexing algorithms. The non-indexing algorithms have only one phase, the search phase[1]. It searches the text for the pattern. Many clever algorithms have been designed for this, but all depend heavily on the size of the text.

The indexing group of algorithms preprocesses the text. When searching for a pattern, information from the preprocessing phase is used to make the search itself quicker. These algorithms should be used when there tend to be many queries on the same text and/or when search time is of the essence. Well known indexing techniques include inverted files, suffix trees and suffix arrays. The inverted files [HBYFL92] are word-based indexes, meaning that you can only search for entire words in them. Suffix trees and suffix arrays are full-text indexes, meaning that you can search for any substring of the text in them.

2.3 Document retrieval

Document retrieval is a special kind of string searching problem. It is about finding all the documents which match a pattern. This can include advanced requirements such as that the pattern has to occur at least X times and at most Y times in certain contexts within a document, or the simple requirement that the pattern must occur at least once within a document for it to match.

We will only focus on the simple problem of finding all documents in which the pattern occures at least once. This is known as the document listing problem (see chapter 3.

We will use a collection of documents, $D = \{D^0, D^1, ..., D^{d-1}\}$. A suffix array is not fit to be constructed from a collection of documents, it deals with a single text, as we shall see in the next section. Thus, we concatenate the documents to produce a single text. Between[2] the documents, we insert special markers, $, so that it is not possible to get a match across document borders. $T = D^0\$D^1\$...D^{d-1}$.

2.4 Suffix arrays

A suffix array is a data structure that is the result of preprocessing a text T of length n. There are a wide variety of methods available to build this data structure, the original by Manber and Myers [MM91], the optimal [KS03], the fast [SS05] or the external [DKMS05]. The selection of algorithms to search for a simple pattern in the text using the suffix array is more limited. Search algorithms are explained in chapter 3. Here we will briefly describe the suffix array data structure.

Although it might appear confusing at first, the suffix array is a really simple data structure. It represents a lexicographically sorted list of all the suffixes of the text T. Consider the text “banana”. Since the length $n = 6$, it has 6 suffixes, banana, anana, nana, ana, na and a. Sorting this list we get a, ana, anana, banana, na and nana. Now we look at the positions of these suffixes in the text. a is at position 5, ana at position 3, anana at 1, banana at 0, na at 4 and nana at 2.

The suffix array for “banana” is the array $SA = [5, 3, 1, 0, 4, 2]$. Each number in the suffix array represents a suffix in the text. $SA[0] = 5$ represents $T_5 = a$. $SA[2] = 1$ represents $T_1 = anana$. Consider that some of the suffixes are prefixes of other suffixes. T_5 is a prefix of both T_3 and T_1. T_3 is a prefix of T_1. The suffixes that are prefixes of other suffixes are sorted first. This is no coincidence, it is the way a suffix array must be sorted.

Note that when we say the way a suffix array must be sorted, we are referring to the way the suffixes to which the entries in the suffix array points, must be sorted. This statement is somewhat tiresome to read, so when there is no

[1]Actually, there is some kind of crossover that tend to be counted among the non-indexing algorithms. Algorithms that preprocess the pattern. Preprocessing of the pattern, however, is an insignificant task compared to preprocessing the text

[2]For simplicity, this marker is actually placed *after* every document in our implementation.

or little room for misunderstanding, we will use the technically inaccurate, but highly understandable statements instead.

Building the suffix array is possible in optimal $O(n)$ time, achieved by [KS03], [KSPP03] and [KA03] independently of each other in 2003.

Searching the suffix array is not possible in optimal time. Indeed, because of the structure itself, the best possible search time is $O(w \log n)$. Using auxiliary data structures it is possible to achive $O(w + \log n)$ search time, and even expected $O(w)$ search time [MM91]. Which brings us to the original design purpose for suffix arrays: *low space requirements*.

When Manber and Myers introduced suffix arrays in 1991 [MM91], they built it in $O(n \log n)$ time using only $9n$ bytes, the text included, assuming an integer was 4 bytes. After construction, the text and suffix array use a mere $5n$ bytes, a major achievement over suffix trees, that use up to $20n$ bytes [Kur99].

2.5 Suffix trees

Suffix trees were invented by Weiner in 1973 [Wei73]. This has long given them a theoretical advantage over suffix arrays – much literature discussing construction of and searching in suffix trees has been published. Recently, however, suffix arrays has gained significant ground, with at least fifteen articles on pure suffix arrays the last four years, and maybe twice as many on closely related structures.

Nevertheless, the suffix tree remains unbeaten on its asymptotically optimal search time, never mind the fact that a suffix array performs comparably fast in practice [FG04]. The suffix tree can be built in $O(n)$ time and a pattern can be found in $O(w)$ time [Wei73].

There are two widely used method for suffix tree construction, that of McCreight 1976 [McC76] and that of Ukonnen 1995 [Ukk95]. They both end up with the same suffix tree, and how it is built is outside the scope of this article. I will just quickly assert the nature of a suffix tree, to disclose its close connection to the suffix array.

The data structure, as the name highly suggests, is laid out as a tree. The tree has n leaves (bottom-most nodes; nodes with no children), each corresponding to exactly one suffix in the text. All the inner nodes (those that are not leaves) of the tree each has at least two children. This ensures that there are at most $n - 1$ inner nodes.

A search consists of the following. You start in the root node. Each inner node has at least two edges out (there is exactly one edge per child). Each edge is labelled with a symbol. If your pattern exists in the text, there is an edge from the root node labelled with the first symbol in your pattern. Following this edge, you come to a new node. Repeat for the "next" symbol in your pattern. What the next symbol is, depends on the edge you have followed. In addition to being labelled with a symbol, each edge also has a length. This length is at least one. You discard as many symbols from your pattern as the length of the edge you follow. This way, you will reach the end of your pattern in w steps or less. As long as the edge labelled with the correct symbol can be found in constant time ($O(1)$), this procedure takes $O(w)$ time.

After following edges until your pattern is exhausted, you will have a node that points you to the position in the text where your pattern might be found. Since the edges may have lengths longer than one, but only one symbol, you must also match your pattern against the the text to make sure you have a match. This takes at most $O(w)$ time.

There are two key points to notice. We have left out tons of implementation decisions, meaning that the suffix tree is more difficult to implement than the suffix array. Another is that of occurence listing. If we want to know all occurences, we must traverse the tree rooted at the node we ended in. To explain this we will reveal the close connection to the suffix array, as promised. Each leaf node corresponds to a suffix in the text. If you traverse the leaf nodes from the leftmost to the rightmost, you encounter the suffixes in sorted order, granted that the edges are sorted. This sorted order is the same order in which a suffix array sorts the suffixes. Thus, a suffix array is nothing more than a left to right listing of the leaf nodes in a suffix tree.

Searching a suffix array will get more coverage in the next chapter, but we will give a brief introduction here. To find all occurences of a pattern, you search for the leftmost and righmost occurence. All entries within that bound in the suffix array, represents suffixes whose prefix match the pattern. When searching a suffix tree, you will end in an internal node (unless there is only one occurence of the pattern). All leaves below this node correspond to suffixes whose prefix match the pattern. The leftmost leafnode below the matched internal node correspond to the leftmost match in the suffix array and likewise for the rightmost leafnode below the matches internal node. The second key point is then that occurence listing is not as trivial as with a suffix array – you will actually have to traverse the entire subtree to find all the leaf nodes corresponding to occurences.

2.6 Memory model

We will assume the external memory model, described in [San02]. This depicts that there are M words of internal memory that can be accessed quickly, and an external memory that can only be accessed by using I/Os to move B contiguous words to internal memory.

There are two factors that might influence the model under some circumstances. The internal cache and the external cache. Internal cache is memory that is even faster than the M words of internal memory. One such cache is called the level 2 cache. Another is level 1 cache, but only the level 2 cache is big enough to affect our experiments. It is about 1 MB large or less. The external cache is located on the harddisk. When requesting the same data twice, this may hold them so that they can be retrieved without physically accessing the disk. This cache is generally 8 MB, and our harddisk is no exception.

Chapter 3

Search algorithms

The suffix array SA is explained in section 2.4. It consists of n number of pointers, sorted such that they correspond to the n sorted suffixes of a text T of length n.

Definition 3.1. The occurence listing problem is to return all the occ positions in a text T where a pattern P occurs. Formally, $Occ_T(P) = \{i|P = T_{i..i+|P|-1}\}$.

The occurence listing problem can be answered fast, using either a suffix array or a suffix tree. The suffix array method is explained in section 3.1. Techniques for searching a suffix array that is stored on disk are investigated in section 3.2. The suffix tree was briefly explained in section 2.5. Here is a quick overview of asymptotic running times in internal memory.

Brute force	$O(w \cdot n)$
Suffix array	$O(w \log n + occ)$
Suffix array w/ lcp	$O(w + \log n + occ)$
Suffix tree	$O(w + occ)$

Definition 3.2. The document listing problem is to return all the $dococc$ documents in a collection D of d documents where a pattern P occurs at least once. Formally, $Dococc_D(P) = j|P = D^j_{i..i+|P|-1}, 0 \leq i \leq |D^j| - |P|$.

The document listing problem can be solved by concatenating all the documents, separated by special markers, \$, into a text T such that $T = D^0\$D^0D^1\$..D^{d-1}\$$. Then we can solve the occurence listing problem for this text and translate text positions to documents indexes. That is the main focus of this thesis. Various solutions are discussed in section 3.3. Here is a quick overview of asymptotic running times in internal memory, assuming the occurence listing problem is solved in $O(w \log n + occ)$ time.

Direct	$O(w \log n + occ)$
Lookup	$O(w \log n + occ)$
Embedded	$O(w \log n + occ)$
Search	$O(w \log n + occ \log occ)$
Optimal	$O(w \log n + dococc)$

3.1 Internal suffix array search

The occurence listing problem (definition 3.1) was solved asymptotically optimal by Weiner in 1973 (see [Wei73]). In 1991, almost 20 years later, Manber and Myers was the first to solve it using the suboptimal suffix array technique [MM91]. The reason it was still of interest was the significantly lower space usage. At about the same time, Gonnet et. al [GBYS92] came up with the same solution, but their article does not focus as much on the search algorithm itself, and they did not descripe the LCP structure.

Manber and Myers explains three search routines in [MM91]. Two that solve the problem in $O(w \log n + occ)$ time and one that uses extra information, the LCP array, to solve it in $O(w + \log n + occ)$ time. Manber and Myers used the faster of the solution without LCP in their comparison with suffix trees. We will also use the solution without LCP information in our experiments. All three methods will be described in the following sections, but first we will look at how the suffix array is used to solve the occurence listing problem.

The suffix array SA represents the sorted suffixes of a text T. The lexicographically smallest suffix of T is $T_{SA[0]}$ and the lexicographically largest suffix of T is $T_{SA[n-1]}$. Generally, we have the ordering

$$T_{SA[i]} \leq T_{SA[i+i]}, 0 \leq i < n-1.$$

We exploit this by searching for suffixes whose prefix is P, $T_{SA[i]..SA[i]+w-1} = P$. As we will see, finding the smallest and largest such suffix, bounds the solution. What we will do is to perform a binary search to find the smallest and largest suffix matching P. The binary search does not return the suffixes, but pointers to the suffix array. Given these pointers, we can read all the pointers to all the suffixes directly, as will be shown soon.

Definition 3.3. $A \leq_j B$ means that the prefix of the j first symbols of A is lexicographically smaller than or equal to B. $A <_j B$, $A \geq_j B$, $A >_j B$ and $A =_j B$ are defined similarly. $A =_j B$ is simply shorthand for $A_{0..j-1} = B_{0..j-1}$.

Since the suffixes are lexicographically ordered by SA, they are also lexicographically ordered by their first j symbols. Thus, if $T_{SA[i]} =_j P, i \geq 1$ we know that $T_{SA[i-1]} \leq_w P$. Also, if $i < n-1$, $T_{SA[i+1]} \geq_w P$.

To find all suffixes whose prefix is P, we need only to find the lexicographically smallest and largest such suffix. The smallest is the one that occurs first in SA, $T_{SA[i]}, min_i(T_{SA[i]} =_w P)$. The largest is the one that occurs last in SA, $T_{SA[j]}, max_i(T_{SA[j]} =_w P)$.

If the pattern P exists in the text, it occurs $(j - i + 1)$ times. When we have found i and j, we know that all $T_{SA[k]} =_w P$ for all $k \in [i, j]$. The reason for this is that $T_{SA[i]} \leq_w T_{SA[k]} \leq_w T_{SA[j]}$. Since $T_{SA[i]} =_w P =_w T_{SA[j]}, T_{SA[k]} = w_P$.

3.1.1 Without LCP

Manber and Myers [MM91] give the algorithm for finding the index into the suffix array for the smallest matching suffix. Rewritten into our notation, it goes as follows:

```
1  if P ≤_w T_SA[0] then
2      return false
3  else if P >_w T_SA[n−1] then
4      return false
5  else
6      L := 0
7      R := n − 1
8      while R − L > 1 do
9          M := (L + R)/2
10         if P ≤_w T_SA[M] then
11             R := M
12         else
13             L := M
14     if P ≠_w TSA[R] then
15         return false
16     else
17         return R
```

This is a simple algorithm that runs in $O(w \log n)$ time. The $\log n$ factor comes from the while loop in line 8. R starts with a value of of $n-1$. Each iteration of the loop halves the value of $R - L$, since either R (line 11) or

L (line 13) is set to M, the middle of L and R. The most expensive operation in each iteration is the comparison in line 10. At most w symbol comparisons are done here in each iteration. Thus the running time of this binary search is $O(w \log n)$.

We must do a corresponding search to find the largest matching suffix. After that we can read the matching interval of the suffix array to retrieve the text occurences. This takes $O(occ)$ time, totalling $O(w \log n + occ)$ time.

It is possible to do better with respect to the comparisons in line 10. If $P =_l T_{SA[L]}$ and $P =_r T_{SA[R]}$, we know that $P =_{min(l,r)} T_{SA[M]}$. That means only $w - min(l, r)$ symbol comparisons in line 10. It is still possible for l or r to remain 0 throughout the entire binary search. In this case, we still need at most w symbol comparisons in line 10, so the asymptotic running time remains the same. This is the search algorithm we have used in our experiments.[1]

The search algorithm we used is basically like the following pseudo code. We first search for the largest matching suffix, then the smallest. Information from the first binary search is passed on to the next, so that we do not have to start from the beginning. This complicates the code a little, but it is still basically the same as the above. To make it clearer how many character comparisons are being made, we use a function to compute equality, lcp, defined as follows:

```
def lcp(A, B):
    ret := 0
    while ret < |A| and ret < |B| and A[ret] = B[ret] do
        ret := ret + 1
    return ret
```

Clearly, this function does at most $max(|A|, |B|)$ comparisons. The binary search can now be desribed as follows.

```
matched := false
L := 0
R := n − 1
l := lcp(P, T_SA[L])
r := lcp(P, T_SA[R])
if r = w then
    j := R
else if P[r] > T[SA[R] + r] then
    return false
else if l < w and P[l] < T[SA[L] + l] then
    return false
else
    while R − L > 1 do
        M := (L + R)/2
        if l ≤ r then
            m := l + lcp(P_l, T_SA[M]+l)
        else
            m := r + lcp(P_r, T_SA[M]+r)
        if m = w then
            if not matched then
                matched := true
                l_copy := l
                L_copy := L
                r_copy := m
                R_copy := M
            L := M
            l := m
        else if P[m] > T_SA[M]+m then
```

[1] Actually, we used something that is a little closer to the LCP approach, in order to make it easy to plug in any LCP information. It does not make for much of a running time difference, though.

```
            L := M
            l := m
        else
            R := M
            r := m
    if l < w then
        return false
    else
        j := L
        l := l_copy
        L := L_copy
        r := r_copy
        R := R_copy


if l = w then
    i := L
else
    while R − L > 1 do
        M := (L + R)/2
        if l ≤ r then
            m := l + lcp(P_l, T_{SA[M]+l})
        else
            m := r + lcp(P_r, T_{SA[M]+r})
        if m = w then
            R := M
            r := m
        else if P[m] > T_{SA[M]+m} then
            L := M
            l := m
        else
            R := M
            r := m
    i := R


return (i, j)
```

Even though this algorithm is not asymptotically better, it "significantly improves search speed" [MM91], according to the experiments of Manber and Myers. That matches our experience from [FG04].

Notice that the search for i picks up where the search for j became "ambiguous". When the first search finds the first match, we know that neither $T_{SA[L]}$ nor $T_{SA[R]}$ are matches. We also know that $T_{SA[M]}$ is a match. The search for j can continue with M as its new L-border. The search for i later continues with M as its R-border. This is the most efficient way to perform these two binary searches, since we use all the information we have from the first search in the next search.

Manber and Myers discuss and use an important speedup to this algorithm, namely precomputing answers for queries up to a certain length, say k. There are σ^k possible queries of length k. We convert the k first symbols of the query using a function $Integer(A)$ such that $Integer(A) \leq Integer(B)$ if and only if $A \leq B$.

If $x = Integer(P_{0..k-1})$, we can create a table $buck$, such that $buck[x] = i | min_i(x = Integer(T_{SA[i]..SA[i]+k-1}))$. With this table we can start searching with $L := buck[x]$ and $R := buck[x + 1]$. We have not implemented this.

3.1.2 With LCP

The LCP structure holds information about the length of the longest common prefixes between certain pairs of suffixes. Manber and Myers [MM91] devised an ingenious way to exploit this precomputed information when searching in a suffix array. However, they do not use it themselves in their experiments. Neither does anybody else aiming for speed and low space usage. Time saving tricks such as the $buck$ array (see previous section) costs less in terms of space and provide adequate speed. This was confirmed by us in [FG04]. Nevertheless, it holds theoretical interest as it provides asymptotically better search time for the suffix array. We will continue to a brief presentation of how the LCP structure can be used.

Assuming that we use a binary search such as in the previous section, always starting at the same place and always shrinking the borders in the same, predictable way, we can make an important observation. Any value for M has unique corresponding values for L and R. That is, there is only $n-2$ unique triplets (L, M, R) that can be encountered during the binary search. This means that it is feasible to precompute $lcp(T_{SA[L]}, T_{SA[M]})$ and $lcp(T_{SA[M]}, T_{SA[R]})$ for all possible combinations of L and M and M and R. These lcp values can be stored in arrays called $llcp$ and $rlcp$, each of size $n-2$. $llcp[M] = lcp(T_{SA[L]}, T_{SA[M]})$ and $rlcp[M] = lcp(T_{SA[M]}, T_{SA_[R]})$.

Using the information in $llcp$ and $rlcp$, we can perform the binary search in $O(w + \log n)$ time. The point is to avoid unnecessary comparisons by knowing whether the search shall proceed to left or right. The LCP arrays provide us with information about this, and we now proceed to prove how they reduce the number of comparisons.

Assuming that $l \geq r$, there are three cases to consider. This applies correspondingly for $r > l$.

Case 1: If $llcp[M] > l$ we know that l would have been greater if there was a match in the left half. Thus we must continue search in right half without question, l being unchanged.

Case 2: If $llcp[M] < l$ we know that there is no hope of a match in the right half and we must continue search in the left half, setting $r := llcp[M]$.

Case 3: If $llcp[M] = l$ we do not yet know where we must continue the search. We must calculate $j := lcp(P_l, T_{SA[M]+l})$. The search will continue in the left or right half depending on P_{l+j} compared to $T_{SA[M]+l+j}$. Now it is fortunate that $j \geq l \geq r$. This means that whether we set $l := l + j$ or $r := l + j$, the value will be larger than l, that was assumed the largest value before. Thus, we only have to compare $P_{l..l+j}$ and $T_{SA[M]+l..SA[M]+l+j}$ once. This means that the total number of comparisons will be less than or equal to $2w$, giving the run time of $O(w + \log n)$.

The pseudocode is almost equal to the improved search algorithm in the previous section. Simply replace

```
if l ≤ r then
    m := l + lcp(P_l, T_SA[M]+l)
else
    m := r + lcp(P_r, T_SA[M]+r)
```

with

```
if l ≥ r then
    if llcp[M] ≥ l then
        m := l + lcp(P_l, T_SA[M]+l)
    else
        m := llcp[M]
else
    if rlcp[M] ≥ l then
        m := r + lcp(P_r, T_SA[M]+r)
    else
        m := rlcp[M]
```

in both the search for the largest and smallest matching suffix. This pseudo code is taken from Manber and Myers' article. They merged cases 1 and 3. This gives us a constant extra number of comparisons to find that lcp returns 0 for case 1 in the inner if. We have presented their version to explain this, but we feel that it would be better to follow the three cases more to the letter in real code. Note also that there are unnecessary comparisons other places in the pseudo code, but that these also add nothing but a constant cost per iteration – we feel that having them there provides clarity for the pseudo code.

One thing that have not been mentioned yet, is how to create the LCP arrays ($llcp$ and $rlcp$), but that is out of the scope of this thesis. Suffice to say it can be done in $O(n)$ time as shown by Kasai et al. in [KLA$^+$01].

3.2 External suffix array search

Articles on searching in suffix arrays and texts that are stored on secondary memory are hard to come by. That is perhaps quite natural, since the more essential construction of suffix arrays that are too large to fit in internal memory, have only recently come under discussion To the best of our knowledge Dementiev et. al [DKMS05] were the first ones, in 2005, to provide feasible suffix array construction on disk (ignoring the thesis [Meh04] from which the article is clearly taking its content).

However, there is one decent exception. As early as 1996, Baeza-Yates et.al [BYBZ96] described, among other things, a structure to support searching PAT arrays[2] that was stored on secondary memory, the Short PAT (SPAT) array. We will now describe the SPAT array, before we look at some other techniques that may be applicable for searching suffix arrays that are stored on disk. Unfortunately, we have not had the time to implement any of the methods mentioned in this section.

3.2.1 SPAT

The SPAT array holds a "summary" of the external suffix array in internal memory. Assume that M internal memory is available. A SPAT entry consists of the prefix of some suffix. If the length of this prefix is p, we can fit $m = \frac{M}{p}$ SPAT array elements in internal memory. The SPAT array elements logically divides the suffix array into groups of size $g = \frac{n}{m}$. The prefix to be stored in the SPAT array is the prefix of the last suffix in each group.

The SPAT array is searched with a standard binary search, possible finding a left and right border for the match. If p is too small we will have to get text from disk during the SPAT array search. p must thus be balanced so that we get as many SPAT array entries as possible, while needing to fetch text from the disk as seldom as possible.

When the SPAT array binary search is done, we fetch the suffix array entries belonging to corresponding groups from disk. We must fetch suffix array entries from at most two groups. A binary search is performed in each group, this time the text have to be read from the disk.

3.2.2 Interpolation search

A normal search of the suffix array is explained in section 3.1.1. This search performs poorly when used directly on disk (see section 6.3.4).

An interpolation search, however, such as the one described in section 3.3.4 might be able to help out. A good interpolation search should be able to outperform the binary search easily, since it would rid itself of many disk accesses. Howerver, finding a good interpolation search algorithm is not trivial.

One of the first difficulties is figuring out how to do the interpolation. In an array with numbers, it is trivial to find an interpolation, using the ratio of the difference between the search key and the number at the left border and the

[2]PAT arrays is the same as suffix arrays.

difference between the numbers at both borders. In a suffix array, the suffixes that make up the keys can not be readily translated into numbers between which we can find differences and ratios.

We are unable to come up with a method of interpolating in a suffix array search. Deciding whether this is possible is future work.

Another issue is that using the like of the LCP search in section 3.1.2 would be futile. The LCP method of Manber and Myers depends on the search being predictable and yielding unique borders for each midpoint. An interpolation search does not have any of these attributes.

3.2.3 LCP

Using LCP information when searching externally would be more profitable than when searching in internal memory. The reason for that is that we can save many disk accesses, something that is much more worth than just a few comparisons.

However, in section 3.1.2 the LCP structure is described as two arrays $llcp$ and $rlcp$. Using this approach would lead to many more disk accesses. Changing the data structure is trivial, since each entry in $llcp$ and $rlcp$ corresponds to one entry in the suffix array, and is always used together with this. We would change the suffix array so that each SA entry holds three values. Let SA_{orig} be the original suffix array. Then

$SA[i].llcp = llcp[i]$,
$SA[i].textindex = SA_{orig}[i]$ and
$SA[i].rlcp = rlcp[i]$,
for $1 \leq i < n-1$. $SA[0].llcp$, $SA[0].rlcp$, $SA[n-1].llcp$ and $SA[n-1].rlcp$ are undefined.

We believe that this suffix array search method has substantial potential for speedup compared to not using LCP information.

3.2.4 Interleaved text

When the suffix array is so large that it must go in external memory, it is reasonable to believe that this applies to the text also. If the text can be held in internal memory, the search time might be as much as halved, because at least half the disk accesses are to the text.

The previous two sections try to contribute toward reducing the number of times it is necessary to look at the text. Another approach is to interleave the text with the suffix array, to be able to load both suffix array value and text on the same disk access. If we could load so much text that we do not need to go to the disk for more, this could halve the search time as well.

The method is straight forward, but it costs much in terms of space. For each suffix array entry, store the first k text symbols of the suffix the entry points to, along with the suffix array entry. Let SA_{orig} be the original suffix array. Each SA entry would then hold two values

$SA[i].textindex = SA_{orig}[i]$ and
$SA[i].suffixprefix = T_{SA_{orig}[i]..SA_{orig}[i]+k-1}$.

If we assume that a text index uses 4 bytes, and a text symbol uses 1 byte, we could answer all queries up to $w \leq k$ length with no extra disk accesses for text in $4+k$ bytes.

Longer queries would lead to disk accesses for the text when they match more than the first k symbols of the current suffix, so depending on the space available, need for speed and average query length, this might be a feasible solution. Especially considering that storage space is extremely low.

3.3 Document retrieval

The document listing problem (definition 3.2) was solved optimally by Muthukrishnan in 2002 [Mut02], using suffix trees. His solution works just as well for suffix arrays, with a running time of $O(w + \log n + dococc)$. The solution does not seem well fit for external memory, so we have not implemented it. It is, however, very interesting, and a short description is given in section 3.3.5.

We have not seen any articles that discuss practical solutions to the document listing problem using suffix arrays. Comparing such solutions is the main objective of this thesis. None of the solutions are pioneering work as such – they are rather quite intuitive. But this is, to the best of our knowledge, the first attempt to show how they must interact with a suffix array. The results in section 6.3 are the more interesting part, but this section should also bring some enlightenment to the possibilities of document listing using suffix arrays.

We will desribe four different methods in addition to Muthukrishnan's optimal method, *direct*, *lookup*, *embedded* and *search*. Embedded is split into *embedded* and *short embedded*, where the latter requires modification to the suffix array values. Search is split into *binary search* and *interpolation search*. Lookup and search work by translating text occurences into documents. Direct and embedded read the documents directly.

All methods deliver listings with *duplicate* documents. By duplicate documents we mean that a document will be listed as many times as there are occurences within the document. If this is not desirable we need to somehow filter away duplicates. There are two easy ways to implement techniques for this. One is to sort the documents, then walk through the sorted list and copy only those that occur once. The other is to have a list of marks of the same length as the number of documents. While reading the documents with duplicates, mark a document when it is reported, and do not report any previously marked document.

Common for all methods is that they depend on a table mapping a document into its position in the concatenated text, the $doctable$.

$$doctable[i] = \begin{cases} 0 & \text{if } i = 0 \\ |D^1| & \text{if } i = 1 \\ n & \text{if } i = d \\ |D^0 D^1 ... D^{d-1}| & \text{else} \end{cases}$$

Some methods use this table to build their search structures, while others use it directly when searching. An index into the $doctable$ correspond to a document, so to find the document corresponding to a suffix T_j in the concatenated text with the doctable, you must search for the index i such that $doctable[i] \leq j < doctable[i + 1]$. Notice that $doctable$ has $n + 1$ values, to simplify searches like the one just mentioned.

3.3.1 Direct

This method relies on a precomputed list of what document corresponds to each entry in the suffix array. If we have an array $direct$ of size n such that $direct[i] :=$ document where $SA[i]$ occurs, document listing is straightforward.

Say that we have an occurence in the text $k = SA[i]$. T_k belongs to the document $direct[i]$.
Given borders L and R for the results in SA, the corresponding documents, with duplicates, are simply $docs = direct[i] | i \in [L, R]$.

3.3.2 Lookup

This method relies on a alignment of the documents and a precomputed lookup array, $lookup$, corresponding to the alignment.

Definition 3.4. A document D^i is *aligned* on an alignment size, a, if the document length is a multiple of a. That is, $|D^i| \mod a = 0$.

Say that we have an occurence in the text $k = SA[i]$. Since the documents are aligned on a, T_k belongs to document $lookup[\frac{k}{a}]$.
Given borders L and R for the results in SA, the corresponding documents, with duplicates, are
$docs = lookup[\frac{SA[i]}{a}]|i \in [L, R]$.

3.3.3 Embedded

This method relies on the suffix array being modified. The short approach assumes that any combination of a document and a position within that document can be saved in the same value. Otherwise, the document and the positions within the text can be save interleaved in the suffix array.

There are at least two reasons for using the short approach. It consumes less space, and is more natural for a suffix array build of documents. Instead of the suffix arrays containing pointers into some artificially created text, they contain pointers to documents and offsets to the suffixes within. The problem is that it is often more practical to do the actual merging into a single text. If that is the case, this method loses much of its value. Another problem is that of the number of documents and their length. If a suffix array value originally is 4 bytes long, it is reasonable to give document pointer 2 bytes and the offset the other 2 bytes. This limits both the number of documents and their maximum size to about 65,000.

We will now introduce a new notation for suffix array values. $SA[i]$ is a value in the suffix array. Normally, this is an index into the text. With the embedded information, $SA[i]$ holds two values, $SA[i].offset$ and $SA[i].docid$. The latter is obviously a pointer to a document. The former depends on whether we are using the short or normal approach. With the short approach $SA[i].offset$ is an offset into the corresponding document. With the normal approach $SA[i].offset$ is an index into the text, just as normally denoted by $SA[i]$.

If we want to use the short approach, and have a single text, we need to be able to translate the document pointer and offset into a text index. Given a pair of values $SA[i]$, the text index $t = doctable[SA[i].docid] + SA[i].offset$. This has nothing to do with the document listing problem, but it affects the suffix array search needed to find L and R.

For both the short and the normal approach, given borders L and R for the results in SA, the corresponding documents, with duplicates, are $docs = SA[i].docid|i \in [L, R]$.

3.3.4 Search

These methods only rely on the $doctable$. All other methods rely on one of these for the construction of their search structure. The embedded method is the only exception – provided that the suffix array is built with document pointers embedded.

Assume that we have some search method, $search$ that takes as its argument an index into the text and returns the corresponding document. Pseudo code for two different such $search$ methods are given below. Here is how to solve the document listing problem using the search method.

Say that we have an occurence in the text $k = SA[i]$. Since the documents are aligned on a, T_k belongs to document $lookup[\frac{k}{a}]$.
Given borders L and R for the results in SA, the corresponding documents, with duplicates, are
$docs = lookup[\frac{SA[i]}{a}]|i \in [L, R]]$.

Say that we have an occurence in the text $k = SA[i]$. T_k belongs to document $search(k)$.
Given borders L and R for the results in SA, the corresponding documents, with duplicates, are
$docs = search(SA[i])|i \in [L, R]$.

Binary search

A binary search for the document in the $doctable$ is almost straightforward. The only thing we must change from any ordinary binary search is that we are not searching for an exact key, we are searching for the lowest index giving a key that is smaller than what we are searching for.

Say that the input to the search algorithm is k. We want to find $i : min_i(doctable[i] \leq k)$. Recall that $doctable[d] = n$, so that this is the same as finding $i : doctable[i] \leq k < doctable[i+1]$. Having the extra element $doctable[d]$ requires only one extra element in $doctable$, but greatly simplifies our algorithm.

```
def search(k):
    L := 0
    R := d
    while R − L > 1 do
        M := (L + R)/2
        if k ≥ doctable[M] then
            L := M
        else
            R := M
    return L
```

Interpolation search

The assumption that was made for $doctable[d]$ in binary search applies here as well.

An interpolation search is very similiar to a binary search. They both use borders, find some point between the borders, and then move one of the borders to that point depending on its value. The difference is that while a binary search always selects the point in the middle of the two borders, an interpolation search attempts to guess which point will be closer to the value we are seeking.

The calculation that must be performed in order to select a new point speaks against interpolation search. However, if the values are approximately uniformely spread out in the array, an interpolation search can find the correct value in much fewer steps than a binary search. And the larger the array, the more the chance of the interpolation search severely beating the suffix array. Since we are dealing with large arrays, the interpolation search ought to be a good competitor to the binary search.

```
def search(k):
    L := 0
    R := 0
    while k > doctable[L] and k ≤ doctable[R] do
        M := L + (R − L) · (k − doctable[L])/(doctable[R] − doctable[L])
        if k ≥ doctable[M] then
            L = M + 1
        else
            R = M − 1
    if k < doctable[L] then
        return L + 1
    else if k > doctable[R] then
        return R
    else
        return L
```

3.3.5 Optimal

The time optimal solution of the document retrieval problem use $O(m + dococc)$ time. This can be achieved using a suffix tree together with a $O(1)$ solution to the range minimum query problem. The suffix tree must be preprocessed in $O(n)$ time. After finding the left and right border nodes of the range of leaf nodes that make up the occurences, the algorithm can proceed exactly equally on a suffix array and a suffix tree. Since finding the corresponding suffix array borders take $O(m + \log n)$ time, the document retrieval problem can be solved in $O(m+\log n+dococc)$ time, which is still asymptotically better than $O(m+\log n+\Omega(occ))$ algorithms that rely on traversing all occurences in the suffix array.

The following solution was presented in [Mut02] for suffix trees. Adaption to the suffix array is trivial.

The method is best explained intuitively, using figure 3.1.

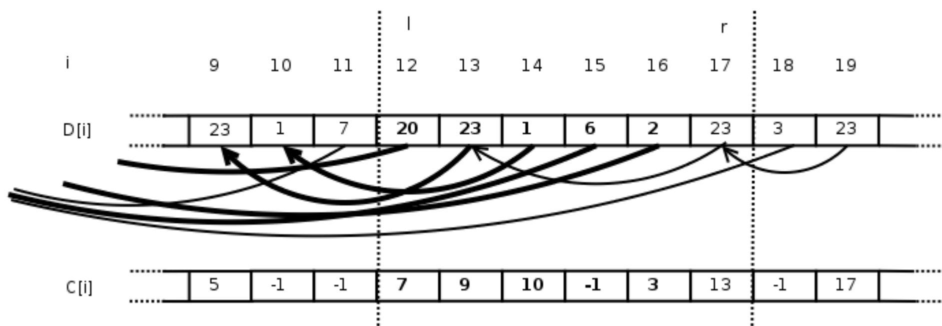

Figure 3.1: Document retrieval tables D and C.

The topmost range of numbers represent indexes of a suffix array. We have performed a search in the suffix array and found the borders for our result l and r. D is in array with the document pointers corresponding to the suffix array entries. D is the same as the $direct$ array of the direct method described in section 3.3.1. Now we want to retrieve the unique values from $D_{l..r}$.

Consider another array C that chains equal document pointers together (see figure). The values in C in figure are the index of the entries being pointed to by the arrows emerging from the corresponding D indexes. That sentence was complicated, but the figure should be pretty intuitive. If there is no previous equal document pointer, the C entry is set to -1.

Note the bold arrowed lines in the figure. If you list the document pointers they come from, you get all the unique document pointers. The common trait of the bold arrowed lines is that they all cross the l border. Look at the C values corresponding to the bold arrows. They are all less then l. That is the key observation.

If you ask for the minimum value in the range $C_{l..r}$, you are guaranteed to get a value that is lower than l. Imagine you get the index of this value, j. In the figure $j = 15$ and $C[j] = -1$. Proceed with asking for the minimum values in the intervals $C_{l..j-1}$ and $C_{j+1..r}$. If the minimum value is less than l you have found a new document that shall be reported. Continue searching on both sides until the interval is empty or the minimum value is greater than or equal to l. When you are done you will have reported all documents without duplicates.

Finding the minimum value can be done in constant time, given some preprocessing, see appendix A. Thus you can find all unique documents in optimal $O(dococc)$ time.

The preprocessing, however, is not fit for doing in external memory.

Chapter 4

Construction algorithms

4.1 Document retrieval

We assume that the $doctable$[1] is existing as input. Further, we assume that if the embedded method is used, the document information is already embedded[2] into the suffix array.

With these assumptions, only the direct and lookup methods require preprocessing. The construction of these are explained in the following sections.

4.1.1 Direct

To facilitate direct reading of the document pointers corresponding to an interval in the suffix array, we create an array $direct$ of size n, such that every element $direct[i]$ represents the document of the suffix starting at $SA[i]$.

$doctable$ and SA will be used to create $direct$. To find the index in $doctable$ corresponding to a SA value, we will use the $search$ method as defined in section 3.3.4. Given an index into the text, it returns the corresponding document.

We are now ready to create every element in $direct$, and the formula is really simple: $direct[i] := search(SA[i]), 0 \leq i < n$.

4.1.2 Lookup

To facilitate a table available for looking up the document corresponding to a text index, we can create a table $lookup$ of size n. However, we can do better if we know that documents are aligned[3] on some alignment size a. Thus we create a table $lookup$ of size $\frac{n}{a}$.

We will only need $doctable$ to create $lookup$. Our goal is that $lookup[\frac{i}{a}] = search(SA[i])$, where $search$ is the same function as in section 4.1.1. It is, however, rather unnecessary to use $search$ to create $lookup$.

What we want to do is to traverse $doctable$. For each value $doctable[i]$, $i < n$, we calculate the document length $doclen_i := doctable[i+1] - doctable[i]$. Then we simply set all $lookup[j] = i$, $\frac{doctable[i]}{a} \leq j < \frac{doclen_i}{a}$, and $doctable$ is ready.

[1] See section 3.3 for a brief definition, or section 6.2.2 for how it is made.
[2] See section 5.4.2 for the implementation details when artificially creating the suffix array with embedded information
[3] Aligned on a means that the size is a multiple of a

Chapter 5

Implementation

The source code can be found on the attached CD.

There were four major tasks in the implementation,

1. Suffix array construction
2. Suffix arary search
3. Document information construction
4. Document retrieval

The implementations of these are described in the last section of this chapter.

Crucial to the performance of our application, is total control of the disk. The operating system's general mechanisms for optimizing disk usage does not apply very well to many special algorithms – such as suffix array construction and searching. The author of the implementation of the algorithm is more likely to know how to make optimal use of the disk. However, optimal disk usage is difficult, and we have been nowhere near achieving it. Optimal usage would require threading of the disk accesses, something which would complicate the program too much for our time frame. Thus, the program could be enhanced later by adding threaded disk access.

In the design to support the above task with full control of the disk we decided to create simple classes that could deal with direct disk access. Because direct disk access requires that the internal memory is aligned, it was decided to create classes that manage internal memory also. To make it general, even the default memory handlers of C++, `new` and `delete` were overloaded. Later, this proved to be a bad design decision, as overloading `new` and `delete` lead to no end of trouble, see section 5.1.3.

How statistics is gathered from inside the main program and how it is later processed is described in section 5.2. All statistics are generated by our own source code, because measures must be done on points where external programs cannot do them. For instance, the disk usage is counted seperately for the suffix array search and the document listing, even though they occur interchangable as all queries are processed within the program.

Finally, we ran a test to see how the disk responded to different number of block reads. We found that a value of 256 blocks (128 KB) per read was a good value, as shown in section 6.3.1. Unless there was a reason to, no disk read ever read less than 256 blocks at once during our experiment.

5.1 Memory management

To accomodate using the disk directly (see section 5.1.2), we created classes to deal with the details. This resulted in a memory class hierarchy.

```
Memory                          Abstract base class
    MemoryInternal              Default internal memory (default new and delete)
        MemoryAligned           Internal memory aligned on block borders
    MemoryDisk                  Abstract disk class - provides aligned operations
        MemoryFile              Open file for direct access
            MemoryFileCache     Open file for cached access, overloading MemoryDisk
        MemoryPart              Open partition for direct access
```

The memory hierarchy did not become as transparent as we had hoped. It works, but there were some flawed design decisions on the way that would have been revised if we had had time. In addition to the memory hierarchy, we created a helper class `MemHolder` to facilitate more transparent access to disk memory. A `MemHolder` is bound to a `MemoryDisk` and acts as an array that reads in blocks when requested. How many blocks should be read into a `MemHolder` at once is decided by the user. Normally this will be the value mentioned above, 256.

The disk operations depend heavily on the operating system. As covered in section 6.1, the implementation for the experiments was done in C++ compiled with GNU's g++ 3.4 on Linux 2.6.10 running on an AMD Athlon64. We will now address some issues when doing low level disk operations in Linux 2.6. After that we will discuss some of the obstacles when messing with the memory mechanisms of C++.

5.1.1 mlock

Linux uses a very agressive disk caching strategy by default. So aggressive, in fact, that it might start swapping out some processes's pages in order to preserve and expand the disk cache. `mlock` is a system call that locks a range of your process's pages in memory, so that they are guaranteed not to be swapped out to disk. `mlockall` is another system call that does the same for all your pages (optionally, both current and future ones).

So after a call to `mlockall`, we are guaranteed control of what remains in memory. From Linux 2.6 onward, a command line call to `sysctl -w vm.swappiness=0` will prevent most swapping from taking place. We only recently became aware of this option, but we will still prefer the lock calls as that gives us certainty.

If a process wants to use the system call `mlockall`, it has to have privileges to do so. Privilege assignment changed in Linux 2.6.9. (And in the 2.4 series, only up to half of the memory could be locked.) So you want to make a call to test whether you are privileged. The documented way of doing this (a library call to `cap_get_proc`) only worked when running programs as the super user, root. We abandoned trying to check whether we had the privilege.

Another more important issue, is that of requesting that too much memory be locked. The library call `malloc` is used to request memory. When requesting memory in Linux, the default is that the Linux kernel is optimistic and might promise more memory than can actually be delivered (see `man malloc`). This is fine when swapping can be applied. When we tried to lock too much memory, we expected the lock call to return an error value. Instead it actually locked too much memory. This, of course, took up all available memory, and the OOM (out-of-memory) killer of the Linux kernel started killing of all process *except* ours, that was the one doing the actual harm. When all but our process had been killed, the kernel panicked (the machine stopped responding) because it had no memory. To avoid this, run the command line call `sysctl -w vm.overcommit_memory=2`, which will force the kernel to check that enough memory is actually before committing to give you memory.

5.1.2 Disk access

After making sure that our process is not swapped out, we went on to decide how to read from and write to the disk. The library call `fopen` opens a (cached) stream. The system call `open` opens a file or device. We were doing low level operations, so streams were out of the question. What remained was what `open` should open.

Linux has two kinds of devices, *character* and *block* devices. The former may be used for terminals and the latter may be used for disks. The difference is that character devices are unbuffered while block devices buffer blocks before they are read from / while they are written to. Disks are block devices by nature. Most disks are divided into "physical" sectors (blocks) of size 512. When using such devices, blocks of size 512 would be read into a buffer, before being copied to our process's memory.

The traditional way to avoid this, was to use the system tool `raw` to bind a "raw" character version of the disk to a block device. When opening this block device, the usual cache would be bypassed under the restriction that any reads and writes had to be

1. *aligned* on a block border, both in memory and on disk, and
2. their length had to be a multiple of the block size.

Alignment on a (block) border means that the address must be a multiple of the alignment size (block size). Given an unaligned address, $addr_u$, we can find the nearest aligned address ,$addr_a \leq addr_u$ as follows

$$addr_a = addr_u \& \tilde{(}ALIGN_SIZE - 1),$$

if $ALIGN_SIZE > 0$ is a multiple of 2, $\&$ is the bitwise and operator, and ĩs the bitwise negation operator.

The raw drivers have been made deprecated and obsolete[1] It is recommended to use the block devices as usual, and open them with the `O_DIRECT` flag. The manual page for `open` explains that if you use the `O_DIRECT` flag, the kernel will "[t]ry to minimize cache effects of the I/O to and from [the] file". When using the `O_DIRECT` flag, we operate under the same two restrictions as when using raw devices, with some minor differences. In the 2.4 kernel series, the alignment size is that of the logical block size of the file system. Under the 2.6 series, alignment size is 512. The documentation does not say, but this implies that the 2.6 series might do buffering when the actual block size is not 512. Most disks block sizes, however, are 512, so we choose to go with this alignment size for disk operations.

One more implementation difficulty should be mentioned, namely that of alignment of the file size. The size of a random input text is probably not a multiple of 512. To accommodate such input, we preprocessed all input files. We added at least 512 bytes at the end, where the last 8 bytes held the original filesize. This meant garbling the files with respect to other applications, so a neccessary assumption is that we had sole rights to the files, something we had. Reading files whose sizes are not multiples of 512 is clearly possible - the file system does it all the time. We, however, had neither need nor wish to find out how to do this in `O_DIRECT` mode, so we chose to use our simple solution. It should not affect our results in any way.

Another thing that should not affect our results, but goes to show that the `O_DIRECT` mode's implementation in Linux might not be wholly thrustable, is the following quote from Linus Thorvalds, creator of the Linux kernel (taken from `man open`):

> The thing that has always disturbed me about `O_DIRECT` is that the whole interface is just stupid, and was probably designed by a deranged monkey on some serious mind-controlling substances.

5.1.3 new

C++ lets you do the most incredible things. One of those things are overloading the default memory allocation and deallocation routines, `new` and `delete`. We use an instance of `MemoryInternal` to handle calls to `new` and

[1]See Device Drivers -> Character Devices -> RAW driver -> Help in the Linux 2.6.10 kernel compilation configuration.

delete. This has two important problems. First, the MemoryInternal instance must be instansiated before the first attempt to use it. Static initalization comes to mind. Static objects are created at the beginning and last throughout the program. However C++ makes no guarantee about the initialization order of various static objects in different compilaton objects (different cpp-files). This meant the tedious use of a function to get the default MemoryInternal instance whenever it was needed.

The second problem is more important – the instance must not be deleted before everything that shall be deallocated with it has been deallocated. We ran into serious problems with this, and we did not manage to correct all of them. Because of the problems with new and delete, a simple class that was used with static objects to register different methods and their command line paramaters had to be discarded. The program crashed without us being able to find the reason. Debugging a program before it has come to the starting point (the main function) is difficult.

5.2 Result gathering

The memory hierarchy desribed in section 5.1 is used to keep count of allocated memory, maximum allocated memory, number of disk accesses, size of disk traffic and similar measures.

Timing is done with the gettimeofday system call, which yields an overhead of 0.1 microseconds per pause of the timer. This is described in section 5.2.1.

Both the memory classes and the timing class (Clock) print their own measures. In addition to that, the suffix array related methods prints such measures as text size, number of documents, number of queries and average query length. All this output is written to stdout in a uniform format and gathered using a python script, see section 5.2.2.

5.2.1 Timing

To measure the time usage in our experiments, we used the gettimeofday system call. Using the system tool time was out of the question, since our program performed many duties that we did not want to include in the timing. The overhead of calls to gettimeoday is small[2].

As can be seen by the code in code/sa/clock.h our clock's timing functions are inlined, so that the maximum error

- for the start function is from the point gettimeofday reads the time until it returns and the return value is checked, and
- for the pause/end function is from the point gettimeofday is called to the point where it reads the time.

For each pair of calls to our clock's start and pause/end function we add to our results the time it takes for one call to gettimeofday to complete and for its return value to be checked. This overhead is about 0.1 micresecond (0.099 microsecond when the calls are inlined, 0.102 microsecond when they're not). This is mostly ignorable.

5.2.2 Scripts

code/tools/gen_data.py has information about what test data to create and what tests to run on these data. It does this automatically and logs the gathered output to a file.

[2]A short article discussing how to measure time on Linux and Windows is to be found at IBM, http://www-106.ibm.com/developerworks/library/l-rt1/. The results are for 2.2.16 and 2.4.2, but we have no reason to believe that the call has any greater overhead in 2.6.10

code/tools/gen_results.py takes as input a test case and the results file. The test case describes what data we are interested in, for what methods, and can give commands as to how the data shall be plotted. gen_results.py automatically generates graphs when given a test case and the result file. Some of the data specified in the test case files is overlapping with information in gen_data.py, so that the two have to be synchronized.

5.3 Disk test

This test aims to see how many blocks we can read without paying extra for it. The implementation in code/sa/test_disk.cpp solves this as follows:

1. Allocate and lock almost all memory to make sure that Linux does not provide cache.
2. Make sure the disk cache is cleared by reading 8 MB sequentially and 1 KB 500 times with 1 MB between each read, from a dummy file
3. Read X number of blocks from the source file 100 times, skipping 10 MB (disk cache is 8 MB) between each read.
4. Increase X.
5. If X is not sufficiently large, go to 2.

When we tried reading the same disk area, it was clear that we got disk cache. When we tried cached reading and not locking the memory, it was clear that Linux cached for us. Since we experienced none of this conditions during normal runs, we think that it is safe to say that our implementation stopped any significant impact on the results from caching.

5.4 Suffix array construction

We used only one suffix array construction algorithm, DC3, as provided in [DKMS05]. From the suffix array created by DC3, we created the suffix arrays with embedded document information, as described in section 5.4.2.

5.4.1 DC3

DC3 is a suffix array construction algorithm for external memory developed by Jens Mehnert. It is available from http://i10www.ira.uka.de/dementiev/esuffix/docu/index.html. DC3 depends on STXXL, a library mimicking the C++ Standard Library's behaviour, but with the data structures on disk. It is available from http://i10www.ira.uka.de/dementiev/stxxl.shtml. STXXL was given 120 GB space split in 60 GB on two disks.

The DC3 source code was modified to use 2 GB of memory.

5.4.2 Embedded

This implementation can be found in code/sa/sa_pure.cpp.

The embedded method assumes that the suffix array has been built with the document information already embedded. Since we use an unmodified version of the DC3 algorithm, this is not the case. However, we pretend that it is

the case, so the construction of the embedded suffix array is not reported, only performed in order to support the search in embedded arrays.

Let the $search$ be as defined in section 3.3.4. Recall that it returns the document corresponding to the text index it takes as argument. For the new suffix array values, we used the notation introduced in section 3.3.3, $SA[i].docid$ and $SA[i].offset$. The source suffix array values are denoted $SA_{orig}[i]$.

The construction of the embedded array is as follows.
For each $SA_{orig}[i], 0 \leq i < n$, we set $SA[i].docid := search(SA_{orig}[i])$.
Then we copy the text index $SA[i].offset := SA_{orig}[i]$.

The construction of the short embedded array is only slightly different.
For each $SA_{orig}[i], 0 \leq i < n$, we still set $SA[i].docid := search(SA_{orig}[i])$.
Then we calculate the document offset $SA[i].offset := SA_{orig}[i] - doctable[SA[i].docid]$.

5.5 Suffix array search

This implementation can be found in `code/sa/sa_search_template.h`, instantiated from `code/sa/sa_pure.cpp`.

The implementation of the suffix array corresponds approximately to the full search pseudo code in section 3.1.1, based on Manber and Myers $O(w \log n)$ algorithm.

There are three exceptions. The first is a minor one. We use macros to make usage of LCP information possible. To avoid unnecessary complications to the code, we mimic the behaviour of the LCP changes in section 3.1.2, using the lcp function call instead of $llcp$ and $rlcp$ table lookups. This will lead to some more comparisons, but this method was comparable to suffix trees when used in [FG04]. Since comparisons might be more expensive on disk, though, this should have been eliminated an it is a weakness of the code such as it is.

The second exception only affects the code when the suffix array has short embedded document information, like described in section 3.3.3. When such information is present, all requests for $SA[i]$ is rewritten to requests for $doctable[SA[i].docid] + SA[i].offset$. This is done with a macro so as not to affect the running time in other cases.

The third is the most obvious and influent change. We do not assume that the suffix array or the text fit in internal memory. This means that every request for $T[i]$ or $SA[i]$ must be retrieved from disk or cache. This is achieved with the help of four `MemHolders`. Three to hold text and one to hold suffix array values. This should almost minimize the need to refetch things from disk as follows.

First $SA[n-1]$ is fetched. Then $SA[0]$ is fetched. These cannot possible be read at once with a large text, without significant overhead. Then we read $SA[M]$, which is in the middle of the two previous. It will not be in the cache of either. In fact, $SA[M]$ has no chance of having been read before until the interval is sufficiently small. When the right border search finds a match, the SA values are stored in temporary variables so that the left border search do not have to reread them.

When it comes to the text, we might have done a better job caching if two texts accidentily are from approximately the same place. We think this is an unlikely event, and our text `MemHolders` work to prevent losing the cache as follows. We have one holder for each of the suffixes starting at $SA[L]$, $SA[M]$ and $SA[R]$. When moving to search in the right half the cache for the $SA[M]$ suffix is used for the new $SA[L]$. When moving to the left half, the $SA[M]$ suffix cache is used for the new $SA[R]$. This scheme works to preserve the text we know we might need. A more general text caching scheme might be better since the new $SA[M]$ can point to any suffix, also one close to another one whose cache was just overwritten. If it is likely that the same substring occurs often within the same document, there might be room for much improvement on the text cache in the suffix array search.

Lastly, to make the search time depending on the number of occurences, all occurences are read from the suffix array to memory in one read.

5.6 Document information construction

Here we will show how we have implemented the methods described in section 4.1.

The $doctable$, described in sections 3.3 (concept) and 6.2.2 (implementation) is assumed to be loaded in internal memory. The construction methods are given 300 MB of internal memory.

5.6.1 Direct

These are the implementation details for the method described in section 4.1.1. The source code can be found in `code/sa/doc_direct.cpp`.

To find the document corresponding to a suffix array value, we use the binary search described in section 3.3.4 as our $search$ method.

The suffix arrays are read in blocks as large as the memory limit allows, while still leaving room for the block to which we will write the corresponding document pointers. For each block we simply calculate $direct[i] := search(SA[i])$ for the i values inside the block and save the block to disk.

5.6.2 Lookup

These are the implementation details for the method described in section 4.1.2. The source code can be found in `code/sa/doc_direct.cpp`.

Given the method, this implementation is really simple. We allocate an array as large as the memory limit allows. Then we start filling it using $doctable$. Once the temporary array is full, we write it to a file with a filename that depends on the alignment size parameter, so that we can test varying fictional document alignments on the same data.

5.7 Document retrieval

Here we will desribe how we implemented the methods in section 3.3.

The $doctable$, described in sections 3.3 (concept) and 6.2.2 (implementation) is assumed to be loaded in internal memory. The construction methods are given 300 MB of internal memory.

Further, the hits from the suffix array are already loaded into memory, as described in section 5.5. This makes the implementation of search and embedded so trivial that we will not mention them further here. The implementations of direct and lookup call for some comments, though. The technique used to remove duplicates, if required, is discussed in section 5.7.3.

5.7.1 Direct

These are the implementation details for the method described in section 3.3.1. The source code can be found in `code/sa/doc_direct.cpp`.

We do not assume that the $direct$ table can fit in memory, because it is most likely just as large as the suffix array. However, we can find all the documents by doing just one long read of the interval corresponding to the interval of the suffix array.

The fact that we read all the results in one chunk, makes this method different from the others. All other methods convert one and one text index into a document pointer. Thus, they can apply the removal of duplicates while they are processing anyway (see section 5.7.3). This tempts us to try an alternate duplicate removal scheme for this method; using the `sort` method of the C++ Standard Library to sort the document pointers, and then traverse them and trivially ignore the duplicate entries while copying to the final array.

5.7.2 Lookup

These are the implementation details for the method described in section 3.3.2. The source code can be found in `code/sa/doc_lookup.cpp`.

When preparing for the search, it is checked whether we have been granted enough memory to hold the $lookup$ table entirely in memory. This depends on the alignment size, which regulates the size of $lookup$. The default alignment size is 32 and the corresponding $lookup$ is small enough to be held in memory. Computing the answer when the $lookup$ table is held in memory is trivial.

When trying to vary the alignment size (section 6.3.3), however, the $lookup$ table may be too large to fit in memory. This is solved by having a `MemHolder` hold as much as possible. The result from the suffix array is in memory. This is sorted using the `sort` method in the C++ Standard Library. Sorting these results avoids random access to the $lookup$ table. All lookups will now be sequential. Thus, we can read one chunk of the $lookup$ table and process all the text indexes within that chunk before moving to the next chunk.

Reading as much as possible from the disk in every chunk is a waste if the text indexes are very spread out. Analyzing the suffix array results and deciding on how much of $lookup$ table to read based on how far the text indexes are from each other, would be a better approach. Unfortunately, we have not had the time to try this.

5.7.3 Removing duplicates

Section 5.7.1 describes how the direct method can use `sort` to remove duplicates. Another duplicate removal method, as mentioned in section 3.3, is to maintain a list of which documents have already been found.

The list is assumed to fit in internal memory. It occupies only $\frac{d}{8}$ bytes, because we use a `vector<bool>` to hold it. Each element in a `vector<bool>` needs only 1 bit ($\frac{1}{8}$ byte) to represent true or false. The structure is called doc_found.

When we detect a document pointer i we check whether $doc_found[i]$ is set. If it is not, we set it and add the document to the result, else, we simply ignore the document – it is a duplicate. After traversing all the potential results, we need to clear doc_found for the next search. This is done with a simple traversal of the results, unsetting $doc_found[i]$ for each result i.

Chapter 6

Experiments

6.1 Machine configuration

Briefly described, our hardware consisted of an AMD 64 bit CPU running at 2.2 GHz, 4 GB internal memory, one 200 GB disk and one, faster 75 GB disk. Details follow below. The operating system of choice was Debian GNU/Linux AMD Athlon64. The slower disk was the one mainly used both for operating system and testing, because of its size.

The machine was ordered with this thesis and the thesis of Nils Grimsmo in mind. However, it had to be ordered earlier than our goals could be clearly established. We had some problems because of this and our lack of previous experience with the 64 bit architecture and such amounts of memory. The 64 bit architecture was picked in spite of our lack of experience because we wanted native support for very long addresses. The issues we encountered will be discussed in brief since they might be of interest – it soon became clear they were not common knowledge.

First the disks. They should have been selected with better care to rule out uncertainty in the experiments. We ended up not doing any serious measuring of suffix array construction, but the suffix arrays still had to be constructed. The faster way to do this, is to use parallel disks. Four should be sufficient [DKMS05]. We used both disks to speed creation. STXXL, a disk data structure library used by the DC3 suffix array construction algorithm, was given 60 GB of work space on each disk. The disks should have been working at the same speed, if we really wanted to measure the construction time.

Second, the operating system. Debian's AMD64 release is not official, and finding the right place to download your packages can be cumbersome. Much documentation was outdated, and we had to attempt installation four times before we found a configuration that worked. The C/C++ compiler played a vital part in our difficulties. The configuration that worked was "unstable" located at `http://debian-amd64.alioth.debian.org/debian-pure64`. It shipped with g++ 3.4 (not 3.3). Expect this to change as g++ 4.0 has been released. Also, interest for AMD64 is picking up, so the release ought to be official soon. Debian developers are not known for releasing things soon, though.

6.1.1 Details

Details about our machine configuration – both hardware and software – are listed in table below.

Part	Description	Technical info
Motherboard	MSI K8N Neo2 PLATINUM (MS-7025) http://msicomputer.co.uk/Products.aspx?product_id=703505&cat_id=77	Socket 939
CPU	AMD Athlon^{TM} 64 3500+	2210 MHz
Memory	4 * 1 GB DDR400	PC3200
Disk #1	200 GB Western Digital Caviar (WD2000JD-19H) http://www.wdc.com/en/products/Products.asp?DriveID=58	7200 RPM, 8 MB cache
Disk #2	75 GB Western Digital Raptor (WD740GD-00FL) http://www.wdc.com/en/products/Products.asp?DriveID=65	10000 RPM, 8 MB cache
OS	Debian AMD64	Linux 2.6.10
Compiler	g++ (GCC) 3.4.4	Debian 3.4.3-13

Both disk #1 and disk #2 were used during suffix array construction. For all other tests, only disk #1 was used.

6.2 Input

The disk speed test used some gigabytes worth of files concatenated into one file that was duplicated. One copy to clear the disk cache, and another to do the test.

The other tests depend on four different kinds of inputs. The text and the suffix array is one; the suffix array is assumed to exists. The text is further explained in section 6.2.1. Section 6.2.2 describes the `doctable` – this describes document borders. Section 6.2.3 describes the queries that are used when searching. The last input is generated in the construction phase for the direct and lookup methods.

6.2.1 Text

Input for the suffix array construction was somewhat limited in range. For most of the tests, only one source file was used, the concatenation of source code used in [DKMS05], called `source`. For varying text length, applicable only in section 6.3.4, files consisting of pseudorandom words were used, called `randomXX` where `XX` gives the filesize in MB. The pseudorandom words had the same length distribution as the King James Bible, that is why the average world length is equal for all `randomXX` files. To stress test the suffix array construction algorithm a little, we used the human genome, `genome`, also from [DKMS05]. This was only used for suffix array construction testing.

Input	Size (MB)	# words	Avr. word len	Alphabet size
`genome`	2927.9	N/A	N/A	5
`source`	522.1	62116060	7.88	246
`random20`	20	3842891	5.46	60
`random30`	30	5763381	5.46	60
`random40`	40	7683562	5.46	60
`random50`	50	9607381	5.46	60

We believe that using only `source` is sufficient because our tests depend on the number of results. This text has so much repetition that we can vary our number of results by varying the number of documents we split the text into, and the query length.

When searching in suffix arrays, the average *lcp* value is an important attribute of the text being searched. This should have been included in the table above, but we did not have the time and means to provide them. Fortunately, our experimental focus on suffix array search is limited, so the average *lcp* values of the texts are not so important to us.

6.2.2 Documents

As section 3.3 asserts, a `doctable` is used by all document retrieval methods. This makes the `doctable` files an important input in our experiments, since our main focus is on the behaviour of different document retrieval methods.

Section 6.2.1 lists five input texts. `source` is a concatenation of several small files by an external entity, so we have no influence on the concatenation format. The `randomXX` files, however, are concatenated by us from several documents with certain length restrictions.

Between each document we inserted a special marker ('\0') so that patterns should not match accross document borders. These text files have what we call real document borders. Each have a corresponding `doctable` file that lists the start position of each document in the text.

`source`, however, has no predefined document borders. We used the data generator to create what we call artificial document borders. `source` has several different `doctable` files, each corresponding to different length restrictions on the artificial documents. Since `source` does not have special markers between the artificial documents, search patterns may cross document borders. We do not think this has any visible effect on our results.

Here is a list of the properties of the five different `doctable` files we are using with `source`.

Min doc len	Max doc len	Avr doc len	# documents
512	1024	752	727416
2048	3072	2544	215147
16384	17408	16880	32435
32768	33792	33268	16457
64512	65536	65001	8423
524288	525312	524430	1044

The document properties of `randomXX` files are locked to a minimum length of 64,512, a maximum length of 65,536, an average of about length 64,950. The number of documents are 323, 484, 646, 807, for `random20`, `random30`, `random40` and `random50`, respectively.

Both real and artificial document borders guarantee that all document sizes (including the special marker, if any) are multiples of 32 (they are 32-aligned). This is important for the lookup method. Now just one final note on the layout of a `doctable`

A `doctable` is a simple file describing the position of the documents in the concatenated text. These are simply stored as a contiguous list of integers. That means the number of documents is the size of the file divided by the size of an integer.

6.2.3 Queries

Query files are inputs to the the suffix array search and document listing tests. All our query files consists of 100 queries. A query has two parts, an integer representing its length and a pattern of the given length. 99% of the patterns are picked from the text. There is a 1% probability that any pattern will be a random collection of symbols, instead of being taken from the text.

Queries for texts with a word distribution (all `randomXX` texts) are aligned to word boundaries. Other queries are just random substrings of the text, possibly even crossing artificial documents borders.

Here is a list of the properties of the queries for `source`:

Number	Min len	Max len	Avr len	SA res num			Doc res num		
100				Avr	Min	Max	Avr	Min	Max
100	10	20	15.00	128405	1	6452843	128405	1	6452843
100	8	9	8.54	141100	1	6066031	141100	1	6066031
100	10	11	10.50	383597	1	10383440	383597	1	10383440
100	12	13	12.50	226781	1	8511536	226781	1	8511536
100	14	15	14.50	73431	1	7035802	73431	1	7035802
100	16	17	16.50	114765	1	5902058	114765	1	5902058
100	18	19	18.44	151513	1	5110741	151513	1	5110741
100	20	21	20.53	242766	1	5016080	242766	1	5016080
100	22	23	22.50	22911	1	1886215	22911	1	1886215
100	24	25	24.56	46430	1	4545660	46430	1	4545660

6.3 Results

Here we will present some result measures from runs of different tests.

6.3.1 Disk speed

These disk speed tests are included to show the obvious characteristics of the disk we used and that our memory classes described in section 5.1 seems to perform adequately. As such, the results will hopefully seem obvious.

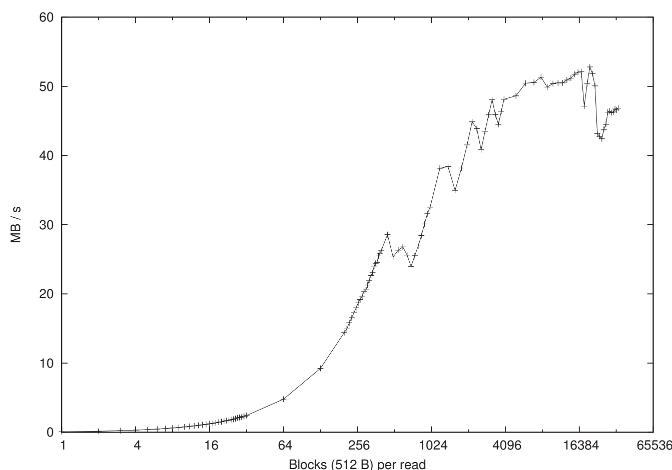

Figure 6.1: Disk performance

As we can see, the max is a little above 50 MB / s, which is close to optimal for a 7200 RPM SATA disk.

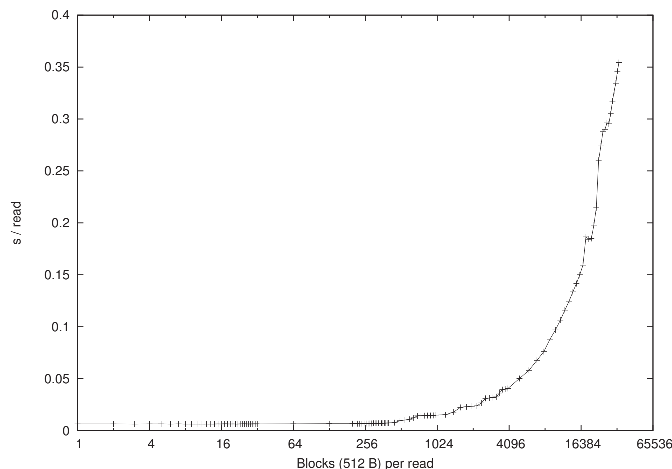

Figure 6.2: Block burst

At about 256 blocks per read, we see that each read is beginning to use more time. Thus, we will use 256 blocks as a minimum unit in further tests to read as much as possible into the cache without causing any overhead.

6.3.2 Suffix array construction

The DC3 algorithm was given 2 GB and of internal memory and 120 GB external memory, split equally on two disks. Here are some key results.

File	Input size	Time usage	Time per MB
genome	2928 MB	1395m 0s	28.6 s
source	522 MB	109m 52s	12.6 s
dummy	0 MB	17s	N/A

This is somewhat different from [DKMS05], where both source and genome use little more than 10 seconds per MB. This might be because they used four disks, which scaled better than our two disks. We have no better explanation for this currently.

On a key note, the startup time is rather long, although admittedly small compared to the anticipated run times.

6.3.3 Document information construction

Of the four document retrieval methods, three require preprocessing of the data, direct, lookup and embedded. The embedded method, however, assumes the information is embedded into the suffix array all the time. Thus, we are not interested in its artificial construction here. That leaves the direct and lookup method.

Varying alignment size

The lookup method depends on the alignment borders of the documents. Figure 6.3 shows the build time for the lookup method for varying document alignments size. The build time for the direct method is included for comparison. It is not a function of the alignment size.

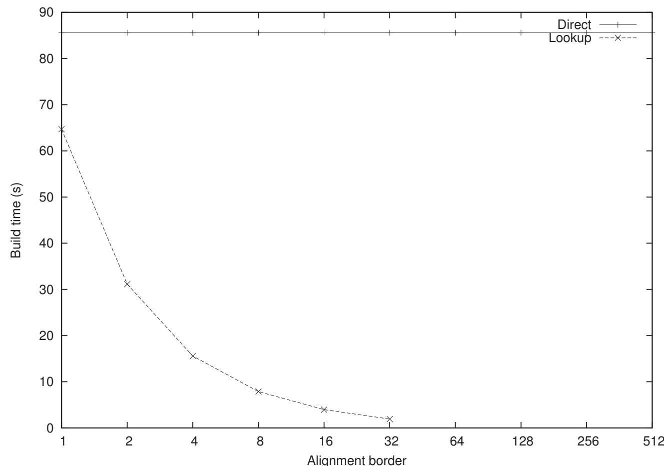

Figure 6.3: Document information construction, varying document alignment

As expected, the lookup method uses about twice the time for half the alignment sizes. This is because the output doubles when the alignment size halves, this is trivial from the algorithm in section 4.1.2.

The construction for direct use about 30% more time than a the lookup construction of the same size. This was also anticipated, since the direct algorithm performs a binary search for value (see section 4.1.1) while the lookup algorithm simply computes values.

Varying number of documents

The direct method depends on the number of documents, because it uses a binary search to find documents. Figure 6.4 shows its build time, with the build time for the lookup method included for comparison.

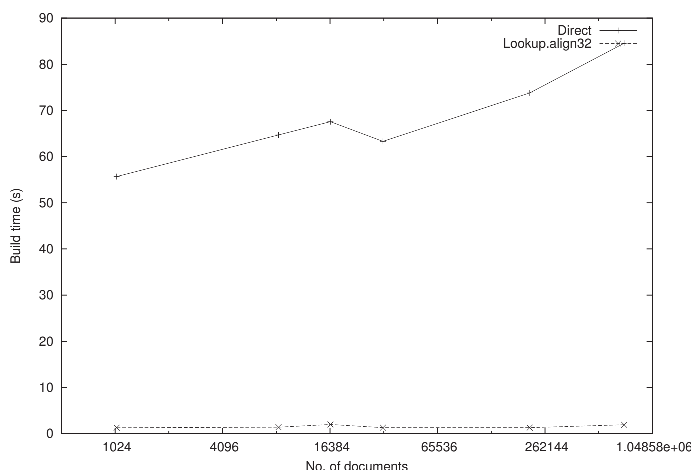

Figure 6.4: Document information construction, varying number of documents

As expected, the lookup construction time is approximately the same regardless of the number of documents.

The construction time for direct increases about as expected when it uses a binary search that must perform about twice the number of iterations for one million documents as for thousand documents. However, there is a strange anomaly on the line, a point where i breaks down and seems to continue in the same fashion. This puzzled us, and we were not able to find out why at this time.

6.3.4 Suffix array search

We operate we three different suffix arrays, the pure suffix array, the suffix array embedded with document pointers and the short suffix array embedded with document pointers. Here we will have a look at how the embedded document pointers affect the suffix array search time.

Varying text length

Since the suffix array search algorithm has running time $O(w \log n)$ we expect the running time to vary slightly as a function of the text length n.

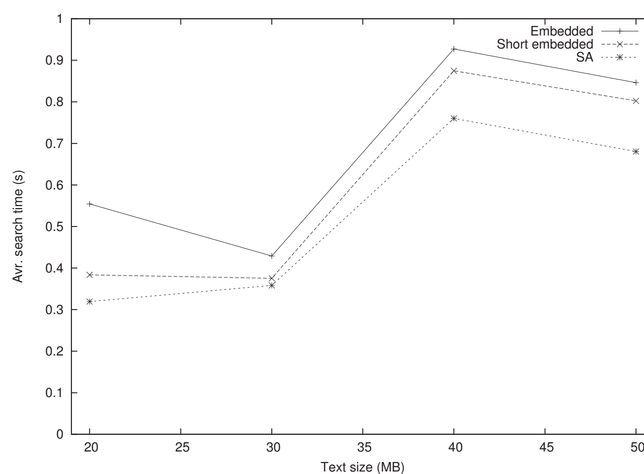

Figure 6.5: Suffix array search, varying text length

The bumps down from 20 to 30 and from 40 to 50 in figure 6.5 due to coincidences in the searching. As it turns out, the average number of results is just 1. Unfortunately, we are unable to run new tests at this time, but some general points are confirmed by this graph.

For one, the search time becomes approximately greater as the text size increases. Second, it takes longer to search with document information embedded. But looking at the internal ordering of the embedded and short embedded methods, we are in for a surprise. The short embedded method outperforms the embedded method even though it has to convert each suffix array entry from document offset to text index (see section 5.5). We believe this is because of cache effects due to the fact that fewer suffix array element fit in cache memory.

6.3.5 Duplicate removal

This is were the asymptotically optimal method for listing document hits could have been thoroughly tested. But as mentioned in section 3.3.5 it would degrade so bad on disk that testing it is not worth the effort.

We compare the three methods for duplication removal on the direct method, none, `sort` and marking. For control, we also include another method, lookup. This method only supports duplication using marking. Removal with the mark method is described in section 5.7.3.

Varying number of documents

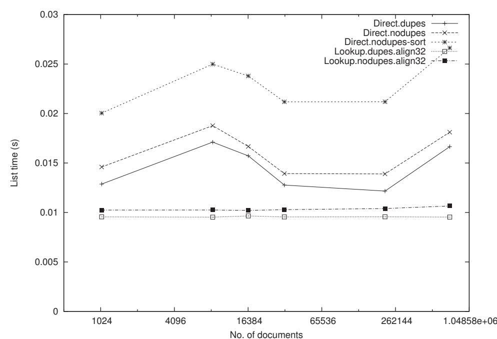

Figure 6.6: Duplication removal, time

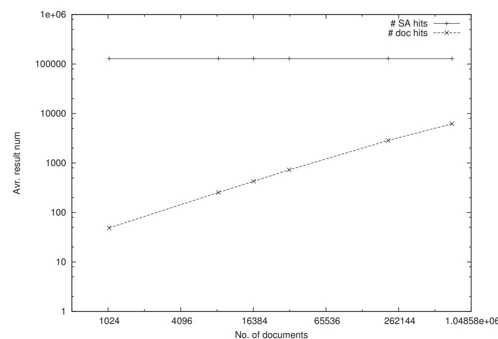

Figure 6.7: Duplication removal result

As expected the `sort` removal method is way slowest. A pleasant surprise is that removing duplicates with the mark method does not use much extra time at all.

The reason that the lookup method performs so much better than the direct method, is that is only has to read and lookup values from internal memory. Since the alignment is 32, the lookup table fits in internal memory. Further, the suffix array results that must be looked up, have been read into to memory by the suffix array search.

The direct method, on the other hand, has to access the direct table stored on disk. It is thus natural that it is slower, but it has a rather unnatural curve that we can not find any explanation for at this time. The top at the end is most likely due to the copying of very many documents pointers, the lookup method is also slightly affected by this.

Looking at the graph with result numbers, we see that the number of occurences is way higher than the number of documents where there is at least one occurence. Even so, the cost of removing all those duplicates is almost non-existing, judging by figure 6.7. This indicates that there is little need for an optimal document listing algorithm in our circumstances.

6.3.6 Document listing

This is the main section of our results, comparison of document listing methods. First, we start out by examining various alignment sizes for the documents input to the lookup method. Then we will compare all methods using a varying number of documents to see which method is more sensitive to that. Finally we will use varying query length to see which method is more sensitive to large numbers of suffix array results.

Varying alignment size

All else being equal, we vary how much the lookup method can assume the document alignment size is. The maximum alignment size we can use is 32, because that is the actual alignment size used for the document lengths.

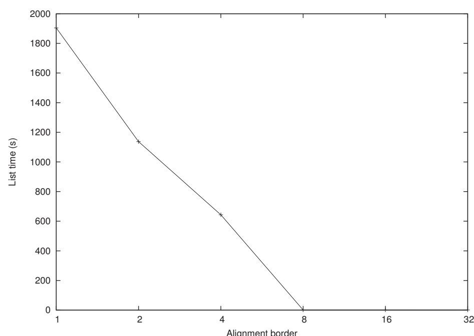

Figure 6.8: Varying alignment size, time

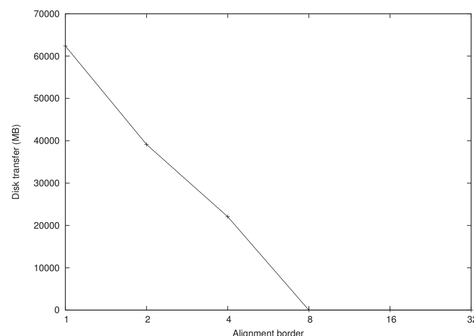

Figure 6.9: Varying alignment size, memory usage

Both graphs clearly show that the $lookup$ table does not fit into memory when the alignment size is 4 or less. Even though the algorithm sorts the suffix array results to avoid unnecessary disk accesses, it degrades too much when it has to fetch the $lookup$ table from disk.

The time values for alignment sizes of 8, 16 and 32 are 1.38, 1.23 and 1.06, respectively. That the time decreases when the alignment size increases is unexpected. We reckon it might be caused be fewer level 2 cache misses when the $lookup$ table is smaller.

Varying number of documents

Here we compare all the document retrieval methods, all of them with duplication removal. The duplication removal does not change the relative ordering since the same technique is applied to all methods. It does, however, provide more realistic timing, in the expected case that you do not want duplicates.

The input data is actually the same as the one we used to test different duplication removal methods. This is evident in the still unexplainable bump in the direct method timing.

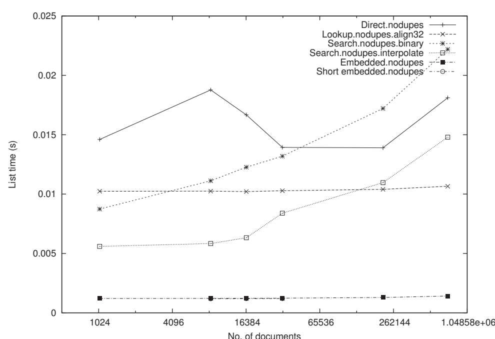

Figure 6.10: Method comparison, time

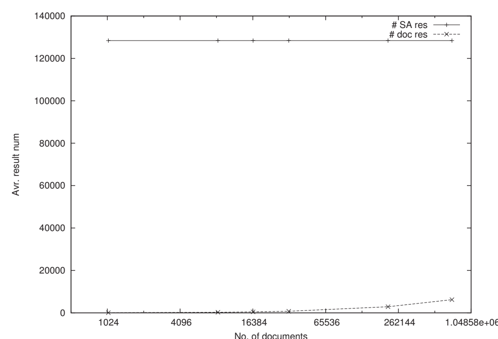

Figure 6.11: Method comparison results

The embedded methods perform equally well and outclass all other methods. This is no surprise, as all they have to do is read the document pointers directly from internal memory. A drawback of the short embedded method, was that is was only able to participate on three of the datapoints, the others had too many or too long documents for its short data types.

The interpolation search is significantly faster than the binary search. This should have been used during creation of document retrieval information (section 5.6).

The promising interpolation search breaks down at more than 100,000 documents and is beaten by the lookup approach. Why it took the lookup so long to beat it remains unsolved. We suspect that it might be a suboptimal

implementation.

The direct and binary search methods clearly perform worst. It seems like these should be avoided.

Varying query length

Here we have a look at how the methods respond when they must cope with many suffix array results. This test was performed with very small documents, so that there would not be so many duplicates to discard. The short embedded method could not participate because the number of documents was to great for it.

We did not care to test the binary search method, since it was clearly outperformed by the interpolation search method. Instead we tested how the direct method would perform if it did not have to remove duplicates.

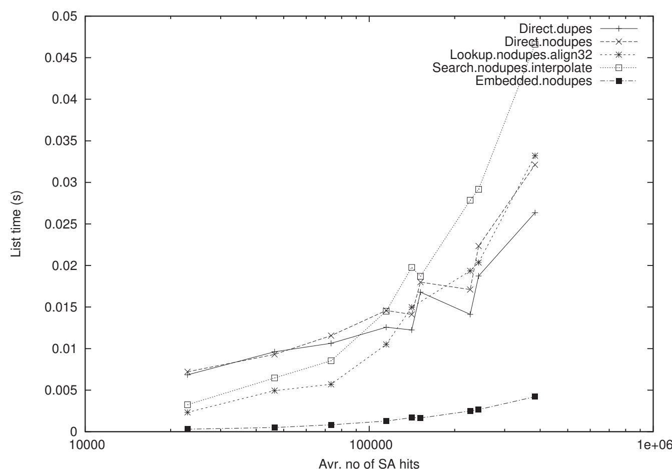

Figure 6.12: Method comparison, time

As before, the embedded method works wonderfully because the suffix array has already loaded its result into memory.

The direct method has to read much data from disk and suffers. The lookup method has to lookup each suffix array result and suffers. The interpolation search method is overloaded with searches and suffers. All these three methods seems to have about equal performance when faced with many suffix array results, except that the search method has to leave the game a little earlier.

Chapter 7

Summary and conclusions

We have reviewed various methods for solving the document listing problem. To the best of our knowledge, there are only three articles that give a thorough discussion of relevant algorithms. Manber and Myers [MM91] give two different search algorithms for suffix arrays, $O(w \log n)$ or $O(w + \log n)$. Baeza-Yates et al. [BYBZ96] give a search technique for better practical search for suffix arrays in external memory. Using these can give us the result for the suffix array, answering the occurence listing problem.

Muthukrishnan [Mut02] gives an optimal solution to the document listing problem. Given the suffix array range from the methods above, it can be used to list the documents in $O(docorr)$ extra time. However, this method does not work well on external memory, so other methods have to used.

We described four methods for solving the document listing problem, given a range in the suffix array. Our methods assume that a table of info about the documents can fit in internal memory, but not that the same applies for the suffix array.

Our experiments show that if the results from the suffix array will be read to internal memory, embedding the document pointers into the suffix array is the best choice.

If the suffix array results will not be read, the direct method might be preferred. It holds the same information as the embedded document pointers, only that they are not embedded. All other methods depend on reading the suffix array results.

Our results are not very solid. Some graphs had some traits whose sources we were unable to locate. Also, the interaction between the suffix array search and the document listing is not very thouroughly documented. The assumption that the suffix array results are read into memory may not be a valid one.

Furthermore we attempted to look at the suffix array searching itself. Due to implementation difficulties and limited time this attempt failed, and a short description of possibilities of using LCP information and interleaved text is all that we have contributed here.

Further work should include finding the source of the pecularities in our results, doing a more thourough analysis of the interaction between suffix array searching and document listing, and finding a new and improved method for searching suffix arrays that are in external memory.

Bibliography

[AGKR04] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 2004. A

[BFC00] Michael A. Bender and Martin Farach-Colton. The lca problem revisited. In *LATIN '00: Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, London, UK, 2000. Springer-Verlag. A, 1

[BYBZ96] Ricardo Baeza-Yates, Eduardo F. Barbosa, and Nivio Ziviani. Hierarchies of indices for text searching. *Inf. Syst.*, 21(6):497–514, 1996. 1, 3.2, 7

[DKMS05] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. In *Workshop on Algorithm Engineering & Experiments*, Vancouver, 2005. 1, 2.4, 3.2, 5.4, 6.1, 6.2.1, 6.3.2

[FG04] Hans Christian Falkenberg and Nils Grimsmo. Introduction to string searching and comparison of suffix structures and inverted files. Technical report, Norwegian University of Science and Technology, 2004. 2.5, 3.1.1, 3.1.2, 5.5

[GBYS92] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: Pat trees and pat arrays. pages 66–82, 1992. 3.1

[HBYFL92] Donna Harman, R. Baeza-Yates, Edward Fox, and W. Lee. Inverted files. pages 28–43, 1992. 2.2

[KA03] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*, 2003. 1, 2.4

[KLA+01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. pages 181–192, 2001. 3.1.2

[KS03] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 13th International Conference on Automata, Languages and Programming*. Springer, 2003. 1, 2.4

[KSPP03] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching: 14th Annual Symposium*, pages 186–199. Springer-Verlag Heidelberg, 2003. 1, 2.4

[Kur99] Stefan Kurtz. Reducing the space requirement of suffix trees. *Softw. Pract. Exper.*, 29(13):1149–1171, 1999. 2.4

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. 2.5

[Meh04] Jens Mehnert. External memory suffix array construction. Master's thesis, Department of Computer Science, Saarland University, 2004. 3.2

[MM91] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. 1991. 2.4, 3.1, 3.1.1, 3.1.2, 7

[Mut02] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 657–666, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. 1, 3.3, 3.3.5, 7

[San02] Peter Sanders. Memory hierarchies - models and lower bounds. In *Algorithms for Memory Hierarchies*, pages 1–13, 2002. 2.6

[SS05] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. In *Proceedings of ALENEX*, 2005. 2.4

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(5):249–260, 1995. 2.5

[Wei73] P. Weiner. Linear pattern matching algorithms. In *IEEE 14th Annual Symposing on Switching and Automata Theory*, pages 1–11, 1973. 2.5, 3.1

Appendix A

Range Minimum Query and Nearest Common Ancestor

Range Minimum Queries (RMQ), also known as Discrete Range Searching, must be solved in O(1) time for asymptotical optimality of many algorithms, including the document retrieval problem. It can be solved in O(1) time after O(n) preprocessing time with O(n) extra storage. There are two reasons why the solution to this problem is presented here. For one, we want this thesis to be self contained. Second, we have not seen any derivation of the actual space usage for a solution. We need to know the space usage more precisely than asymptotic notation can offer to determine when it is possible to use the optimal solution to this problem. Presenting the solution is necessary to prove the space usage, which we will do at the end of this chapter.

The RMQ problem is closely related to the Nearest Common Ancestor (NCA) problem, also known as the Least Common Ancestor and Lowest Common Ancestor problem. The solution presented for the RMQ problem was formulated in an easily readable, understandable and implementable form by Bender and Farach-Colton in 2000 ([BFC00]). It depends on solving the NCA problem, which in turn is solved as a special version of the RMQ problem. Alstrup et al. presented a survey of different solutions to the NCA problem in 2004 ([AGKR04]), but we still find the presentation of Bender and Farach-Colton the better one, so we will present their solution here. It might be that other solutions have better space usage, but an analysis is out of the scope of this thesis and Alstrup et al. gives no empirical results, only asymptotic ones. We now proceed to define the RMQ and NCA problems and show their solutions (reductions to each other and a independent RMQ solution) along with the real space cost of each solution.

To ease later notation, we define some concepts:

Definition A.1. For a binary tree T with nodes V and $u, v \in V$ then

$root(T)$	is	the root node of the tree,
$parent(u)$	is	the parent node of w ($parent(root(T))$ is undefined),
$left(u)$	is	the root node of the left subtree of u or θ,
$right(u)$	is	the root node of the right subtree of u or θ,
a leaf	is	any node u in V such that $left(u) = \theta$ and $right(u) = \theta$,
an ancestor of u	is	any node on the path from u to $root(T)$ — including u,
$anc(u)$	=	$\{v \mid v$ is an ancestor of $u\}$,
$height(u)$	=	$\lvert anc(u)\rvert - 1$, that is, u's distance from $root(T)$,
$euler(T)$	=	$[root(T), left(root(T)), ..., parent($leftmost leaf$)$, leftmost leaf, leftmost leaf, $parent($leftmost leaf$)$, $right(parent($leftmost leaf$)), ..., right(root(T))]$, that is, the list of nodes visited in a depth first search of T and
$eulerindex(u)$	=	$i \mid min_{i \in [0, \lvert euler(T)\rvert - 1]} euler(T)[i], euler(T)[i] = u$.

Definition A.2. The Range Minimum Query (RMQ) problem is to find the index in a table $A[0...n-1]$ for the minimum value in a subarray $A[i...j], 0 \leq i \leq j < n$. Formally, $rmq_A(i, j) = k \mid min_{k \in [i,j]} A[k]$.

Definition A.3. The Nearest Common Ancestor (NCA) problem is to find the node in a tree T that is an ancestor of both a node u and a node v and is furthest from the root of all such nodes. Formally, $nca_T(u, v) = w|min_{w \in anc(u) \cap anc(v)} height(w)$.

The proof of lemma A.2 relies on the Cartesian Tree. It is defined as follows.

Definition A.4. A Cartesian Tree (CT) is a recursively defined binary tree holding information about an array $A[0...n-1]$ such that the minimum value in a subarray $A[i...j]$ can be found. The root of a CT for array A is labeled with the index, x, of the minimum element of A. The left subtree of the root is the CT of the subarray left of x and the right subtree of the root is the CT of the subarray right of x. We say that the root splits A at x. If a subarray is empty, so is the coresponding subtree.

Lemma A.1. *The Cartesian Tree of an array $A[0...n-1]$ can be created in $O(n)$ time and space.*

Proof. A CT, C, can be created as follows. Assume C_i is the CT of $A[0...i]$. We want to create C_{n-1} for array $A[0...n-1]$. C_0 consists of one node labeled 0. C_{i+1} can be created by adding $A[i+1]$ to the rightmost path on C_i as follows. Start by traversing the rightmost leaf node of C_i, ascending towards the root. When a node is found whose value in $A[0...i]$ is smaller than $A[i+1]$, move that node's right subtree to be the left subtree of a new node labeled i and replace it with the new subtree rooted by the i node. This way, the new node becomes the new rightmost leaf. If there is no node with a smaller value, the new node becomes the new root, with C_i as its left subtree. When the tree has been fully contructed, each node has been passed on the traversion of the rightmost path at most once. Thus, the construction time is linear on the size of the C. Since C has one node for each in A, its size is clearly $O(n)$ from which we can conclude that the construction time is also $O(n)$. ☐

Lemma A.2. *If NCA problems can be solved in $O(g(n))$ time with $O(f(n))$ preprocessing time, RMQ problems can be solved in $O(1 + g(n))$ time with $O(n + f(n))$ preprocessing time.*

Proof. We want to answer RMQ problems for a table $A[0...n-1]$. This can be reduced to the NCA problem in $O(1)$ time. Create the Cartesian Tree, C, for table A along with a table T_{A2C} mapping indexes of A into nodes of C and solve the NCA problem for C. To see why that is sufficient, consider the following.

If we want to find the minimum value of $A[i...j]$ we start by finding the nodes labeled i and j in C. This can be done in $O(1)$ time using T_{A2C}. Their nearest common ancestor is labeled with the index of the minimum value in $A[i...j]$.[1] Formally, $rmq_A(i, j) = label(nca_C(T_{A2C}[i], T_{A2C}[j]))$. Thus, we can solve an RMQ problem in $O(1)$ time given a solution to the NCA problem of C, which is linear on size of A. In other words, the RMQ solution can be found in $O(1 + g(n))$ time.

The preprocessing time for T_{A2C} is $O(n)$ (trivially, it can be constructed from one traversion of C). The preprocessing time for C is also $O(n)$, according to lemma A.1. Clearly, the preprocessing time to solve RMQ problems is then $O(n + f(n))$. This concludes our proof. ☐

The proof of lemma A.4 relies on some observations and a lemma. The lemma assumes an array has been splitted into groups of size $\frac{\log n}{2}$. This value is chosen to reduce the workload of the undelying RMQ solution, so that it becomes linear on the size of the input to the NCA problem.

Observation A.2.1. *The NCA of nodes u and v in a tree T is the shallowest node encountered between the visits to u and v during a depth first search traversal of T.*
Formally, $nca_T(u, v) = w|min_{w \in euler(T)[eulerindex(u),...,eulerindex(w)]} height(w)$.

Observation A.2.2. *If two arrays, $X[0...k-1]$ and $Y[0...k-1]$ differ only by some fixed value in each position, they have the same RMQ solutions. Formally, if $X[i] = Y[i] + c, 0 \leq i < k, c \in \mathbb{R}$ then $rmq_X(i, j) = rmq_Y(i, j), 0 \leq i \leq j < k$.*

Lemma A.3. *Assume we have blocks (subarrays) $B[0...\frac{\log n}{2} - 1]$ whose adjacent values differ by ± 1. Normalize the blocks by substracting each blocks first value from all its values. There are $\frac{\sqrt{n}}{2}$ distinct such blocks.*

[1]The proof for this can be found in [BFC00]. It is trivial and irrelevant for this discussion.

Proof. $B[0] = 0$ for all normalized blocks. The rest of B's values can be specified by a ± 1 vector of length $\frac{\log n}{2} - 1$. There are $2^{\frac{\log n}{2} - 1} = \frac{\sqrt{n}}{2}$ such vectors. □

Lemma A.4. *If RMQ problems can be solved in $O(g(n))$ time with $O(f(n))$ preprocessing time, NCA problems can be solved in $O(O(1) + g(2n-1))$ time with $O(n + f(2n-1))$ preprocessing time.*

Proof. We want to answer NCA problems for a tree T. This can be reduced to RMQ problem in $O(1)$ time. Create a table for T's Euler Tour, $E[0...2n-2] = euler(T)$, a table for the corresponding height values, $L[0...2n-2], L[x] = height(E[x])$ and a lookup table for the first Euler Tour occurence for node indexes, $R[0...n-1], R[x] = eulerindex(x)$.

According to observation A.2.1, $nca_T(i,j) = E[rmq_L(R[i], R[j])]$.

Split L into blocks of size $b = \frac{\log n}{2}$, let $n' = \frac{n}{b} = \frac{2n}{\log n}$ and create a table holding the minimum value of each such block, $L'[0...n'-1], L'[x] = min_{y \in [xb...(x+1)b-1]} L[y]$, and a table holding the corresponding nodes, $E'[0...n'-1], E'[x] = E[y], y | min_{y \in [xb...(x+1)b-1]} L[y]$.

If $R[i]$ is on the left border of a block and $R[j]$ is on the right border of a block, $rmq_L(R[i], R[j]) = rmq_{L'}(\frac{R[i]}{b}, \frac{R[j]}{b})$. To answer other RMQ problems, we need to preprocess the answers to all possible blocks. Due to observation A.2.2, we only need to do this for all possible normalized blocks, namely $q = \frac{\sqrt{n}}{2}$ blocks, according to lemma A.3. Each block has $O(b^2)$ possible RMQ problems. To be exact, each block has $p = \sum_{i=0}^{b-2} \sum_{j=i+1}^{b-1} 1 = \frac{b^2 - b}{2} = \frac{\log^2 n}{8} - \frac{\log n}{4}$ possible RMQ problems.

□

Lemma A.5. *RMQ problems can be solved in $O(1)$ time with $O(n \log n)$ preprocessing time directly with dynamic programming.*

Proof. We want to answer RMQ problems for a table $A[0...n-1]$. To do this in O(1) time we only need to store the answers for all RMQ problems with a range of length $2^k, 0 \leq k < \lfloor \log n \rfloor$ in a table. For each range length 2^k, there are $n - 2^k + 1$ RMQ problems. For simplicity of the following discussion we assume that each range length has n associated RMQ problems; that is, we shorten the length of the problems hitting the right boundary of A. The dynamic programming solution will be simpler, but in an implementation we could leave out the computation of these boundary values (but that does not save any significant amount of time).

When we can lookup the answer to all RMQ problems of length 2^k it is easy to find the answer of a RMQ problem of any length. The minimum value in a range is the minimum of the answers to two overlapping RMQ problems of length 2^k, one starting at the left and the other ending at the right border of the range.

Formally, to answer $rmq_A(i,j), 0 \leq i \leq j < n$ for a table $A[0...n-1]$ create a table
$M[i][k] = rmq(i, 2^k), 0 \leq i < n, 0 \leq k < \lfloor \log n \rfloor$
using dynamic programming. The values of $M[i][1]$ are trivial ($M[i][1] = i$). For $M[i][j], j \geq 2$ we have,

$$M[i][j] = \begin{cases} M[i][j-1] & \text{if } A[M[i][j-1]] \leq A[M[i+2^{j-1}][j-1]], \\ M[i+2^{j-1}][j-1] & \text{else.} \end{cases}$$

Select $k = \lfloor \log(j - i + 1) \rfloor$. $rmq_A(i,j) = min(M[i][k], M[j - 2^k + 1][k])$.

Clearly, computing this minimum value can be done in $O(1)$ time. Finding k is the same as finding the 0-indexed position of the most significant bit of $(j - 1 + 1)$. This, unfortunately, takes $O(logn)$ time in the general case, using the naive solution of incrementing k from -1 while bitshifting the length until it is 0. However, since the

operation can be implemented in constant time given our practical unit cost assumption, finding k is assumed to take constant time.[2] Thus, the RMQ problem is solved in $O(1)$ time after preprocessing.

The table M is of size $n(\lfloor \log n \rfloor + 1)$ and can clearly be filled in $O(n \log n)$ time using the dynamic programming formula described above. Thus, the preprocessing to solve the RMQ problem is done in $O(n \log n)$ time and this proof is concluded. □

[2] The x86 processor family actually has an instruction, `bsr` (bit shift right), that returns the index of the most significant bit. Note, however, that this function has been criticized for being slow, `http://www.codecomments.com/A86_Assembler/message459724.html`. Another solution is to create a table of size 256 with the index of the most significant bits of the numbers in the range $[0, 255]$. We can then find k with at most 4 lookups on a 32-bit machine (where the assumption is that $n < 2^{32}$) and similarly with at most 8 lookups on a 64-bit machine. If we should choose to take the view that finding k takes $O(logn)$ time, the given solution for solving the RMQ by reduction to the NCA problem would take $O(\log n)$ time with $O(n \log n)$ preprocessing time.