Preface

This is a master thesis for the Master of Technology program at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The assignment was given by Fast Search and Transfer ASA. The report and underlying work was done in five months, spring 2005.

The author would like to thank fellow student Hans Christian Falkenberg for useful discussions, and collaboration on previous work, and mentor Øystein Torbjørnsen for constructive feedback.

Trondheim, 22nd June 2005

____

Nils Grimsmo

Abstract

This report explores the problem of substring search in a dynamic document set. The operations supported are document inclusion, document removal and queries. This is a well explored field for word indexes, but not for substring indexes. The contributions of this report is the exploration of a multi-document dynamic suffix tree (MDST), which is compared with using a hierarchy of static indexes using suffix arrays. Only memory resident data structures are explored. The concept of a "generalised suffix tree", indexing a static set of strings, is used in bioinformatics. The implemented data structure adds online document inclusion, update and removal, linear on the single document size.

Various models for the hierarchy of static indexes is explored, some which of give faster update, and some faster search. For the static suffix arrays, the BPR [SS05] construction algorithm is used, which is the fastest known. This algorithm is about 3-4 times faster than the implemented suffix tree construction. Two tricks for speeding up search and hit reporting in the suffix array are also explored: Using a start index for the binary search, and a direct map of global addresses to document IDs and local addresses.

The tests show that the MDST is much faster than the hierarchic indexes when the index freshness requirement is absolute, and the documents are small. The tree uses about three times as much memory as the suffix arrays. When there is a large number of hits, the suffix arrays are slightly faster on reporting hits, as there they have better memory locality. If you have enough primary memory, the MDST seems to be the best choice in general.

Contents

Preface 3

Abstract 4

1 Introduction 10

2 Background 11

2.1 Notation . . . 11
2.2 Suffix trees . . . 12
2.2.1 Tries . . . 12
2.2.2 Suffix tree definition . . . 13
2.2.3 Representing suffix trees . . . 14
2.2.4 Searching in suffix trees . . . 16
2.2.5 Building suffix trees . . . 17
2.3 Suffix arrays . . . 21
2.3.1 Searching in suffix arrays . . . 22
2.3.2 Building suffix arrays . . . 23
2.4 Hierarchic indexes . . . 24
2.4.1 Method 1 . . . 24
2.4.2 Method 2 . . . 26

3 Multi-document Dynamic Suffix Trees 28

3.1 Definition . . . 28
3.2 Document insertion . . . 29
3.3 Document deletion . . . 29
3.4 MDST implementation . . . 31
3.5 Node model . . . 31
3.5.1 `struct` nodes . . . 31
3.5.2 Compact array nodes . . . 31
3.5.3 Large and small internal nodes . . . 32
3.5.3.1 Re-using space for internal nodes . . . 34
3.5.3.2 Deleting nodes from chains . . . 35
3.5.4 Leaf node addresses . . . 35
3.6 Parenthood . . . 36
3.6.1 Sibling lists . . . 36
3.6.1.1 Sibling list space cost summary . . . 37

3.6.2 Hash map . . . 37
3.6.2.1 Lost head positions . . . 38
3.6.2.2 Hash map keys . . . 38
3.6.2.3 Implicit small nodes . . . 38
3.6.3 Linked hash map . . . 39
3.6.3.1 Insertion and removal in linked hash map . . . 39
3.6.3.2 Split end markers . . . 40
3.6.3.3 Linked hash map space cost summary . . . 40
3.6.3.4 Hash map implementation . . . 40
3.6.4 Child arrays . . . 42
3.6.4.1 Unsorted child arrays . . . 43
3.6.4.2 Sorted child arrays . . . 43
3.6.4.3 Deleting from child arrays . . . 44
3.6.4.4 Doubling factor . . . 44
3.6.5 Parent–child implementation summary . . . 44
3.7 Document ID update model . . . 44
3.7.1 Eager bottom-up document ID clean-up . . . 46
3.7.2 Lazy document ID clean-up . . . 47
3.7.3 Updating document IDs in small–large chains . . . 48

4 Hierarchic index implementation **49**
4.1 Method 1 . . . 49
4.2 Method 2 . . . 50
4.3 Comparing method 1 and method 2 . . . 50
4.4 Suffix arrays for multiple documents . . . 51
4.5 Document ID map . . . 52
4.6 Start index for binary search . . . 53

5 Results **54**
5.1 Test data . . . 54
5.2 Test system . . . 55
5.3 Comparison of node models for MDST . . . 55
5.3.1 General test on zipf data . . . 56
5.3.2 Varying text randomness . . . 58
5.3.3 Varying alphabet size . . . 59
5.3.4 Varying number of hits . . . 60
5.3.5 Real world data . . . 61
5.3.6 MDST conclusion . . . 62
5.4 Tuning of hierarchic models using BPR . . . 62
5.4.1 Testing method 1 and method 2 . . . 63
5.4.2 Testing docID map . . . 64
5.4.3 Testing binary search start index . . . 65
5.4.4 Hierarchic index conclusion . . . 66
5.5 Comparison of static and dynamic indexes . . . 67
5.5.1 Varying document size . . . 67
5.5.2 Varying freshness requirement . . . 69

5.5.3 Varying number of hits 69
5.5.4 Varying query length 70
5.5.5 Increasing total data size 71
5.6 Critique of the tests 72

6 Conclusion **75**
6.1 Use cases 75
6.2 Further work 76

Bibliography **78**

List of Figures

2.1	Suffix trie for the string ”abbabaabab”	13
2.2	Suffix tree for the string “abbabaabab”	14
2.3	Python program for searching in a suffix tree.	16
2.4	Python program for finding all hits.	17
2.5	Suffix tree for “abbabaabab” after step 6 and 7	18
2.6	Conceptual suffix tree building	19
2.7	Python program for building suffix tree	19
2.8	`rescan` function	20
2.9	`scan` function	21
2.10	Suffix array for the string “abbabaabab”.	22
2.11	Searching for the “leftmost” match of a pattern.	23
2.12	Method 1 for hierarchic indexes	25
2.13	Method 2 for hierarchic indexes	26
3.1	Traversing a suffix tree	29
3.2	Deleting a string from a suffix tree	30
3.3	Suffix tree for the string ”ababa”	33
3.4	Large and small internal nodes in the suffix tree for “ababa”	34
3.5	Example field sizes	37
3.6	Node fields required for storing node using sibling lists	37
3.7	Node references with implicit small nodes	39
3.8	Example field sizes	40
3.9	Fields with linked hash maps	41
3.10	Child arrays.	42
3.11	Summary of parent–child models	45
3.12	MDST for “xag”, “xabcd”, “xabe” and “xabcf”	46
3.13	Inheriting $docid$ and $hpos$ from children.	47
4.1	Pseudocode for method 1	50
4.2	Pseudocode for method 2	51
4.3	Work per document	52
4.4	Maximum number of indexes	52
5.1	Test parameters	55
5.2	Using zipf data. Values sampled during run.	57
5.3	Varying text randomness.	58
5.4	Statistics for varying text randomness.	59

5.5 Varying alphabet size, random data. 60
5.6 Average query time, varying number of hits. 60
5.7 Document inclusion time in seconds 61
5.8 Average query time in μ-seconds 61
5.9 Memory usage in megabytes 62
5.10 Average document inclusion time. 63
5.11 Average query time. 64
5.12 Maximum memory usage for sampled period, over total size of data indexed. 64
5.13 Increasing number of documents, constant data size. 65
5.14 Average query time. Varying data size. 66
5.15 Inclusion time. Varying data size. 66
5.16 Total inclusion with varying document size, constant data size. 67
5.17 Minimum and maximum inclusion time, 40MB total data. 68
5.18 Memory usage 68
5.19 Average inclusion time for varying freshness requirements 69
5.20 Reporting a variable number of hits for a set of documents. 70
5.21 Reporting one hit, varying query length. 71
5.22 Inclusion time. Increasing data size. 72
5.23 Query time. Increasing data size. 73
5.24 Average for sampled period. Increasing data size. 73
5.25 Memory usage. 74

Chapter 1

Introduction

Searching for a pattern in a string is a well known problem, which has very different properties in different applications. In desktop applications, such as word processors, you search for simple patterns in a small dynamic text. In bioinformatics, you search for complex patterns in a larger static text. In web search engines, you search for very simple patterns in huge amounts of data. There are many applications with complexities between these. For a more thorough overview of the text search problem, the reader is referred to any text book on the subject, or [FG04], which is the report from a preliminary project for this master degree thesis.

The main problem discussed in this report is providing fast substring search on a dynamic set of documents. For term search, this is a very mature field. For example, web search engines give term search on a dynamic document set. In the bioinformatics literature, a suffix tree for a static set of strings is sometimes called a generalised suffix tree. The author has not been able to find any literature on such trees for dynamic sets of strings.

An example of an application were there is a lot of data, which must be searchable instantaneously, is the news bulletins of news agencies, or the stock prices at an exchange market. The former usually requires only word searches, while the latter also requires more complex range searches for numbers and so on. This report investigates full substring search for a rapidly changing document set. Whether this ever will be useful in any major application is an open question. The most probable might be to offer substring search as an addition to other types of search, where this is useful and affordable.

Chapter 2 of this report gives an overview of the existing methods which were used as a foundation for this work: The suffix tree, the suffix array, and hierarchic indexes in general. Suffix trees and suffix arrays are well known substring indexes. A hierarchy of indexes is necessary to use the very static suffix array on a dynamic problem. Chapters 3 and 4 describe the details and implementation for solving the dynamic document set problem. Test results are given in chapter 5.

Chapter 2

Background

This chapter describes the basics of suffix trees (section 2.2) and suffix arrays (section 2.3), which are substring indexes, allowing fast substring search. The trees are dynamic structures allowing updates, while the arrays are static of nature. Therefore we also investigate how to turn static searching problem into dynamic searching problems by using hierarchic indexes (section 2.4). The first parts of this chapter are only concerned with single document problems, while the last part discusses dynamic sets of documents.

2.1 Notation

The following notation is used in the rest of the report. Less frequently used notation is introduced as needed.

Σ The alphabet.

σ The size of the alphabet.

T The text we search in.

T_i The ith suffix of T.

$T_{i...j}$ The substring of T from *inclusive* position i to *exclusive* position j.

n The length of T in symbols.

P A query string.

m The length of P in symbols.

occ The number of occurrences of P in T.

2.2 Suffix trees

Suffix trees are very powerful text indexes with optimal asymptotic space and time requirements for both construction and search: $\Theta(n)$ and $O(m)$. The first description of linear time suffix trees was given by Weiner in [Wei73], but in this text we use the description by McCreight in [McC76] as a starting point, as it gives a simpler and more readable overview. Although suffix trees are asymptotically optimal, they have not been widely used outside the realm of bioinformatics, because the construction algorithm is complex, and because they use a lot of space compared to word indexes. They also do not work well on disk in their basic form. The implementation from [Kur99] uses $20n$ bytes in the worst case, and around $10n$ bytes in practice, while word indexes typically use less than n bytes per input character.

In the following descriptions, Greek letters will be used to denote strings, uppercase letters will be used to denote unknown characters, and lowercase letters will be used to denote constant characters. An over-lined string, such as $\overline{\alpha}$, refers to the node representing the string α in a tree.

2.2.1 Tries

Before we define suffix trees, we define the trie, which is a flexible data structures that can be used for indexing words.

Definition 2.1: A trie of a set of words is a rooted tree such that each edge represents a symbol, the symbols on the outgoing edges of a given node are unique, and the paths from the root node to the leaves spell the words in the set.

Note that you can check whether a trie contains a word P of length m in $O(m)$ time. Since the paths of unique words are unique, there is one leaf node for each of the words. In the worst case, no words share common edges (start with a unique symbol), and if the total length of the words is N, there are N edges and $N + 1$ nodes.

Definition 2.2: A suffix trie is a trie representing all suffixes T_i of a string T of length n padded with a unique end-of-string symbol $.

A suffix trie for the string “abbabaabab” is shown in figure 2.1.

Since $ is a unique symbol, all suffixes are unique words, and will correspond to a unique path in the tree and a unique leaf node. Counting the last suffix of the padded string, “$”, there will be $n + 1$ leaf nodes. The suffixes are of length $n + 1$ to 1. In figure 2.1, the number in a leaf node is the starting position of the string spelled by the path to the node. The left to right order of the leaves gives the sorted order of all the suffixes of the string.

Remark 2.3*:* Given a random string, the expected maximal longest common prefix of a suffix with any other suffix increases on n, and decreases on σ. If you view the alphabet size as a constant, the expected number of edges not shared with another suffix is $\Theta(n) - \Theta(\log n) =$

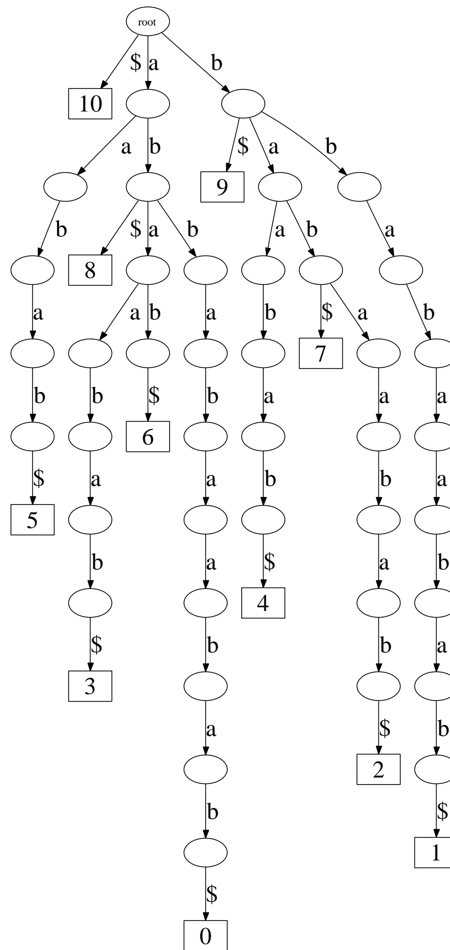


Figure 2.1: Suffix trie for the string "abbabaabab"

$\Theta(n)$ for all suffixes, as they have an average length of $\frac{1}{n}\sum_{i=1}^{n+1} i \in \Theta(n)$. Therefore the expected number of edges in a suffix trie for a random string is in $\Theta(n^2)$.

2.2.2 Suffix tree definition

A suffix tree is an improvement over the suffix trie, which retains the $O(m)$ lookup time, but consumes $\Theta(n)$ space. It is given by the following definition:

Definition 2.4 ([McC76]): A suffix tree is a suffix trie where all non-root nodes with a single child have been removed, and the related edges have been merged. The resulting tree has a root node, branching nodes and leaf nodes.

A suffix tree of a string is the same as a Patricia trie [Mor68] of all suffixes.

Remark 2.5: Note that the first symbol on each outgoing edge of a given node is still unique. If the unpadded string T is empty, the root will have only one child, on the symbol "$". If T is non-empty, the root will have at least two children, as "$" does not occur in T.

You see the suffix tree for the string “abbabaabab” in figure 2.2. Notice how all suffixes are represented by a unique path from the root to a leaf. The dashed lines are suffix links, explained in section 2.2.5.

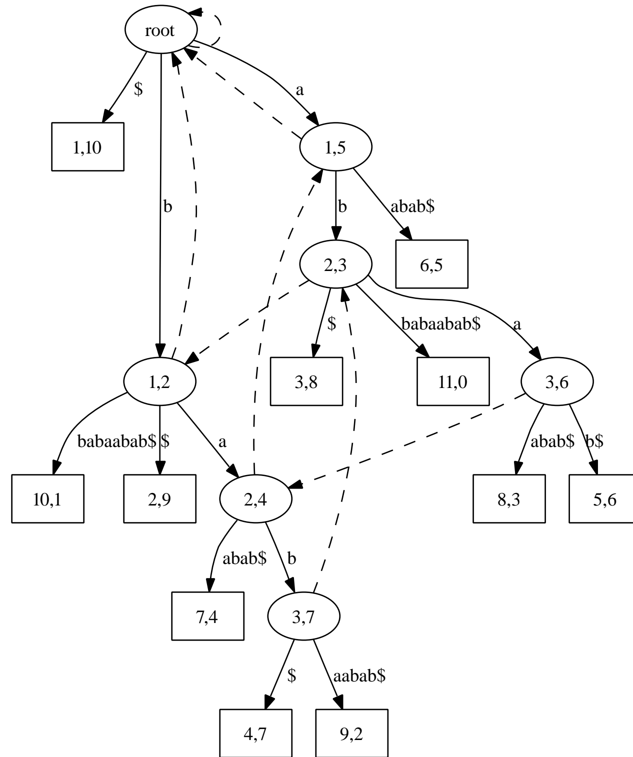


Figure 2.2: Suffix tree for the string “abbabaabab”

Theorem 2.6 ([McC76]): A suffix tree of a string of length n has $\Theta(n)$ nodes.

Proof. Trivial from the fact that all internal nodes are branching and that there are $n + 1$ leaf nodes. □

2.2.3 Representing suffix trees

The actual representation of suffix trees used here is adapted from [Kur99], which the most acclaimed suffix tree implementation, and is used in the MUMmer software [MUM]. This representation is also used as a starting point for the multi document tree described in chapter 3.

Even though a suffix tree has $\Theta(n)$ nodes, it might not be clear how to represent it in $\Theta(n)$ space, as the length of the strings represented by the edges is not constant. We solve this by representing an edge by the starting and ending point in the actual string T, stored in the child node. This requires keeping a copy of T. As shown in [Kur99], instead of saving the starting and ending-point of the edge, you can save the head position ($hpos$) and $depth$ of a node, defined below.

- T_i — The ith suffix of T.
- $head_i$ — The longest prefix of T_i which is equal to a prefix of T_j for a $j < i$.
- $tail_i$ — T_i minus the prefix $head_i$.
- $node.depth$ — The length of the string represented by the edges from $root$ to $node$.
- $node.hpos$ — The starting position of the string represented by the edges from $root$ to $node$. (Explained below.)

In figure 2.2, the pair of numbers inside the nodes represent the $depth$ and $hpos$ of the node respectively.

Given that $head_i$ is the head of the suffix T_i, there will be an internal node $node$, with the edges from $root$ to $node$ spelling $head_i$. This must be, since $head_i$ was the longest common prefix with any T_j with $j < i$, which means the symbol after $head_i$ must be different in T_i and T_j for any T_j with prefix $head_i$ and $j < i$. $node.hpos$ can be set to i, and $node.depth$ is $|head_i|$. In an implementation, the $hpos$ of a node is naturally set to the lowest i such that $node$ represents $head_i$, that is the second lowest index i such that the string spelled by the path to the node is a prefix of T_i.

Remark 2.7 *([Kur99]):* The starting point of an edge from $parent$ to $child$ is $child.hpos + parent.depth$, and the ending point $child.hpos + child.depth$.

Example 2.1*:* Consider the edge between the nodes labelled $(2, 3)$ and $(3, 6)$ in figure 2.2. The starting position of the string represented by the edge between them is $(3, 6).hpos + (2, 3).depth = 6 + 2 = 8$, and the ending position $(3, 6).hpos + (3, 6).depth = 6 + 3 = 9$. The represented string is then $T_{8...9}$ = “c”.

Remark 2.8 *([Kur99]):* The following property is always true: If the node x has a descendant y, then $T_{x.hpos\ ...\ x.hpos+x.depth} = T_{y.hpos\ ...\ y.hpos+x.depth}$. This just states that the path from the root to an internal node is a prefix of all strings represented by paths from the root to the nodes in the subtree of this node. Likewise, if for nodes x and y, $T_{x.hpos}$ and $T_{y.hpos}$ share a common prefix of length $x.depth$, y must be a descendant of x.

There are many different ways of representing the parent-child relationships of the nodes, as described in [Kur99]. The simplest is to let each node have a $firstchild$ and a $branchbrother$ pointer, forming *sibling lists*. To find the edge starting with a given character, you must traverse all the children of a node, following the $firstchild$ and $branchbrother$ pointers. This adds an extra factor $O(\sigma)$ to the child lookup time, giving total costs of $O(m\sigma)$ and $O(n\sigma)$. (The alphabet size is often viewed as a constant and ignored.)

Chapter 3 also describes other solutions. If you use hash tables, you can get $\Theta(1)$ expected lookup, or if you use balanced search trees or sorted arrays, you can get $O(\log \sigma)$ lookup. The reason why sibling lists is often used, is that they are space efficient and simple. Given very small alphabets, as in DNA, it is near optimal. There are many possibilities for combining

various techniques, which is one of the reasons there are so many variants on suffix trees and arrays [FG04].

2.2.4 Searching in suffix trees

Searching for a pattern P in the suffix tree of a text T is straight forward. You start at the root node, and match the characters of P with characters on edges of of the tree, until either the entire pattern is matched, or there is no outgoing edge on the correct character, meaning there were no hits. A function `findnode` for searching for a pattern in a suffix tree is shown in figure 2.3. It returns the node representing the longest common prefix with P found in T, and the length of this prefix. Notice that this search might end up on an edge between two nodes.

```
def findnode(P):
    node = root
    i = 0
    while i < len(P):
        next = node.getchild(P[i])
        if not next:
            break
        i += 1 # know first char matches
        while i < next.depth and i < len(P):
            if P[i] != T[next.hpos + i]:
                return node, i - 1
            i += 1
        node = next
    return node, i
```

Figure 2.3: Python program for searching in a suffix tree.

If all of P is matched, the paths from the root to the nodes in the subtree below all have P as a prefix, by the definition of the suffix tree. For a given node, the path from the root to the node spells $T_{hpos...hpos+depth}$. This means all the $hpos$ found in the subtree represent starting positions where P is a substring of T. The $hpos$ of leaf nodes are unique, and no internal node has a $hpos$ not seen on a leaf node in the subtree below it. Therefore, the $hpos$ of the leaf nodes gives the complete hit set. Code for the function `findhits`, finding all hits for a pattern, is given in figure 2.4.

If there is an internal node which has a $hpos$ not seen in any leaf below it, there are three possibilities: Either T_{hpos} is not represented by a leaf in the tree, or $depth = n + 1 - hpos$, meaning it is both an internal node and a leaf node, or, remark 2.8 does not hold. In either case, it is not a valid suffix tree for T.

Theorem 2.9: If $node$.`getchild()` takes $\Theta(1)$ time, finding all occurrences of a pattern P of length m in a text T using the function `findhits` takes $O(m + occ)$ time.

Proof: In the function `findnode`, both the while loops increment the variable `i`, which starts at zero and is never more than m. All operations inside the loops take $\Theta(1)$ time, giving a total of $O(m)$. In the function `findhits`, the subtree below is traversed depth first. This subtree has occ leaf nodes. Since all internal nodes are branching, there are at most $occ - 1$ internal nodes in the subtree, and the traversal takes $\Theta(occ)$ time. Hence, the total running time is $O(m + occ)$.

```
def findhits(P):
    hits = []
    node, matched = findnode(P)
    if matched < len(P):
        return hits
    else:
        hits.append(node.hpos)
        stack = [node]
        while stack:
            node = stack.pop()
            for child in node.getallchildren():
                if child.isleaf:
                    hits.append(child.hpos)
                else:
                    stack.append(child)
        return hits
```

Figure 2.4: Python program for finding all hits.

2.2.5 Building suffix trees

An online demo on linear time construction of suffix trees can be seen at `http://grimsmo.dyndns.org/~nils/suffix/`. This was created for a lecture held by the author and a fellow student, Hans Christian Falkenberg, in the course TDT4125 Algorithm construction at NTNU. The slides from the lecture can be found at `http://www.idi.ntnu.no/~nilsgri/diploma/suffix_lecture.pdf`.

The linear time suffix tree construction algorithm of [McC76] is a bit complex and difficult to explain, as is other construction algorithms known to the author. Examples and informal explanations are given between the formal definitions. The algorithm adds the suffixes in decreasing order of their length. T_0 is added first, and T_n last. If the suffixes are naively added to the tree one by one, the running time is $O(n^2)$, as the total length of all the suffixes is $\sum_{i=1}^{n+1} i \in \Theta(n^2)$. Linearity is achieved by avoiding to redo work which has been done before. To make it possible, we need to introduce suffix links:

Definition 2.10 ([McC76]): If internal node $\overline{X\alpha}$ represents the string $X\alpha$, where X is a single character, and α is a possibly empty string, $\overline{X\alpha}$ will have a suffix link to the internal node $\overline{\alpha}$.

Remark 2.11*:* During the suffix tree construction, only the internal node created in the previous iteration will fail to have a suffix link.

Example 2.2*:* In figure 2.2, $\overline{\text{ab}}$ has a suffix link to $\overline{\text{b}}$, and $\overline{\text{aba}}$ has a suffix link to $\overline{\text{ba}}$.

The algorithm can be explained informally as follows. In each step you add a suffix of the string. After this string is added, there will be a unique path from the root node to a new leaf node, representing this suffix. The path is guaranteed to be unique because we have an end marker which is does not exist anywhere else in this string. The end marker does not occur in the same position in any of the suffixes, as they all have unique lengths. To add a suffix, you search downwards through the tree as long as the prefix of the suffix seen so far exists in the tree. You have then found the longest common prefix with any of the suffixes added previously.

You are now either in a node, or you are on an edge. If you are on an edge, you split it, and insert a new internal node. Finally, you add a leaf child, representing the rest of the suffix.

During the construction, we need the following definitions:

- *locus* of step i — The internal node representing $head_i$.
- *contracted locus* of step i — The last internal node on the path from $root$ to $leaf_i$ which existed before iteration i.

In figure 2.5 you see the suffix tree for the string "abbabaabab" after iteration 6 adding $T_6 =$ abab, and iteration 7 adding $T_7 =$ bab. The nodes created in the current iteration are drawn filled. The contracted locus and locus of iteration 6 are the nodes $\overline{\text{ab}}$ and $\overline{\text{aba}}$, while for iteration 7 it is $\overline{\text{ba}}$ and $\overline{\text{bab}}$.

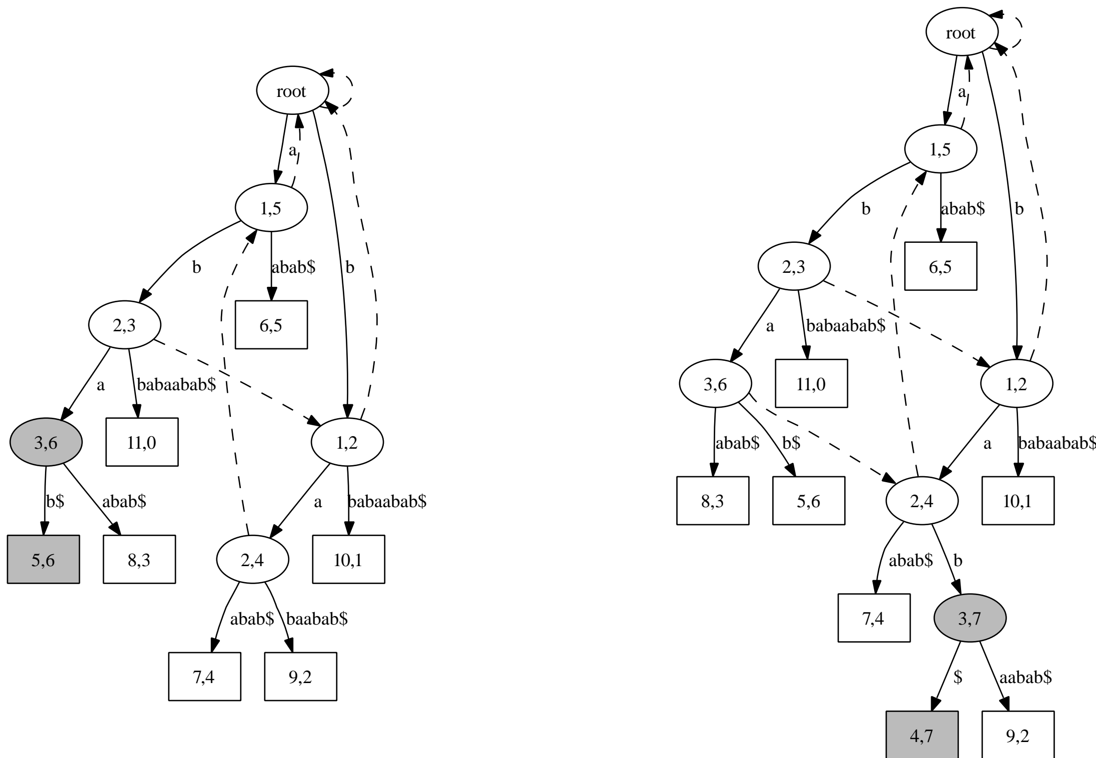


Figure 2.5: Suffix tree for "abbabaabab" after step 6 and 7

The following explanation is visualised in figure 2.6.

The formal algorithm is as follows: Assume that we are in iteration i of the algorithm, and that $head_{i-1}$ from the last iteration was $\chi\alpha\beta$, where χ, α and β are possibly empty strings. Let χ have a length of at most one. Let $\overline{\chi\alpha}$ be the *contracted locus*, and $\overline{\chi\alpha\beta}$ the *locus* of iteration $i-1$. χ is empty only if $\chi\alpha\beta$ is empty.

Begin iteration i by following the suffix link of $\overline{\chi\alpha}$ to $\overline{\alpha}$. $\overline{\chi\alpha}$ must have a suffix link, as a it was created in an iteration previous to the last. From $\overline{\alpha}$, you `rescan` the string β, and from the node

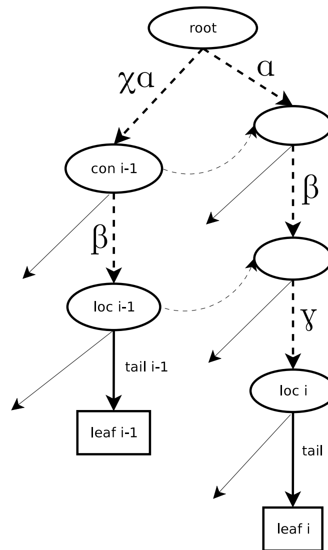


Figure 2.6: Conceptual suffix tree building

$\overline{\alpha\beta}$, you `scan` the string γ. $\alpha\beta\gamma$ is the longest common prefix of T_i with any suffix T_j added previously, meaning $j < i$. A Python implementation of the McCreight construction algorithm is given in figure 2.7. The function `rescan`, given in figure 2.8, jumps from node to node, following a path which you know is in the tree. The function `scan`, looks at every character, finding a string γ of unknown length. These functions will create a new node if the locus of T_i did not already exist. The function build adds a leaf node in each iteration.

```
def build():
    last_locus = root
    last_contracted = root
    for i in xrange(0, N + 1):
        old, betaend = rescan(last_contracted, last_locus, i)
        last_locus.sufflink = betaend
        contracted, locus = scan(old, betaend, i)
        locus.addchild(LeafNode(depth = n - i, hpos = i))
        last_locus = locus
        last_contracted = contracted
```

Figure 2.7: Python program for building suffix tree

Example 2.3*:* In iteration 7 in figure 2.5, $head_6$ from iteration 6 was aba. Using the naming from figure 2.6, χ = a, α = b and β = a. We `rescan` β = a, and `scan` to find γ = b. We conclude that $head_7 = \alpha\beta\gamma$ = bab. The string “abbabaabab” was chosen because it has non-empty χ, α, β, and γ in a single step.

During `rescan` you jump from node to node, until you reach the position $\overline{\alpha\beta}$. If it is not an explicit node, you create it. After the rescan, you set the suffix link of $\overline{\chi\alpha\beta}$ to $\overline{\alpha\beta}$.

Lemma 2.12: If $\chi\alpha\beta$ is the *head* of T_{i-1}, the string $\alpha\beta$ exists in the tree, at the start of iteration i of the suffix tree construction,

Proof: Since $\chi\alpha\beta$ was the *head* of T_{i-1}, there exists a $j < i - 1$, such that T_j has $\chi\alpha\beta$ as a

prefix. Since suffixes are added to the tree in decreasing order of their length, T_j is represented in the tree in iteration $i-1$. As $j < i-1$ and $j+1 < i$, T_{j+1} must be represented in the tree. Therefore the string $\alpha\beta$ must exist in the tree before iteration i, and `rescan` can jump from node to node, as the first character on all downward edges from a given node are unique.

Remark 2.13*:* If a new node was created during `rescan`, γ will be empty. The reason is as follows: $\chi\alpha\beta$ was the longest common prefix of T_{i-1} and T_j for any $j < i-1$. This means $T_{i-1}[|\chi\alpha\beta|] \neq T_j[|\chi\alpha\beta|]$, and $T_i[|\alpha\beta|] \neq T_{j+1}[|\alpha\beta|]$. If the γ in iteration i is non-empty, there must be a T_k, $j \neq k < i$, such that $|lcp(T_k, T_i)| > |\alpha\beta|$. But then, $T_i[|\alpha\beta|] = T_k[|\alpha\beta|] \neq T_{j+1}[|\alpha\beta|]$, $lcp(T_k, T_{j+1}) = \alpha\beta$, and since $k < i$ and $j+1 < i$, $\overline{\alpha\beta}$ must exist in the tree before iteration i.

Lemma 2.14 ([McC76]): The total time used on rescanning is $O(n)$.

Proof: For every old node encountered during the rescan, there is a non-empty part of the string T that will be part of the $\chi\alpha$ of next round, and hence never be rescanned again. Therefore, the number of nodes encountered during the rescan is $O(n)$. The work done in rescan is linear on the number of nodes seen.

Code for the `rescan` function is given in figure 2.8.

```
def rescan(last_contracted, last_locus, i):
    start = last_contracted.sufflink
    if last_contracted == last_locus:
        # beta was empty
        return start, start
    charsleft = last_locus.depth - last_contracted.depth
    if last_contracted == root:
        # alpha was empty
        charsleft -= 1
    node = start
    while charsleft > 0:
        old_node = node
        node = node.getchild(T[i + node.depth])
        charsleft -= node.depth - old_node.depth
    if charsleft < 0:
        # we stopped on an edge between two nodes
        new = BranchNode(depth = node.depth + charsleft, hpos = i)
        old_node.replacechild(node, new)
        new.addchild(node)
        return old_node, new
    else:
        # we stopped on a node
        return node, node
```

Figure 2.8: `rescan` function

After you have rescanned the string β, you must `scan` downward from the node $\overline{\alpha\beta}$, until you have found the entire of $head_i$, denoted $\alpha\beta\gamma$. The function scan looks at every character on the path from $\overline{\alpha\beta}$ to $\overline{\alpha\beta\gamma}$. If $\overline{\alpha\beta\gamma}$ does not exist explicitly, you create it. Finally, you create the child $leaf_i$ of $\overline{\alpha\beta\gamma}$. Code for the `scan` function is given in figure 2.9.

Lemma 2.15 ([McC76]): The total time used on scanning is $O(n)$.

Proof: Every character successfully scanned will be a part of the $\chi\alpha\beta$ of next iteration, and hence never scanned again.

```
def scan(old, beta_end, i):
    if old != betaend:
        return old, betaend
    node = beta_end
    while True:
        parent = node
        node = node.getchild(T[i + node.depth])
        if not node:
            return parent, parent
        matched = 0
        for k in xrange(parent.depth, node.depth):
            if T[node.hpos + k] != T[i + k]:
                newnode = BranchNode(depth = parent.depth + matched, hpos = i)
                parent.addchild(newnode)
                newnode.addchild(node)
                return parent, newnode
            else:
                matched += 1
```

Figure 2.9: `scan` function

Together, following the suffix link from the last locus, running `rescan` and `scan`, finds the longest common prefix of of T_i with any T_j with $j < i$.

Theorem 2.16 ([McC76]): The algorithm `build` builds a suffix tree for a string T. The paths from the root to the leaves in this tree have a one to one correspondence to the suffixes of the string.

Proof: In iteration i you add one leaf node to the tree, such that the path from the root to this leaf spells T_i. Since it is a tree, this results in only one new path. If an internal node is created, it is inserted to split an edge, and then given a leaf child. Hence all internal nodes have at least two children.

Theorem 2.17 ([McC76]): The suffix tree construction runs in $\Theta(n)$ time.

Proof: Excluding the cost of `rescan` and `scan`, the cost of the operations in each of the $n + 1$ iterations is $\Theta(1)$, totalling to $\Theta(n)$. As the time spent in `rescan` and `scan` is $\Theta(n)$, the total cost of the construction algorithm must be $\Theta(n)$.

Note that all this is under the assumption that child lookup and insertion is $\Theta(1)$. If for example sibling lists are used, construction takes $O(n\sigma)$ time, and search is $O(m\sigma + occ)$. This is discussed in chapter 3.

2.3 Suffix arrays

Suffix arrays (also known as PAT-arrays [GBYS92]) where discovered almost twenty years after the suffix tree. The suffix array is a more space efficient, and in many ways simpler, alternative

to the suffix tree. [MM91] gives a thorough description of construction and use. The build algorithm in the article is $O(n \log n)$, and search is done in $O(m + \log n + occ)$ time. Since their discovery, a lot of research has been done on suffix array construction and use.

Definition 2.18: A suffix array SA for a string T of length n padded with \$ is a list of integers such that $T_{SA[i]} < T_{SA[j]}$ for $0 \leq i < j \leq n$.

In other words, the suffix array gives the order of the suffixes of T. As long as the end marker \$ is unique, no two suffixes are lexicographically equal.

Example 2.4*:* The suffix array for the string “abbabaabab” is shown in figure 2.10. If you look at the suffix tree in figure 2.2, you notice that the $hpos$ of the leaf nodes in the lexicographically ordered depth first walk of the tree are the elements of the suffix array for the string.

i	$SA[i]$	$T_{SA[i]}$
0	10	$
1	5	aabab$
2	8	ab$
3	3	abaabab$
4	6	abab$
5	0	abbabaabab$
6	9	b$
7	4	baabab$
8	7	bab$
9	2	babaabab$
10	1	bbabaabab$

Figure 2.10: Suffix array for the string “abbabaabab”.

2.3.1 Searching in suffix arrays

Searching with the suffix array is the same as performing a binary search in the leaf nodes of the suffix tree. Note that a straight forward implementation gives $O(m \log n + occ)$ worst case time complexity, as you may compare almost the entire pattern P with $T_{SA[i]}$ for every jump in the binary search.

This can be avoided by knowing how much of P matched in last iteration, and by pre-generating information about the longest common prefixes of the centre and the left and right border in the binary search. This is described in [MM91]. If cleverly implemented, you need about two bytes extra per input character to get $O(m + \log n + occ)$ search. In practise, a simplified variant without extra data structures is often used. If you maintain how much is matched with both the left and the right border in the search, you know that the match with the centre is at least the minimum of these two values. This gives good search performance on real data (see [MM91]), and no overhead on construction, but has worst case search time $O(m \log n + occ)$.

When you want to enlist all hits for a pattern, you need to binary search for both an index i with the property that P is a prefix of $T_{SA[i]}$ and, $i = 0$ or $T_{SA[i-1]} < P$ (the "leftmost" hit), and an index j with the property that P is a prefix of $T_{SA[j]}$ and, $j = n$ or $P < T_{SA[j+1]}$ (the "rightmost" hit). Code for finding the leftmost hit is given in figure 2.11. Finding the rightmost hit is done similarly.

```
def search(T, SA, P):
    left = 0
    right = len(T) - 1 # assuming already padded
    matched = 0
    l_matched = 0 # matched on left border
    r_matched = 0 # matched on right border
    while left < right:
        middle = (left + right) / 2
        j = matched
        while True:
            if j == len(P) or P[j] < T[SA[middle] + j]:
                right = middle
                r_matched = j
                break
            elif P[j] > T[SA[middle] + j]:
                left = middle + 1
                l_matched = j
                break
            j += 1
        matched = min(l_matched, r_matched)
    while matched < len(P) and T[SA[left] + matched] == P[matched]:
        matched += 1
    if matched == len(P):
        return left
    else:
        return -1
```

Figure 2.11: Searching for the "leftmost" match of a pattern.

Since the $O(m + \log n + occ)$ search of [MM91] is not used in the following experiments, it will not be discussed further. It gives a higher construction cost, but significantly faster search only on artificial worst-case data.

2.3.2 Building suffix arrays

Suffix array construction has received a lot of attention the last few years. Doing a radix sort of all the suffixes gives a running time of $O(n^2)$. [MM91] shows how to sort the suffixes in $O(n \log n)$ time. In 2003, [KA03], [KS03] and [KSPP03] independently discovered similar algorithms for doing it in $\Theta(n)$ time. Recently, however, attention has been given to various $O(n^2)$ algorithms which are much faster than the known linear algorithms in practise. The fastest so far is the Bucket Pointer Refinement (BPR) algorithm of [SS05].

The BPR algorithm will be used for index construction in the comparisons in chapter 5. Since the software from the authors of [SS05] will be used directly, the algorithm will not be discussed further here.

2.4 Hierarchic indexes

To make an index for a dynamic set of documents, you must somehow include the changes to the set in your index. Some index structures are dynamic of nature, and allow updates. The suffix tree is such a structure. How to adapt the suffix tree to dynamic document sets is explored in chapter 3. Other structures are very static of nature. For example, the suffix array does not allow updates. The cost of inserting one value into the middle of the array has a cost of $O(n)$, as you must move $O(n)$ entries to the right. Since the cost of building a suffix array has almost linear cost in practise, updating the suffix array is not much cheaper than just rebuilding it (given that the array representation stays in its basic form).

To solve our problem of searching dynamic document sets using static indexes, we must find a way to include our updates. One extreme is to maintain one index, which is rebuilt on every change. This gives effective search, but a very high inclusion and deletion cost. Another extreme is to maintain one index for every document. This gives low inclusion and deletion cost, but expensive search.

Using something in between these two extremes gives both effective searches and updates. You maintain a set of indexes of different sizes, which are rebuilt with proportional intervals. In a practical application, the biggest index might be rebuilt once a week, but the smallest every minute. Deletions can be handled using a blacklist, leaving the document data in the index until it is rebuilt.

Two principles for hierarchies of static indexes are explored in this chapter. Both are taken from [OvL80], which is a theoretical analysis of general dynamic searching problems. Let them be called method 1 and method 2. Both methods have a parameter k used to prioritise between search and update cost. In short terms, method 1 maintains indexes of sizes k^i, and up to $k-1$ indexes for each $i \in \mathbb{Z}^+$, while method 2 maintains one index of size $b_i \cdot k^i$ for each i, with $b_i < k$. Method 1 with high k is fastest on updates, while method 2 with high k is fastest on searches.

We use the following terminology from [OvL80]:

- $Q_S(n)$ is the cost of querying a static structure with n elements.
- $P_S(n)$ is the cost of building a static structure with n elements.

Likewise, we use $Q_D^1(n)$, $P_D^1(n)$, $Q_D^2(n)$ and $P_D^2(n)$ as the cost for the dynamic indexes using method 1 and 2. In the following discussion, documents are considered to have unit length. This differs from the practical implementation in chapter 4.

2.4.1 Method 1

Method 1 is defined as follows:

Definition 2.19 ([OvL80]): A dynamic index using method 1 consists of a set of static indexes of sizes $k^i \mid i \in \mathbb{Z}^+$ for a given constant k, with less than k indexes of each size k^i.

The index is maintained by upholding this property. When a document is added, it is saved as an unsorted index of size $k^0 = 1$. If there are now k indexes of size 1, they are merged into an index of size $k^0 k = k$. If there are now k indexes of size k^1, these are merged into an index of size k^2, and so on. Note that you only have to build in the last merging step, after gathering all the contents of this final static index.

A set of indexes for $k = 3$ is shown in figure 2.12. When a document is added, the 3 indexes of size 1 are merged to an index of size 3. Now there are 3 indexes of size 3, which are merged to an index of size 9.

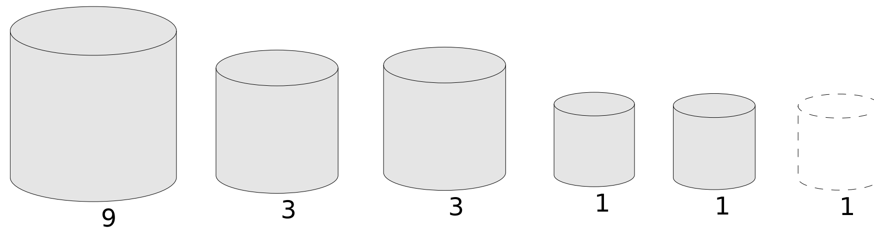


Figure 2.12: Method 1 for hierarchic indexes

Theorem 2.20 ([OvL80]): The cost of a query to a dynamic index of size n using method 1 is $Q_D^1(n) = O(k \log_k n) Q_S(n)$, and the cost of building the index iteratively is $P_D^1(n) = O(\log_k n) P_S(n)$.

Intuitively, $Q_D^1(n)$ has a $O(k \log_k n)$ factor because there are in the worst case $k-1$ indexes of each of the $\lfloor \log_k n \rfloor + 1$ sizes, and each of these must be queried.

$$Q_D^1(n) \in \sum_{i=0}^{\lfloor \log_k n \rfloor} O(k) Q_s(k^i) \subseteq \sum_{i=0}^{\lfloor \log_k n \rfloor} O(k) Q_s(n) = O(k \log_k n)\ Q_S(n)$$

Each document will be built into an index of size k^i at most once, and the cost per document for building a static index of size m is $P_S(m)/m$. Hence the total cost per document for a document throughout its history is $\sum_i O(1) P_S(k^i)/k^i$

$$P_D^1(n) \in n \sum_{i=0}^{\lfloor \log_k n \rfloor} \frac{O(1)\ P_S(k^i)}{k^i} \subseteq n \sum_{i=0}^{\lfloor \log_k n \rfloor} \frac{O(1)\ P_S(n)}{n} = O(\log_k n)\ P_S(n)$$

Remark 2.21*:* Notice that when setting $k = n$, $P_D^1(n) = O(1) P_S(n)$, and there is no overhead building a dynamic index, but $Q_D^1(n) = O(n) Q_S(n)$, giving a very high query cost.

2.4.2 Method 2

Method 2 differs from method 1 in that no two indexes have the same size. It is similar to k-ary number systems.

Definition 2.22: A dynamic index of size n using method 2 consist of i static indexes of sizes $k^i b_i \mid 0 \leq b_i < k, 0 \leq i \leq \lfloor \log_k n \rfloor$.

A set of indexes for $k = 3$ is shown in figure 2.13. If one document is added, it will be merged into the smallest index. If another is added, it will be merged into the index with capacity 18 together with the two smallest indexes.

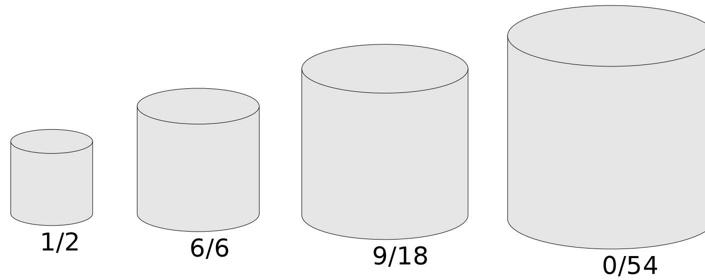


Figure 2.13: Method 2 for hierarchic indexes

Adding a document to the index is similar to adding 1 to a k-ary number: Find the lowest digit which has a b_j less than $k-1$. For all i below j, $b_i = k - 1$, which means that the value of $1 + \sum_{i=0}^{j-1} k^i b_i = 1 + (k-1) \sum_{i=0}^{j-1} k^i = 1 + (k-1)(k^j - 1)/(k-1) = k^j$. We set $b_i = 0$ for all $i < j$, and increase b_j by one. When indexing, this equals moving the contents of indexes $0 \ldots j-1$ into index j.

If documents are of varying size, we do things a bit different than when adding k-ary numbers. We find the smallest index which has room for its own content, the new document, and contents of all smaller indexes. Still only one index will be built during a document inclusion.

Theorem 2.23 ([OvL80]): The cost of a query to a dynamic index of size n using method 2 is $Q_D^2(n) = O(\log_k n) Q_S(n)$, and the cost of building the index iteratively is $P_D^2(n) = O(k \log_k n) P_S(n)$.

The query time is trivial from the fact that there are at most $\lfloor \log_k n \rfloor + 1$ indexes with a maximal query cost $Q_S(n)$.

$$Q_D^2(n) \in \sum_{i=0}^{\lfloor \log_k n \rfloor} O(1)\ Q_S(k^i(k-1)) \subseteq \sum_{i=0}^{\lfloor \log_k n \rfloor} O(1)\ Q_S(n) = O(\log_k n) Q_S(n)$$

With method 2, a document may be added to an index more than once, at most $k-1$ times. Hence, the indexing cost is

$$P_D^1(n) \in n \sum_{i=0}^{\lfloor \log_k n \rfloor} O(k-1) \frac{P_S(k^i(k-1))}{k^i} \subseteq n \sum_{i=0}^{\lfloor \log_k n \rfloor} O(k) \frac{P_S(n)}{n} = O(k \log_k n)\, P_S(n)$$

Remark 2.24*:* Notice that the costs for $Q_D^2(n) = P_D^1(n)$, and $P_D^2(n) = Q_D^1(n)$. We can get $Q_D^2(n)$ arbitrarily low by increasing k, at the cost of more expensive document inclusions.

An implementation of method 1 and 2 is discussed in chapter 4.

Chapter 3

Multi-document Dynamic Suffix Trees

This chapter describes a data structure for maintaining a substring index for a dynamic set of documents.

3.1 Definition

We begin by stating the goals of the data structure. We want to maintain a substring index of a set of d documents $D = \{T^0, T^1, \ldots, T^{d-1}\}$ of total length N (when the strings are padded with an end marker). Given a document T^k of length n_k, and a query string P of length m, we want the following functionality:

- Insert document T^k in $\Theta(n_k)$ time.
- Delete document T^k in $\Theta(n_k)$ time.
- Find one occurrence of P in D in $O(m)$ time.
- Find all occ occurrence of P in D in $O(m + occ)$ time.

We also want the data structure to be space efficient.

Updating a document is the same as deleting the old version from the index and inserting the new. [McC76] shows how to update the suffix tree for a document in time expected sub-linear on the length of the document, but linear in the worst case. For typical applications of indexes for document sets, your input would usually be a new version of the document, and a full scan to find changes is necessary anyway. Therefore, incremental editing along the lines of [McC76] is not explored.

Definition 3.1: A multi-document dynamic suffix tree (MDST) for a set of documents $D = \{T^0, T^1, \ldots, T^{d-1}\}$ is a compressed trie of all suffixes of $T^k\$_k$ for all $k \in 0 \ldots d-1$, where all end markers $\$_k$ are unique symbols.

End markers are preferably kept unique, to simplify tree maintenance. If we use a common end marker \$, and two documents end with the string α, there will be at least two suffixes that share the node $\overline{\alpha\$}$ as a terminal node. We could solve this by either having outgoing edges of length zero to leaf nodes containing document IDs and head positions, or we could keep a list of such pairs in $\overline{\alpha\$}$. Either solution gives a more complex and space consuming implementation. Unique end markers are simple and fast, if implemented well.

Remark 3.2*:* Instead of having explicit end markers, increasing the size of your alphabet by d, you simulate this uniqueness in your implementation, where all strings are extracted from $depth$ and $hpos$ of parent and child nodes. ($depth$ and $hpos$ were defined on page 14).

3.2 Document insertion

The algorithm for inserting a document into the tree is identical to that of `build` in figure 2.7, except that all nodes are given a document ID $docid$. The string on the edge between the nodes $parent$ and $child$ are taken from the document referenced by $child$, with the starting and ending point extracted as shown in example 2.1. The code for rescanning β (figure 2.8) and scanning γ (figure 2.9) is almost identical.

Inserting a document into a MDST has the same time complexity as building a suffix tree for this document.

3.3 Document deletion

Given a suffix tree for a string T of length n, you can traverse all suffixes of decreasing length in $\Theta(n)$ time. Code for this is given in figure 3.1. The proof for the running time is similar to that for `rescan` in figure 2.8 in chapter 2. For all internal nodes visited in downward traversal, there is a substring of T that will never be visited in downward traversal again.

```
def traverse(tree, T):
    node = tree.root
    for i in xrange(len(T)):
        while not node.isleaf:
            parent = node
            node = node.getchild(T[i + parent.depth])
        node = parent.sufflink
```

Figure 3.1: Traversing a suffix tree

Lemma 3.3: The leaf nodes for all suffixes are visited in the decreasing order of their length by the algorithm `traverse`.

Proof: By induction on i: Assume that $T_i = \chi\alpha\tau$ is seen in iteration i, where χ is at most one character and empty only if $\chi\alpha$ is empty, and $\overline{\chi\alpha}$ is the last internal node seen in the downward

traversal. Then $\overline{\chi\alpha}$ will have a suffix link to $\overline{\alpha}$. α must be a prefix of T_{i+1} since $\chi\alpha$ is a prefix of T_i, and hence downward traversal from $\overline{\alpha}$ will lead to the leaf $\overline{\alpha\tau}$, and $\alpha\tau = T_{i+1}$, since $\chi\alpha\tau = T_i$.

When you want to disassemble a suffix tree, you find and remove all the leaf nodes in some order, and maintain the property that all internal non-root nodes are branching. Then you must end up with a tree consisting of only the root node.

To remove a document T^p from the MDST, we traverse its suffixes in order longest to shortest, as in figure 3.1. In iteration i, we remove $leaf_i^p$ (with $hpos = i$) representing T_i^p. If the parent node has only one child left after the removal, it is merged into the grandparent. Python code for this is given in figure 3.2. If we do not have parent pointers in the internal nodes, we follow the suffix link of the grandparent node of last iteration, to make sure we always have a valid grandparent set. This gives a constant extra cost per iteration, and hence a linear extra total cost.

```
def deldoc(tree, docid, T):
    node = parent = tree.root
    for i in xrange(len(T)):
        while not node.isleaf:
            grandparent = parent
            parent = node
            node = node.getchild(T[i + parent.depth])
        parent.removechild(node)
        if len(parent.children) == 1 and parent != tree.root:
            onlychild = parent.getallchildren()[0]
            grandparent.replacechild(parent, onlychild)
        node = grandparent.sufflink
```

Figure 3.2: Deleting a string from a suffix tree

Lemma 3.4: If there is an internal node $\overline{\chi\alpha}$ for a non-empty χ, it will be deleted before an internal node $\overline{\alpha}$.

Proof: Assume we are removing the document T^p. If there is an internal node $\overline{\chi\alpha}$ for a non-empty χ, and $\chi\alpha$ is a prefix of T_j^p, there must be a q and i such that $\chi\alpha$ is the longest common prefix of T_j^p and T_i^q, and either $p \neq q$ or $j < i$. (Or else the node $\overline{\chi\alpha}$ would not exist.) If $\overline{\chi\alpha}$ is merged with its parent, we must be in iteration j of removing T^p. Since $j < j + 1$, and $j < i < i + 1$ or $p \neq q$, by lemma 3.3, there are at least two leaf nodes in the subtrees below $\overline{\alpha}$ that are not visited before the leaf node representing T_j^p, and $\overline{\alpha}$ must still be in the tree when $\overline{\chi\alpha}$ is deleted.

Lemma 3.5: After iteration i, all internal nodes are deleted or branching.

Proof: At iteration 0, all internal nodes are branching, by definition 2.4. Before iteration i, all internal non-root nodes are deleted or branching by induction. During iteration i, the only nodes deleted are $leaf_i$ and possibly its parent. If the parent has only one child left after the deletion, the parent is also deleted, or it is still branching. If the parent is not the root, the grandparent of $leaf_i$ must have at least two children, by the inductive hypothesis. If the parent of $leaf_i$ is deleted, the single child will be merged into the grandparent, which will have the same number of children as before iteration i.

Theorem 3.6: The algorithm `deldoc` in figure 3.2 removes a document T^k from a suffix tree.

Proof: After iteration i, the leaf nodes representing suffixes $T^k_0, \ldots, T^k_i$ are deleted by lemma 3.3, and all internal non-root internal nodes are deleted or branching by lemma 3.5. After iteration n, all leaves from T^k are deleted, so no suffixes of the padded string exist in the tree. Since all internal non-root nodes have children, and all leaves refer to unremoved documents, all strings in the tree are substrings of unremoved documents.

After deleting a document from the MDST, it might be left with internal nodes with $docid$ referring to this document. How to deal with this is described in section 3.7.

3.4 MDST implementation

Almost all of the code for the MDST was prototyped in Python, and then rewritten in C++. Different solution were tried concerning how nodes are represented, how the parent-child relationship is implemented, and how document IDs were updated. This is described in the following sections.

3.5 Node model

The implementation used in the following work is heavily inspired by [Kur99]. The observations added are those related to updatability and managing multiple documents.

3.5.1 `struct` nodes

The easiest way to model a tree is by allocating a memory chunk for each node, using structures built into the language, a `struct` in C, or an object in an object oriented language. This causes a space overhead per node needed for memory management.

3.5.2 Compact array nodes

Instead of using such structures, you can manually allocate chunks of memory, and pack your nodes into them. The simplest and fastest way of storing the array of nodes is defining each field in a node to be one computer word. The size of a node will then be the number of fields multiplied by the size of a word. Another way is letting each field require an individual number of bytes. This requires a few more operations when storing or extracting a value. To use a minimum amount of space, you might want to use an individual number of bits per field. Note that a higher addressing resolution requires more bits for storing a node address in a suffix link

or a child pointer. To avoid this problem, you could let the size of a node be divisible by some factor, and use a lower resolution for node addresses, and a higher resolution for node content.

As there is a variable number of internal nodes added per document, and some internal nodes are left when removing a document, all internal nodes are preferably put into one array. References to internal nodes can be a node number, or an offset into the array. The latter is needed if nodes are of variable size. Typical data stored in an internal node is $docid$, $depth$, $hpos$, $sufflink$ and possibly $parent$. In addition you need information about its children. This is described later in section 3.6.

Leaf nodes have different properties. When you add a document of length n, you know that you will need $n + 1$ leaf nodes. When you remove a document, you also know that all of the leaf nodes will be deleted, as shown in chapter 3. Therefore, we could allocate one array for the leaf nodes of each document, which is deallocated when the document is removed. This would give a space overhead for the memory manager linear on the number of documents, which is usually affordable.

If the leaves are inserted into the array in the order in which they are created, the $hpos$ and $depth$ of a leaf can be deduced from the position in the array, and the $docid$ is given by which array the leaf resides in. If a leaf is placed in position j, then

- $leaf.hpos = j$
- $leaf.depth = (n + 1) - j$

where n is the unpadded length of the related document.

You can see the space usage for the node models in figure 5.2(d) on page 57.

3.5.3 Large and small internal nodes

As observed in [Kur99], a suffix tree contains a lot of redundant information on average. Consider an iteration of the construction algorithm, illustrated in figure 2.6. If an internal node $\overline{\chi\alpha\beta}$ is created in iteration $i - 1$, and an internal node $\overline{\alpha\beta}$ is created in iteration i, we know the following (adapted from [Kur99]):

- $\overline{\chi\alpha\beta}.docid = \overline{\alpha\beta}.docid$
- $\overline{\chi\alpha\beta}.depth = \overline{\alpha\beta}.depth + 1$
- $\overline{\chi\alpha\beta}.hpos = \overline{\alpha\beta}.hpos - 1$
- $\overline{\chi\alpha\beta}.sufflink = \overline{\alpha\beta}$

So, when a chain of nodes with following $hpos$ are created during the addition of a document, we can compact the nodes in the chain. We name the last node a large node, and each of the other nodes in the chain small nodes. For each of the small nodes, we record the distance $dist$ to the large node, which is the difference in $hpos$ and $depth$. We know that:

- $small.docid = large.docid$
- $small.depth = large.depth + small.dist$
- $small.hpos = large.hpos - small.dist$
- $small.sufflink = small + 1$

If the nodes are saved consecutively in an array, $small.dist$ multiplied with the size of a small node will also be the distance to the large node in the array. Node that all nodes in a small–large chain always refer to the same document.

Example 3.1*:* Consider the suffix tree for the string "ababa", shown in figure 3.3. The internal nodes $\overline{\text{aba}}$ $(docid = 0, depth = 3, hpos = 2, sufflink = \overline{\text{ba}})$, $\overline{\text{ba}}$ $(docid = 0, depth = 2, hpos = 3, sufflink = \overline{\text{a}})$ and $\overline{\text{a}}$ $(docid = 0, depth = 1, hpos = 4, sufflink = root)$, will be created consecutively by the algorithm `build` in figure 2.7 on page 19. We see that we only need to save the $docid$, $depth$, $hpos$ and $sufflink$ of $\overline{\text{a}}$. An array containing the internal nodes for this tree is shown in figure 3.4

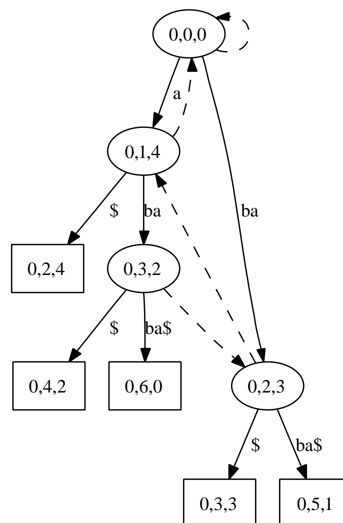


Figure 3.3: Suffix tree for the string "ababa"

Remark 3.7*:* One bit of the $docid$ field of a large node, and one bit of the $dist$ field of a small node could be reserved for distinguishing between large and small nodes. These fields should have the same offset in the nodes in the implementation.

0	1	2	3	4	5	6	7	8	9
root				$\overline{\text{aba}}$	$\overline{\text{ba}}$	$\overline{\text{a}}$			
0	0	0	0	2	1	0	1	4	0
$docid$	$depth$	$hpos$	$suflink$	$dist$	$dist$	$docid$	$depth$	$hpos$	$suflink$

Figure 3.4: Large and small internal nodes in the suffix tree for “ababa”

Note that small and large nodes was only implemented for the linked hash map parent-child model. The reason for that is that in this model, small nodes need no space (see 3.6.2.3). With sibling lists and variable size internal nodes, memory management is more complex, as described below.

3.5.3.1 Re-using space for internal nodes

Internal nodes are also added and removed, as the set of indexed documents changes. To maintain the linearity in space, the space freed deleting nodes must be reused. If we implement only large nodes (see section 3.5.3), all stored items have the same size. We can keep track of the free space by maintaining a linked list of all free node positions. The $nextfree$ pointers of the elements of the linked list can be saved in the free node positions, requiring no extra space. Allocating and deallocating space for a node takes $\Theta(1)$ time.

If you have both large and small nodes, you get two problems. The size of the next chunk available might not be the same size as the element you want to save. Secondly, a chain of small nodes and their large node must be saved consecutively.

The first problem can be solved by making sure the size of a large node is a multiple of the size of a small node. When creating a chain of small nodes, you make sure that the space consumed by the small nodes is a multiple of the size consumed by a large node. If, for example, the size of a large node is twice the size of a small node, the number of small nodes in a chain must be 0 modulo 2. This reduces the small to large node ratio on average.

The second problem is not trivial. When creating small nodes, you do not know how long the chain of nodes is going to be, and hence do not know the amount of space required to save the entire chain. One possibility is creating the chain of nodes in a temporary array, and copying them into the global array when the chain is finished. This leads to a new problem: How do we locate consecutive free space in the array? If free node locations are saved in a linked list, chunks of free space cannot be merged without traversing the entire list, giving $O(N)$ time cost for node allocation. If you keep a doubly linked list, you can merge two free chunks in $\Theta(1)$ time. As long as all nodes have space for saving two node offsets, this works.

You still have the problem of finding a free chunk of proper size when inserting a chain. If there are F free chunks, this takes $O(F)$ time, and if the memory is very fragmented, this is a high

price to pay.

Instead of re-solving this general problem, you could use the space allocation routines inherent in your programming language. But this might give a higher space cost than not using small nodes at all. Luckily, for the linked hash map child model discussed in section 3.6, small nodes take zero space in the node array. Only large nodes need to be stored explicitly, meaning the simple linked list memory management can be used.

3.5.3.2 Deleting nodes from chains

If the large node in a chain is deleted before a small node, the small node will lose its $docid$, $hpos$ and $depth$ information. Luckily, this will never happen.

Corollary 3.8: If $node_a$ is stored to the left of $node_b$ in a small–large chain, $node_b$ will not be deleted before $nobe_a$.

Proof: Directly from lemma 3.4 and the fact that $node_a = \overline{\alpha\beta}$ and $node_b = \overline{\beta}$ for some α and β.

Remark 3.9*:* When a node is deleted from the front of a small–large chain, the $backdist$ field of the large node must be decremented.

3.5.4 Leaf node addresses

If you limit the document size to 4GB, and the number of documents to 4G, $docid$, $depth$, and $hpos$ require 4 bytes each, while pointers to leaf nodes ($docid||hpos$) require 8 bytes each,

A more space efficient and flexible solution, is setting a cap on the total size of all documents inserted. To do this, you need a global node addressing. It is trivial for internal nodes, as they are allocated dynamically in a global array. A single bit can be used for determining whether a node is a leaf or an internal node.

It is a bit more complicated for leaf nodes, which are preferably saved in one dynamically allocated array per document. Instead of addressing the leaf by an index into an array, you can address a leaf by an absolute memory address. Then the size of the node pointers must be a full memory address. Absolute addressing should preferably not be used for internal nodes, as the address of a node may change when this array is reallocated as more space is required.

Another possibility is managing the space for the leaves yourself, letting all leaf pointers be offsets into a global array. This adds the complexity of fragmentation and reusing free chunks of variable sizes.

One problem with global leaf addressing is extracting the $hpos$, $docid$ and $depth$ fields. Given a leaf address, you must find which document it belongs to, and the offset from the first leaf. This

can be done by maintaining a sorted list or search tree of the starting offsets of the leaf arrays of all documents. Extracting a field from a leaf node would then take $O(\log d)$ time, where d is the number of documents in the MDST.

Alternatively, you can pad all documents so that their size is a multiple of r for a given integer r, and store the document ID for every rth position. Such a solution is described for suffix arrays for multiple documents in section 4.5.

In our implementation, one array of leaf nodes per document is used, and leaf indentifiers are $docid||hpos$.

3.6 Parenthood

Perhaps the most difficult choice for the implementation of a suffix tree is the $parent \rightarrow child$ relationship. These are the main goals:

- Finding the child of a node on a given character should take $\Theta(1)$ time.
- Adding or removing a child should take $\Theta(1)$ time.
- Listing all c children of a node should take $\Theta(c)$ time.
- The space required for storing all $parent \rightarrow child$ relationships should be $O(N)$. (There are at most $2N - 1$ nodes, each having at most one parent.)

In addition, the solution should be space efficient *in practise*. If you want to index 1GB of data, it makes a lot of difference whether you use $10n$ or $100n$ extra bytes of memory. We shall see that fulfilling all these requirements is not easy. A summary of the properties of the different solutions is given in section 3.6.5.

3.6.1 Sibling lists

A simple and quite effective solution is using sibling lists. Each node have a pointer to its "leftmost" child ($firstchild$), and each of the children have a pointer to a "right" brother ($branchbro$), forming a chain of siblings. This is one of the solutions described in [McC76] and [Kur99], and is used in the MUMmer software [MUM].

The fields required for the various node types are shown in figure 3.6. Notice that the $parent$ field is only needed with the eager document ID update model, described in section 3.7. The $backdist$ field is only needed if small nodes are used, as described in section 3.5.3.

3.6.1.1 Sibling list space cost summary

Example 3.2*:* Let the maximum field values be those given if figure 3.5. The fields required for each data type is shown in 3.6. Large and small nodes should be padded so the size of a small node is a multiple of the size of a large node. In the worst case, this gives $N \cdot 28 + N \cdot 6 = 34N$ bytes space needed.

Remark 3.10*:* [Kur99] gives a conjecture that there is an upper bound on the share of internal nodes that are large nodes is somewhere around $0.7n$. Indexing `bible.txt` from the Canterbury corpus gave $0.3n$ large and $0.4n$ small nodes. Indexing `world192.txt` gave $0.22n$ large, $0.43n$ small nodes. When doing a run with 60% inclusions and 40% deletions on 4KB texts from a zipf distribution, $0.15n$ small and $0.24n$ large nodes were seen after 5000 requests.

Variable	Bits	Bytes	Max	Fields
$maxdoc$	23	3	8M	$docid$
$maxlen$	24	3	16M	$depth$, $hpos$
$noderef$	48	6	-	$firstchild$, $branchbro$, $sufflink$
$maxdist$	7	1	128	$dist$, $backdist$

Figure 3.5: Example field sizes

Node type	Large node	Small node	Leaf node
Field	$firstchild$ $branchbro$ $(parent)$ $docid$ $depth$ $hpos$ $sufflink$ $backdist$	$firstchild$ $branchbro$ $(parent)$ $dist$	$branchbro$
Bytes	28	13 (14 padded)	4

Figure 3.6: Node fields required for storing node using sibling lists

3.6.2 Hash map

Hash map implementations are discussed in both [McC76] and [Kur99].

There is both a constant and a linear space overhead when maintaining a hash map. You have to store the size of the map and the fill rate. To keep the amortised cost of insertion and deletion at $\Theta(1)$ expected time, you must use a doubling scheme for increasing the size of the map. As the number of children per node is not known during construction, half of the slots in the maps are wasted in the worst case. Given that the number of children has a uniform distribution in $2 \ldots \sigma$, the expected amount of wasted entries in a hash map using doubling is $(\sum_{i=2}^{\sigma} 2^{\lceil \log_2 i \rceil} - i)/(\sigma - 2)$, which is 42.5 for an alphabet of size 256.

[McC76] and [Kur99] propose that you should maintain *one* hash map for the children of *all* nodes, using the ID of the parent concatenated with the symbol as key, and the ID of the child as value. We name this map $childmap$. We know that there is an upper bound on the total number of children, $2N - 2$, and if we known the string distribution the documents are taken from, we can make a lower educated guess.

3.6.2.1 Lost head positions

[Kur99] proposes using the $hpos$ as the ID of an internal node, as the $hpos$ of an internal node in a single suffix tree is unique. This is because in iteration i, at most one internal node is created, and it is given $hpos = i$. Using a combination of $docid$ and $hpos$ does *not* give uniqueness in our MDST implementation, as $hpos$ are inherited upwards when removing a document. As there are more leaves than internal nodes in a subtree, and all the leaves have a unique combination of $docid$ and $hpos$, it is possible to to maintain $docid||hpos$ uniqueness. Then you would lose the $\Theta(1)$ amortised update cost per node, as you possibly need to traverse the entire subtree to find a unique $docid||hpos$. A better idea is to use the node number, or the node offset in the case of variable size nodes, as keys in the hash. Document ID update is discussed in section 3.7.

3.6.2.2 Hash map keys

Since the internal nodes are allocated globally, the size of the keys are dependent on the maximum total document size, and not the sum of the maximum number of documents and the maximum document size. The space required for a key is this, plus the space required for storing a symbol. The size of a value in the hash map depends on the model for addressing leaves, described in section 3.5.4. The difference for the hash map solution, is that leaf nodes do not have to be stored explicitly anywhere, as they have no $branchbro$ field.

In our implementation we refer to a leaf by $docid$ and $hpos$.

3.6.2.3 Implicit small nodes

In the sibling list implementation with large and small nodes of section 3.6.1, the fields in the small nodes were $distance$, $firstchild$ and $branchbro$. With the hash map solution, the latter two are not required. The $distance$ field can be encoded in the node identifier, so all nodes in the chain of $large$ are identified by $distance||large$, giving a space overhead equal to the size of the distance field. Small nodes do not have to be stored explicitly. Note that this solves all the memory management problems discussed in section 3.5.3.1.

Figure 3.7 proposes a reference scheme for the various node types, which can be used for the values in the hash map. The $nextchar$ and $prevchar$ fields are used in the linked hash map described in section 3.6.3 The size required for a reference is one bit, plus the maximum of the space for a small node and for a leaf node.

Node type	Leaf bit	Distance	Address	Next character	Prev. character
Large node	0	$0 \dots 0$	$nodeno$	$nextchar$	$prevchar$
Small node	0	$dist$	$nodeno$	$nextchar$	$prevchar$
Leaf node	1	$docid$	$hpos$	$nextchar$	$prevchar$
	Leaf bit	Document	Head pos	Next character	Prev. character

Figure 3.7: Node references with implicit small nodes

3.6.3 Linked hash map

The hash map implementation so far gives expected $\Theta(1)$ child lookup and child insertion, but to list all children you need to traverse the entire alphabet, and look for children on all symbols, with a total cost of $\Theta(\sigma)$. This cost is critical when you want to list all occurrences of a string, and must traverse an entire subtree. The price for listing all occ occurrences of a pattern P of length m is $O(m + occ \cdot \sigma)$. A sparse tree gives the worst performance.

It is possible to combine the properties of sibling lists and hash maps, with less extra space than the sum of the space needed for implementing the individual techniques. If each value in the hash map is expanded with a reference to the next and previous child of the parent, a sibling list is formed. This reference needs only to be the first symbol on the edge to the next child, which does not require much space, given a small alphabet. The node itself should also have a field for the first character referenced. Note that this does not have to be the first character in alphabetical order.

With a linked hash map, you can lookup a child in $\Theta(1)$ expected time, and list all c children in $\Theta(c)$ expected time.

Since we only want to save large nodes explicitly, we would like the small nodes to extract the first character ($firstchar$) in the linked list of their children from the related large node.

3.6.3.1 Insertion and removal in linked hash map

When inserting a node into the child chain, it should be inserted in the second position, so that you neither have to update $firstchar$ field of the parent nor traverse the entire list. This gives $\Theta(1)$ child insertion. The reason you do not want to change the $firstchar$ field, is that this must be valid for all nodes in a node chain if you use small and large nodes. To get $\Theta(1)$ child removal, you need a double linked list. We add a $prevchar$ field to the entries in the hash map. A hash map with sibling lists gives optimal $\Theta(1)$ lookup, insertion and deletion, and $\Theta(c)$ traversal. This is under the assumption that a hash map has $\Theta(1)$ cost of all operations, which is not true in theory. But in practise, this is the amortised cost.

Theorem 3.11: The character on the edge to the first child added to the first node in a small–large chain, is a valid $firstchar$ for all the nodes in the chain.

Proof: Given the nodes $\overline{\alpha\beta}$ and $\overline{\beta}$ are in the same small–large chain, and that the first seen

occurrence of $\alpha\beta$ is at position i in the document T^k. T_i^k is $\alpha\beta\tau$ for some τ. Then $T_{i+|\alpha|}^k$ is $\beta\tau$, and the first character in τ is a valid $firstchar$ for both $\overline{\alpha\beta}$ and $\overline{\beta}$.

Remark 3.12*:* When deleting a child on the same character as given in the $firstchar$ field, this field must be updated. As when updating $docid$ and $hpos$, this value should be chosen from the leftmost element in the small–large chain. Proofs for this can be found in section 3.7.3 later.

3.6.3.2 Split end markers

In a single document suffix tree, a node will only have one child on the unique end marker "$". In MDST we need multiple end markers, as many documents can end with the same substring. If the end markers are treated as characters, the fields $firstchar$, $prevchar$ and $nextchar$ must be large enough to hold a document id. Given a maximum number of documents of 16 million, this enlarges these fields from 1 to 3 bytes.

An elegant solution to this is to split the end markers into strings representing the $docid$s. These do not have to be stored explicitly, but can be generated on the fly. In an implementation, the end marker should preferably always start with the same unique character, typically binary zero, to reduce the number of outgoing edges from nodes when indexing texts which do not use the entire alphabets.

3.6.3.3 Linked hash map space cost summary

Example 3.3*:* Let the maximum field values be those given if figure 3.8. The fields required for each data type is shown in 3.9. A large node would requires 16 bytes. For $childmap$, a hash key 6 bytes, a hash value $\max(7, 8) = 8$ bytes, totalling to 14 bytes. If the space overhead for the hash map is a factor of γ, the total space requirement is $N \cdot 16 + 2N \cdot 14 \cdot \gamma = (16 + 28 \cdot \gamma)N$ bytes.

Variable	Bits	Bytes	Max	Fields
$maxdoc$	23	3	8M	$docid$
$maxlen$	24	3	16M	$depth$, $hpos$
$nodeoff$	32	4	4G	$nodeno$
$maxdist$	7	1	128	$dist$, $backdist$
$maxchar$	8	1	256	$char$, $nextchar$, $prevchar$

Figure 3.8: Example field sizes

3.6.3.4 Hash map implementation

There are two main ways of implementing a hash map. Either you use open hashing or closed hashing. The idea in either, is that for each key, you want to find a position to store or get the

Large	Key	Int. ref	Leaf ref
$docid$ $depth$ $hpos$ $sufflink$ $backdist$ $firstchar$	$dist$ $nodeno$ $char$	$leafbit$ $dist$ $nodeno$ $nextchar$ $prevchar$	$leafbit$ $docid$ $hpos$ $nextchar$ $prevchar$
16	6	7	8

Figure 3.9: Fields with linked hash maps

related value in expected $\Theta(1)$ time. The problem is that you have limited storage space, and you do not know anything in advance about the incoming keys. This means that you probably get collisions between differing keys, hashing to the same position in your storage area.

In open hashing, you solve this problem by maintaining an array of linked list. If two keys hash to the same value, the key–value pairs will be in the same linked list. It is common to let this structure grow until the number of key–value pairs is equal to the size of the array before it is resized. A good hash function hashes keys evenly distributed into the array. There is a space overhead in open hashing related to the pointers in the linked list. If the pointers are memory addresses, the overhead per pair is the size of such an address. On a 32 bit machine, it is 4 bytes, while on a 64 bit machine, it is 8 bytes. This means that the linked hash map parent–child implementation will be much less space efficient on a 64 bit machine if open hashing is used. Open hashing is what is used in Silicon Graphics' freely available implementation of `hash_map`, which is an extension of the C++ Standard Template Library.

In closed hashing, all key–value pairs are put directly into the array. If there is a key collision, a new position is found for the second pair. The new position can be found by increment (linear probing), or by applying the hash function on a combination of the old position and the key. The latter is called double hashing, and spreads the keys better, but has worse cache performance. In closed hashing, you cannot let the array fill to 100% before you resize, as you then would need linear time to locate pairs. It is common to let the fill be at most 50%. You then need $\sum_{i=0}^{\infty} i \cdot 0.5^{i+1} = 1$ jumps on average to locate a key.

Closed hashing has much better space performance than open hashing when the sizes of the keys and values are small. You need approximately twice the space needed for just storing the pairs. If the keys and values are larger than memory addressees, open hashing is the most space effective.

A closed hash map has been implemented and tested. It was more space efficient, but much slower than SGI's `hash_map` on large numbers of pairs. Therefore the standard open hash implementation is used.

3.6.4 Child arrays

The linked hash map has better expected asymptotic behaviour than the sibling lists, but they are rather complex. Looking up an entry requires many operations: Finding the hash value of the key, handling key collisions, and extracting the value. For a low number of children, the sibling lists are faster. Child traversal is rather slow in practise for both these models, as they have bad memory locality. The children traversed in the linked lists are spread throughout memory.

A simpler solution is to store pointers to the children of each node in a separate array, giving good memory locality on traversal. If these arrays have room for σ entries, there is $O(N\sigma)$ space overhead, as there are N or less internal nodes. Instead we can use arrays of varying sizes, are reallocate the child pointers when more space is needed. We apply the common strategy of array doubling. When an internal node is created, it is given space for two children, which we know it will use. If a third child is added, it is given space for four children, and so on. To get good cache performance, the first character of the edge of each child should be stored in the array with the pointers. If we did not do this, we would have to extract the $hpos$ and $docid$ from each child, and then extract $T^{child.docid}[child.hpos + parent.depth + 1]$.

To avoid the time and space overhead of allocating and deallocating arrays, we use the type of container which is used for storing the fields of nodes, explained in section 3.5.2. Child arrays of equal size are stored in one container, so that all elements in this container are of equal size. This means free elements can be kept in a linked list, and space allocated and deallocated in constant time. When more elements are needed, the container space is increased.

A layout for this design is given in figure 3.10.

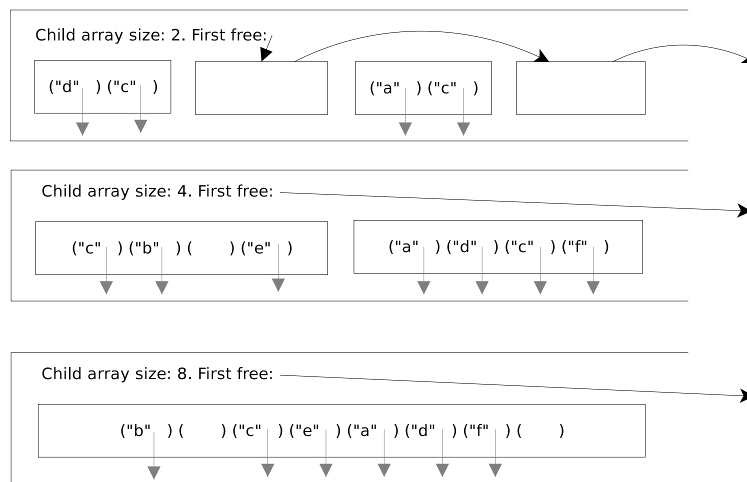


Figure 3.10: Child arrays.

Theorem 3.13: Child arrays using doubling has a space overhead of $6N$ times the size of a node reference, as long as no nodes are deleted. Space logarithmic on the size of the alphabet is needed for managing the containers.

Proof: Each container is at least half full, and each child array in each container is at least half full, so at least one fourth of the space is used for children. There are at most $2N$ children, which means we an overhead of $(4-1)2N = 6N$ pointers. For each container, you need to store its size, and a pointer to the first free element in the free list. When we use doubling, there are $\log_2 \sigma$ child array containers.

Note that as with the linked hash map, there is no need to store leaves explicitly. They are uniquely identified by the document ID and head position stored in their identifier. But small nodes need more than zero space, as they need a pointer to their child array. Small nodes has not been implemented for this model.

3.6.4.1 Unsorted child arrays

The entries in a child array can be stored in a random order. To find space for an entry, locate the child on a given character, or delete an element, you traverse the array until you find the right slot, using $O(c)$ amortised time, where c is the number of children for a given node. The array has less than twice as many slots as the node has children. If reading memory is cheaper than writing, a random order is favourable, as you only have one write for each insertion/deletion, except when reallocating.

3.6.4.2 Sorted child arrays

Another strategy is to let the children in an array be sorted on the first character on the edge going to them. Then you could do child lookup in $O(\log c)$ time, where c is the number of children. Traversal, insertion and deletion would be $O(c)$. When inserting or deleting, you would need to shift a portion of the array left or right. If you kept a counter on the number of children in each array, you would get a constant factor speedup, at the cost of the space for the counter. Remember that the child arrays are half full in the worst case.

In practise, binary search is only faster than linear traversal if the number of children is above a certain limit, as more arithmetic operations are needed in each step. For small alphabets, sorting the arrays does not pay off. Usually, only nodes near the root have a large number of children. One possibility would be to only use sorted arrays for such nodes, and unsorted arrays for the rest.

If you have a very small alphabet, for example DNA, the best solution might be to fix the child array size to σ. You would then get direct lookup, insertion, and deletion, as all symbols would have a predefined position.

3.6.4.3 Deleting from child arrays

When removing an entry from a child array, the number of elements left should be counted. If the fill is below a certain level, typically 25%, the array should be reallocated. Child operations still take linear amortised time, as with regular array doubling.

You get a more difficult problem if the number of child arrays of a given size decreases over time. Internal nodes have pointers into this container, which must be updated if the child arrays are moved around. The problem is the same both when the pointers are internal indexes and direct memory addresses. You would need to scan $O(N)$ internal nodes for references, a high price to pay. In a test with 3000 document insertions and 2000 document deletions run in a mixed order, the number of child arrays did not decrease over time for any size. Shrinking child array containers has not been implemented.

3.6.4.4 Doubling factor

When using array doubling on a regular array, the worst case amortised work done on n insertions is

$$\sum_{i=0} \frac{n}{d^i} \leq \frac{d}{d-1} n$$

where d is the doubling factor. We can tune the space and time performance by adjusting this parameter. If we set $d = 1.5$, we get at most 50% space overhead in each child array container. Reallocating memory is a cheap operation. The `realloc` system call is used, which only moves data if it cannot get the adjacent memory area. Compared to the rest of the suffix tree construction, it is fast even when data is moved. In the tests in chapter 5, a doubling factor of 1.1 was used, at the cost of about 10% time cost increase on document inclusions over a doubling factor of 2.

3.6.5 Parent–child implementation summary

The table in figure 3.11 gives a summary of the properties of the various node models. σ is the size of the alphabet, and c is the number of children for a given node.

3.7 Document ID update model

In a multi-document dynamic suffix tree, an internal node can have multiple children with a $docid$ different from itself. Therefore, after you have deleted a document from a MDST, there

Implementation	**Lookup**	**Insertion**	**Deletion**	**Traversal**
Sibling lists	$O(c)$	$\Theta(1)$	$O(c)$	$\Theta(c)$
Sorted sibling lists	$O(c)$	$O(c)$	$O(c)$	$\Theta(c)$
Hash map	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\sigma)$
Linked hash map	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(c)$
Unsorted array	$O(c)$	$O(c)$	$O(c)$	$\Theta(c)$
Sorted array	$O(\log c)$	$O(c)$	$O(c)$	$\Theta(c)$
Full array	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\sigma)$

Figure 3.11: Summary of parent–child models

might be internal nodes left with a $docid$ referring to the deleted document. Because you traverse the tree following suffix links, you do not necessarily see all related internal nodes. These nodes must be updated, if you do not want to keep a copy of the document.

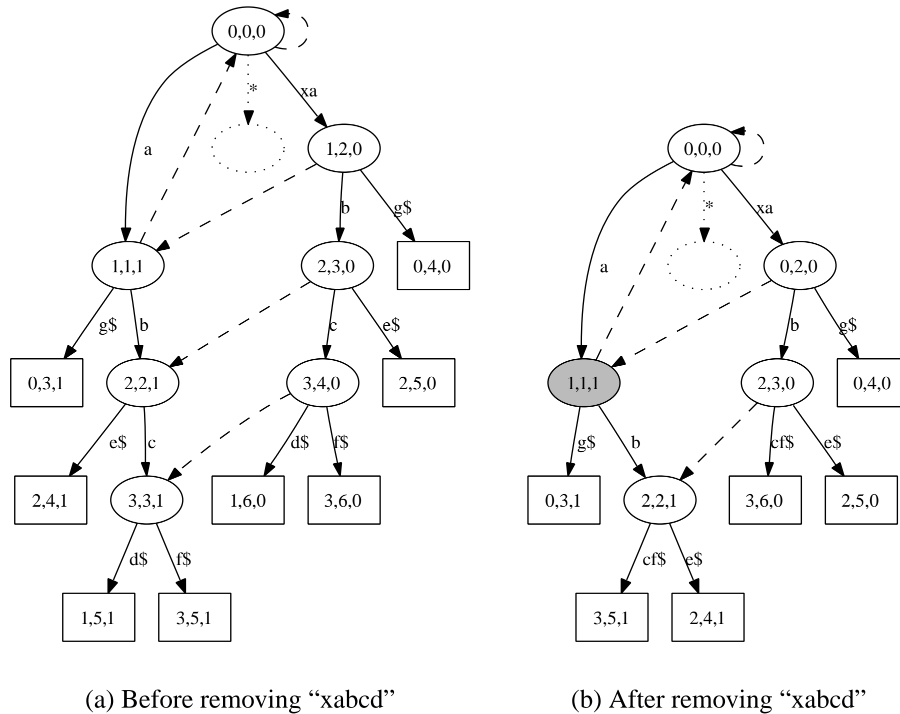
Remark 3.14*:* A suffix tree representing multiple strings may contain a node $\overline{\alpha}$, even when none of the suffix trees representing the individual strings contain a node $\overline{\alpha}$. Proof by construction in example 3.4.

Example 3.4*:* Consider the MDST for the strings “xag”, “xabcd”, “xabe” and “xabcf” shown in figure 3.12(a). The numbers inside the nodes are $docid$, $depth$ and $hpos$. Notice that none of the suffix trees for the individual strings have an explicit node $\overline{\text{a}}$, as “a” is always followed by the same character (it occurs only once in each string). The first string will have a node $\overline{ag\$}$, the second a node $\overline{abcd\$}$, and so on. But in the MDST, there will be a node $\overline{\text{a}}$, after you have added “xag$” and “xabcd$”, which between them have both ‘g” and “b” after “a”. This node is naturally described by the tuple $(1, 1, 1)$, as the scan for $T_1^1 =$ “abcd” will halt between $root$ and $\overline{\text{ag}}$.

Remark 3.15*:* When removing a document from an MDST with `deldoc` from figure 3.2, there may be a node referencing the document ID, which is not visited. Proof by construction in example 3.5.

Example 3.5*:* We remove the string “xabcd” from the MDST in example 3.4. Downward traversal takes us via $\overline{xa}$, $\overline{xab}$ and $\overline{xabc}$ to the leaf $\overline{xabcd}$, which is deleted, along with the parent $\overline{xabc}$. Then we follow the suffix link from grandparent $\overline{xab}$ to $\overline{ab}$. Downward traversal takes us via $\overline{abc}$ to the leaf $\overline{abcd}$, which is deleted. Notice that the node $\overline{a}$ was never visited. The following iterations will not touch the subtree rooted at $\overline{a}$, as neither “bcd”, “cd” nor “d” contain “a”.

The following sections describe two methods for solving this problem.



(a) Before removing "xabcd"

(b) After removing "xabcd"

Figure 3.12: MDST for "xag", "xabcd", "xabe" and "xabcf"

3.7.1 Eager bottom-up document ID clean-up

After the removing a document T_k from an MDST with `deldoc` the tree may be left with internal nodes referring to that document, even if `deldoc` is modified to update nodes seen during traversal. It may not be affordable to keep a copy of all deleted documents. One possibility is doing a top-down traversal of the entire tree, but the cost would be $\Theta(N)$, linear on the size of *all* documents.

A better solution is doing a bottom up traversal from the nodes seen in `deldoc`, updating all nodes with invalid $docid$. If done naively, this would cost $O(n_k^2)$, as you may walk the same path multiple times. If you traverse each path at most once, the cost will be $\Theta(n_k)$.

Remark 3.16*:* A bottom up traversal of the suffix tree requires parent pointers, which will give a linear extra total space cost.

When a node with an invalid $docid$ is seen, a new $docid$ and $hpos$ must be inherited from one of the children, possibly recursively. Python code for this is given in figure 3.13.

Lemma 3.17: If there is a node having a $docid$ referring to a deleted document, there is a node in the subtree below it with different $docid$.

Proof: By theorem 3.6, after deleting a document with `deldoc`, all leaves referring to the given document are deleted. So any internal node must have a descendant with a $docid$ referring to

```
def heritdocid(self, node):
    oldid = node.docid
    child = node.children[0]
    newid = child.docid
    if isremoved(newid):
        heritdocid(child)
        newid = child.docid
    node.docid = newid
    node.hpos = child.hpos
```

Figure 3.13: Inheriting $docid$ and $hpos$ from children.

another document.

Theorem 3.18: After removing the document T_k, a bottom up traversal running `heritdocid` on all internal nodes with $docid = k$ takes $O(n_k)$ time.

Proof: Each call to `heritdocid` takes $\Theta(1)$ time, excluding the recursive step. `heritdocid` is called on a total of $O(n_k)$ nodes, as there are at most n_k internal nodes with $docid = k$ because `build` creates more leaf nodes than internal nodes. No node is visited more than once by `heritdocid`, as all nodes visited have have $docid = k$ before the visit, and $docid \neq k$ after the visit. The upward traversal takes $O(n_k)$ time if nodes are marked when first seen, and seen nodes are not traversed again.

3.7.2 Lazy document ID clean-up

A method of updating document IDs that does not require parent pointers, is doing a lazy update on nodes when an invalid document ID is seen when using the MDST. In the bottom-up traversal, only internal nodes need parent pointers, as all internal nodes with leaves from the deleted document are always visited in the traversal. So the amount of space you save is the size of a pointer times the number of internal nodes.

The cost of this solution is having to check if the document ID is valid each time you acquire a node by following a suffix link, or retrieving a child node. When a node with an invalid document ID is found, you must call `heritdocid` on the node.

Theorem 3.19: The total work of lazy update of nodes is worst case linear on the total length of all documents removed, R.

Proof: From the proof of theorem 3.18, we know that the number of invalid $docid$s left after removing T_k is $O(n_k)$. So after removing documents of total length R, $O(R)$ invalid $docid$s are left in the tree. From theorem 3.18 we know that a constant amount of work is done in `heritdocid` for each invalid $docid$ seen. Hence, the total work is $O(R)$.

Remark 3.20*:* Given lazy update, lookup of a single pattern P of length m might take longer than $O(m)$ time. That would happen if it hits a long column of nodes with invalid $docid$s. The probability of this is low in practise.

Timings and memory usage for lazy and eager update is found in chapter 5.

3.7.3 Updating document IDs in small–large chains

When a document is removed from an MDST, some internal nodes with $docid$ referring to the given document may be left in the tree, as described in section 3.7. When updating a node belonging to a small–large chain, you must reset the $docid$ and $hpos$ of the large node. But, you have to make sure that these values are representative for *all* the nodes in the chain.

Theorem 3.21: A $docid$ and $hpos$ valid for first node in a small–large chain will always be valid for all the nodes in the chain when they are translated by the distance to the leftmost node.

Proof: If a $node_a$ and $node_b$ belong to the same small–large chain, and $node_a$ is created before $node_b$, it means $node_a = \overline{\alpha\beta}$ and $node_b = \overline{\beta}$, for some α and β. We know that $\alpha\beta$ must be a prefix of a suffix T_i^k of an unremoved document T^k, since $node_a$ is not deleted. Let $\alpha\beta\tau = T_i^k$, where T^k is an undeleted document. Chose $docid_a$ and $hpos_a$ from the child of $node_a$ on the first symbol in τ. $T_{i+|\alpha|}^k = \beta\tau$, so β is a prefix of $T_{i+|\alpha|}^k$. Hence $docid_b = k = docid_a$ and $hpos_b = i + |\alpha| = i + node_a.dist - node_b.dist = hpos_a + node_a.dist - node_b.dist$ is valid for $node_b$.

Remark 3.22*:* To find the leftmost node in a chain, the large node must have a $backdist$ field, which is equal to the $dist$ of the leftmost node.

Chapter 4

Hierarchic index implementation

In section 2.4, two strategies for hierarchic indexes were discussed, called method 1 and method 2, both having a parameter k. This chapter discusses the implementation of these two methods, and gives two tricks for speeding up search: A document ID map, and a start index for the binary search.

A difference from the theoretical description earlier, is that we do not consider documents to have unit size. Sizes of sets of documents and indexes are rounded up to the nearest valid size. Notice that numbers are added before they are rounded. An index using method 1 containing two documents of size $k^{10}/2$ will be considered to have size k^{10}, not $2k^{10}$.

4.1 Method 1

In method 1, there are at most $k-1$ indexes of each size k^i. The indexes can be maintained on a stack, where smaller indexes are closer to the top. When a new document is put onto the stack, we leave it as an unprocessed index. Then, as long as there are k indexes of equal size on top of the stack, we pop them off the stack and push their content as an unprocessed index onto the stack. Then we process the unprocessed index on top of the stack.

Example 4.1*:* We want to add a document to the indexes shown in figure 2.12 on page 25, where the stack grows from left to right, and $k = 3$. We push the new document onto the stack as an unprocessed index. Then, there are 3 indexes of size 1 on top of the stack, and these are merged. Then, as there are now 3 indexes of size 3, these are merged. There are now 2 indexes of size 9. At last, the topmost index is processed.

If the documents are not considered to have unit size, an alteration of the algorithm must be done. When a document is put onto the stack as an unprocessed index, it is merged with the index below it as long as that is considered smaller. Then the normal procedure is followed. Interpreted sizes should be rounded up to the nearest power of k. This is shown in figure 4.1.

```
from math import *

def reprsize(k, realsize):
    return ceil(log(realsize) / log(k))

def add1(k, stack, document):
    stack.append(document)
    while (True):
        while len(stack) > 2 and reprsize(k, stack[-1]) > reprsize(k, stack[-2]):
            stack.push(stack.pop() + stack.pop())
        if len(stack) < k:
            break
        merge = True
        for i in xrange(len(stack) - 1, len(stack) - k, -1):
            if reprsize(k, stack[i]) != reprsize(k, stack[i-1]):
                merge = False
                break
        if merge:
            content = 0
            for i in xrange(k):
                content += stack.pop()
            stack.append(content)
        else:
            break
    # return amount of work done and the number of indexes in use
    return stack[-1], len(stack)
```

Figure 4.1: Pseudocode for method 1

4.2 Method 2

In method 2, there are at most one index of each "size type", $b_i k^i \mid 0 \leq b_i < k, 0 \leq i \leq \lfloor \log_k n \rfloor$. When a document is added, the smallest index which can contain the new document, itself, and *all smaller* indexes is rebuilt rebuilt to contain all of this.

Example 4.2*:* A set of indexes with $k = 3$ are shown in figure 2.13 on page 26. They have fill 1/2, 6/6, 9/18, and 0/54, which means that $b_0 = 1$, $b_1 = 2$, $b_2 = 1$ and $b_3 = 0$ (using the notation of chapter 2). Notice that index i has a maximum capacity of $k^i(k-1)$. If we want to add one document to the index, b_0 is increased to 2. The total number of documents is now 17. When we want to add another document, we find that the first $b_j < k - 1$ is $b_2 = 1$. We set b_0 and b_1 to 0, and increase b_2 to 2, meaning that we move the new document, and the contents of index 0 and 1, into index 2. The fill of the indexes is now 0/2, 0/6, 18/18, and 0/54.

Code for method 2 is shown in figure 4.1. When a maximum fill is given for each index, you do not have to calculate the nearest power of k.

4.3 Comparing method 1 and method 2

If $P_S(n)$ and $Q_S(n)$ are the costs of indexing and querying of static document sets, we have the following for dynamic sets with method 1 and 2:

```
def add2(k, fill, limit, document):
    content = document
    j = 0
    while True:
        content += fill[j]
        fill[j] = 0
        if content < limit[j]:
            break
        j += 1
    fill[j] = content
    numind = 0
    for i in xrange(len(fill)):
        if fill[i] > 0:
            numind += 1
    # return amount of work done and the number of indexes in use
    return content, numind
```

Figure 4.2: Pseudocode for method 2

$$\begin{aligned}
P_D^1(n) &= O(\log_k n)P_S(n) \\
Q_D^1(n) &= O(k \log_k n)Q_S(n) \\
P_D^2(n) &= O(k \log_k n)P_S(n) \\
Q_D^2(n) &= O(\log_k n)Q_S(n)
\end{aligned}$$

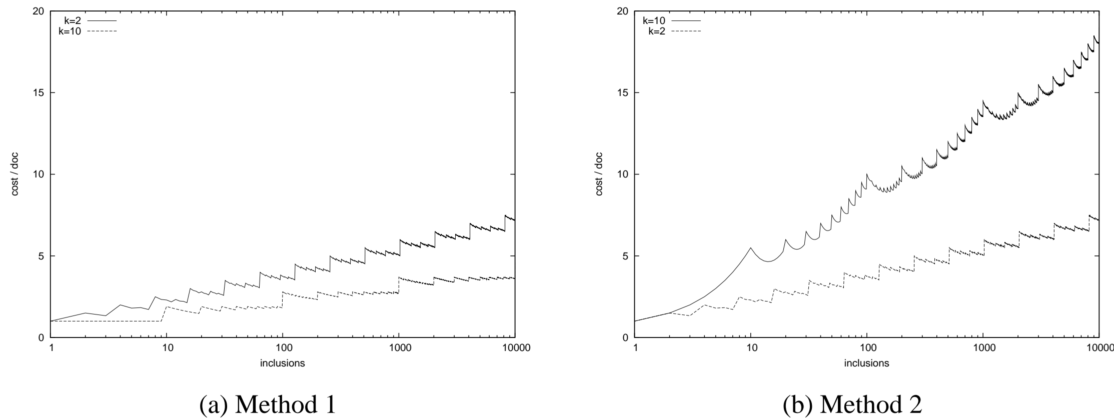
With method 1, you can get the indexing cost arbitrarily close to $P_S(n)$ by increasing k, at the price of getting a very high query time. With method 2, you get a query time close to $Q_S(n)$, at the price of a very high indexing cost.

Below in figures 4.3(a) and 4.3(b) you see the average work per document plotted against the number of documents added. It is assumed that documents have unit size, and that $P_S(n) = n$. Figures 4.4(a) and 4.4(b) shows the maximum number of indexes during the runs, which is equal to the upper limit of $Q_D(n)$ if $Q_S(n) = 1$.

See how both methods are equal for $k = 2$. Real timings for method 1 and 2 used with BPR can be found in chapter 5.

4.4 Suffix arrays for multiple documents

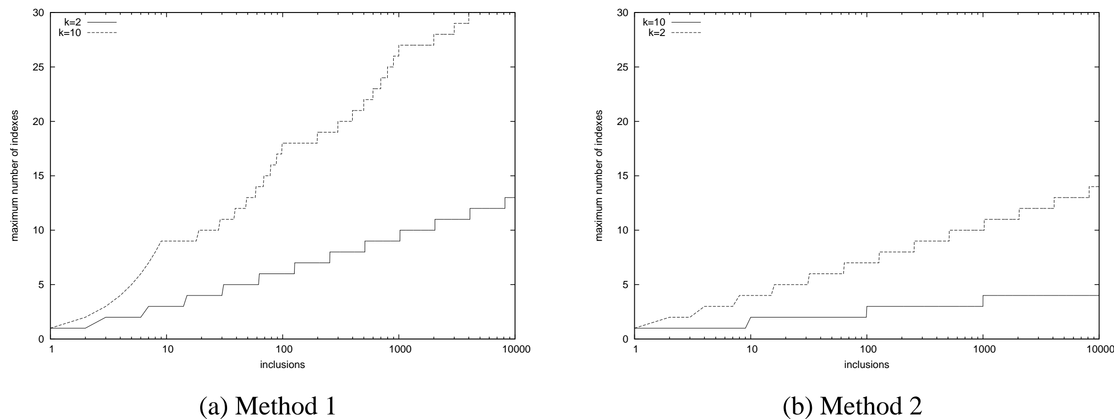
When you index multiple documents with a suffix array, you concatenate them, and build the suffix array for the resulting string. You can either separate the documents with a unique symbol, or check that hits are within document boundaries when you list them. It is much more efficient than having multiple suffix arrays, because of the nature of the binary search. If you have documents of sizes $n_0, n_1, \ldots, n_d$, greater than 1, you know that



(a) Method 1

(b) Method 2

Figure 4.3: Work per document



(a) Method 1

(b) Method 2

Figure 4.4: Maximum number of indexes

$$\log(n_0 + n_1 + \cdots + n_d) \ll \log n_0 + \log n_1 + \cdots + \log n_d$$

When you search for a pattern, you get a set of positions in the global string, which you want to map into local positions in documents. You can save the starting positions of the documents in an array, with a space cost of $\Theta(d)$. To map a hit, you perform a binary search in the starting positions. When you have found the right document, you subtract its starting position from the global position. With this approach, finding occ occurrences of a pattern takes $O(m + \log n + occ \log d)$ time.

4.5 Document ID map

To avoid the binary search for the right document, you could save the document ID for every position in the global string in an array. But this is not very space efficient. It almost doubles

the space requirements of the suffix array.

Instead, you can save the document ID for every r-th position for some factor r, and pad all documents to a multiple of r. When you want to map the global position i, you find the document ID in position $\lfloor i/r \rfloor$.

As long as r is much less than the average document length, there is very little space overhead. r does not need to be very large. If $r = 8$ and document IDs take 4 bytes, you need $\frac{1}{2}N$ bytes extra for the map.

4.6 Start index for binary search

When you perform many binary searches in a suffix array, you often do steps which were identical in previous searches. Given an even character distribution in the string, all searches starting with the same character share the first couple of steps.

To avoid repeatedly doing to much of the same work, you could make an index of prefixes of length s of the suffixes of the string, giving the binary search left and right border for each prefix. As the length of the prefixes increases, the number of entries in the index grows towards N, giving a high space overhead, so you should keep s small. In practise, you should *not* use a hash map for the index, as the overhead of lookup in the hash map is probably greater than the cost of performing the extra steps in the binary search. This is assuming that you have a RAM model machine. If the suffix array resides on disk and the start index in memory, it would not be true.

A good solution in practise is to have an array indexing all possible substrings of length s from σ. For any string from Σ^s found in the text, the index should give the position of the leftmost and rightmost occurrence in the suffix array. The space cost for this is $2\sigma^s$ times the size of a pointer. A good choice for s is 2, having a space requirement of $2 \cdot 256^2 \cdot 4\text{B} = 2^{19}\text{B} = 512\text{KB}$ with an alphabet of size 256 and pointers of 4 bytes. For $s = 3$, the space requirement is 128MB.

If you have an even character distribution, you would on average save $\log_2 n - \log_2 \frac{n}{\sigma^s} = \log_2 n - \log_2 n + \log_2 \sigma^s = s \log_2 \sigma$ steps. If n is sufficiently large, the save is independent of n, meaning the gain is relatively smaller the more data you have.

Note that when searching for longer strings, or strings with many hits, the search time is dominated by the m or the occ term. Then using the start index does not give much advantage.

Chapter 5

Results

5.1 Test data

To be able to test various dimensions of the problem complexity, most of the tests are run on computer generated test data. Some of the data contains just random words in a given length range, from a given alphabet, while some data contains words from a dictionary, to simulate real-world data.

A dictionary of a given number of words is created, and words are selected from this dictionary using a zipf distribution. The probability for seeing a word is inversely proportional to the rank of the word. For the rth word:

$$p(r) = \frac{1}{r} \cdot \frac{1}{\sum_{i=1}^{R} \frac{1}{i}}$$

where R is the total number of words in the dictionary. Words for the zipf dictionary were generated with lengths from the length distribution found in `bible.txt` from the Canterbury Corpus [BP]. Random words were generated with random lengths uniformly distributed between an upper and a lower limit. The characters in the words in the zipf dictionaries were selected with a first order Markov chain, using a zipf distribution.

A test contains a set of requests to an index: Include a document, remove a document, or find matches for a query. Document update is simulated with removal and addition. Finding what has changed in a document, and updating the indexes accordingly, usually has a higher cost than just flushing the old version and inserting the new. The queries are randomly selected phrases from the indexed documents, of a minimum and maximum length. For some tests, a maximum is given for the number of reported hits. There is also a freshness requirement for the index. This gives how many new documents can be requested for addition before they are actually added. This is included because some of the methods like to batch updates, and are greatly affected by this parameter.

The parameters for the tests in this chapter are given i figure 5.1. When “inclusions”, “removals” and “queries” are given in percentages, such requests are run in a random order. When they are given as integer numbers, the requests of each type are run separately, in the listed order.

	alphasize	datasize	docsize	freshness	general	general2	numdocs	random	reporting
inclusions	30%	1	40960-20	10000	30%	30%	1-16384	30%	30%/17%
removals	20%	0	0	0	20%	20%	0	20%	20%/0%
queries	50%	10000	0	0	50%	50%	1000	50%	50%/83%
num. requests	2000	-	-	-	10000	30000	-	4000	10000/6000
file size	4KB	1KB-64MB	1KB-2MB	4KB	4KB	4KB	2MB-128B	4KB	4KB
alphabet size	4-128	26	26	26	26	26	26	26	26
random words	100%	0%	0%	0%	0%	0%	0%	0-100%	0%
zipf dictionary	0%	100%	100%	100%	100%	100%	100%	100-0%	100%
zipf dict. size	-	10000	10000	100000	100000	100000	10000	100000	100000
min. word	-	1	1	1	1	1	1	1	1
max. word	-	20	20	20	20	20	20	20	20
freshness req.	0	0	0	0-256	0	0	0	0	0
min. query	30	5	-	-	30	5	3	30	3
max. query	30	5	-	-	30	5	3	30	3
max hits	1	1	-	-	1	∞	1000	1	1-1000

Figure 5.1: Test parameters

5.2 Test system

All the tests were run on an 2.2 GHz AMD Athlon 64 3500+, giving 4374 bogomips. It has 64+64 KB level 1 cache, and 512 KB level 2 cache. 3.6 GB of RAM was available. The system was running Debian for AMD 64, and Linux version 2.6.10. All software was compiled with GCC version 3.4.4, using the options `-O3 -finline-functions -fno-rtti -fno-exceptions`.

5.3 Comparison of node models for MDST

Many different underlying node models for the MDST were tested. They differ in how nodes are stored, how the parent-child relationship is implemented, and how *docid* update is done for internal nodes after document removal. See chapter 3 for detailed descriptions.

Nodes are either stored using the `struct` concept in the C programming language, or in a compact byte-addressed array. The former has a space and time overhead during node creation/deletion. The latter gives the possibility for data fields of any number of bytes. Many different ways of implementing the parent–child relationship of nodes are possible. We test sibling lists, linked hash maps and child arrays. After a document has been removed, it may leave internal nodes with its now invalid *docid*. This can be fixed with a eager or lazy update scheme.

The following implementations were tested:

- compact node, sibling lists, eager update (MDST comp.sibl.e)
- compact node, sibling lists, lazy update (MDST comp.sibl.l)
- compact node, linked hash map, eager update (MDST comp.lhm.e)
- compact node, linked hash map, lazy update (MDST comp.lhm.l)
- large/small compact node, linked hash map, lazy update (MDST comp.lhm.s)
- compact node, child array, eager update (MDST comp.arr.e)
- compact node, child array, lazy update (MDST comp.arr.l)

Originally, `struct` nodes were included in some of the tests, but they performed worse than the compact nodes in all manners. Inclusion, removal and query time was higher, and they used more memory. They are slower because of worse memory locality and space allocation costs. They are excluded from the test plots to make the differences between the rest of the models clearer.

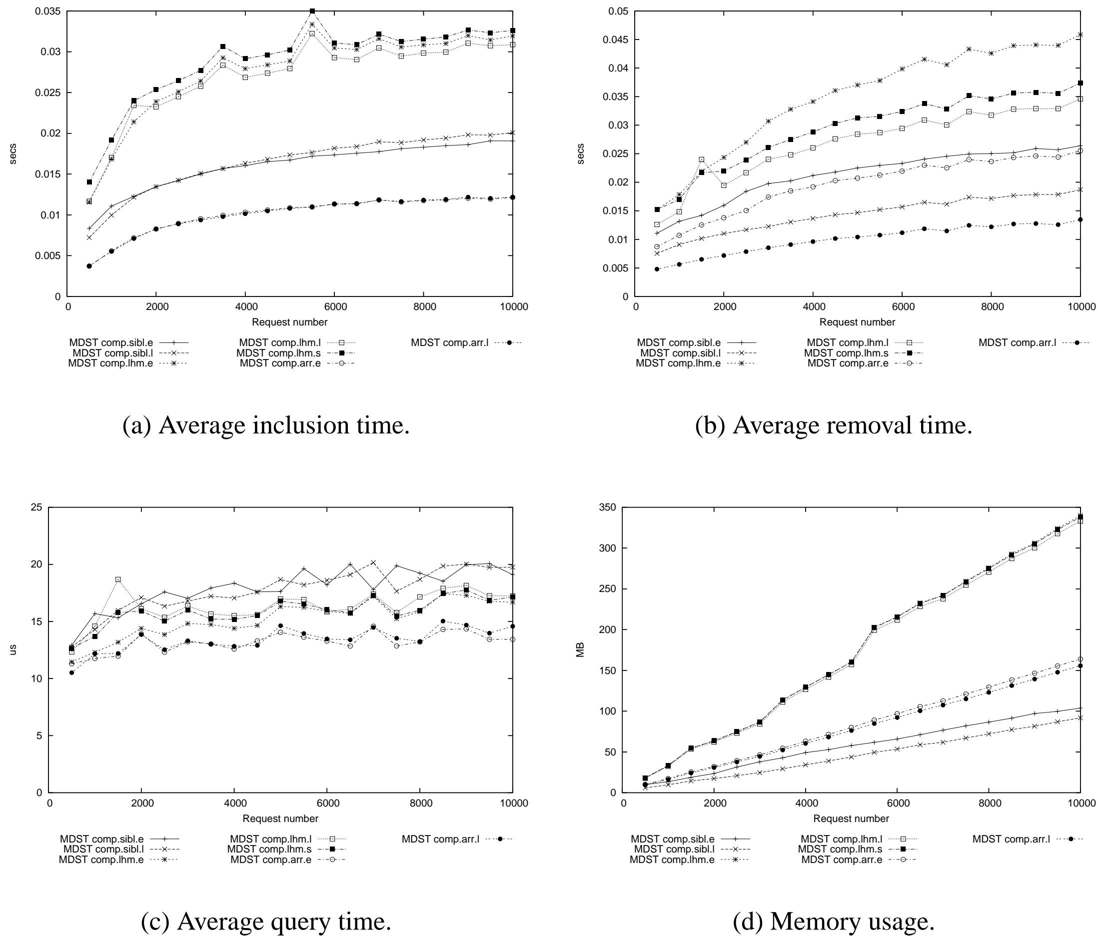
5.3.1 General test on zipf data

The first test is a general test on document inclusions, removals and queries. The parameters from the column "general" in figure 5.1 are used. You see the results in figure 5.2. Requests of different types are given in a random order, and averages, minimums and maximums are given for each time interval in the graphs. Since there are more inclusions than removals, the amount of data increases over time. Only one hit is reported for each query, to make sure the cost is equal for all queries. A test of hit reporting is included in section 5.3.4.

In general, inclusion time increases over time. For all methods, the cost of memory reallocation increases as the size of the data increases. For the sibling lists and the child arrays, the inclusion time also increases because the out-degree of the nodes increases as the amount of data increases, and there is a $O(\sigma)$ factor in finding the child of a node. For the linked hash map, the inclusion time varies more. This is probably because of the cost of reallocating space for the child map. When a hash map is resized, all elements must be re-entered.

On document inclusion, child arrays (MDST *.arr.*) are faster than sibling lists (MDST *.sibl.*), which again are faster than the linked hash maps (MDST *.lhm.*), even though the former two have a linear dependency on the size of the alphabet. This is due to the greater overhead in maintaining the linked hash map. (See section 5.3.3 for a test of varying alphabet sizes.) Using small nodes (MDST comp.lhm.s) gives a small performance penalty on inclusion, but almost no space advantage. This is because most of the space is used for the parent-child map, which is very memory inefficient on the used 64-bit architecture.

Lazy update (MDST *.*.l) gives much faster document removal than eager update (MDST *.*.e). This is because in the latter, we do a bottom up traversal of the entire tree after removing



(a) Average inclusion time.

(b) Average removal time.

(c) Average query time.

(d) Memory usage.

Figure 5.2: Using zipf data. Values sampled during run.

the superfluous nodes. Lazy update only gives a minor performance penalty on inclusion and queries.

Removal time increases the most over time for the eager update. Even though it should be linear on the size of the document, there will on average be more internal nodes above the leaf nodes of a document the bigger the tree is. As the tree fills up, the number of internal nodes above the leaves of a document grows. When removing a document, there is also a downwards traversal, and this becomes more expensive with sibling lists and child arrays as the tree fills up.

On queries, child arrays are fastest, linked hash maps second, and sibling lists slowest. Child arrays have the same asymptotic performance as sibling lists, but have much better cache locality. The linked hash map has optimal expected theoretical performance, but worse memory locality and more expensive operations make them slower than the child arrays. In this test, only one hit was reported. You will see in a test in section 5.3.4 that the sibling lists are more favourable than linked hash maps when many hits are reported.

From this test, we conclude that the implementation using compact nodes, child arrays and lazy update gives the best overall performance. In later tests we drop the small node implementation,

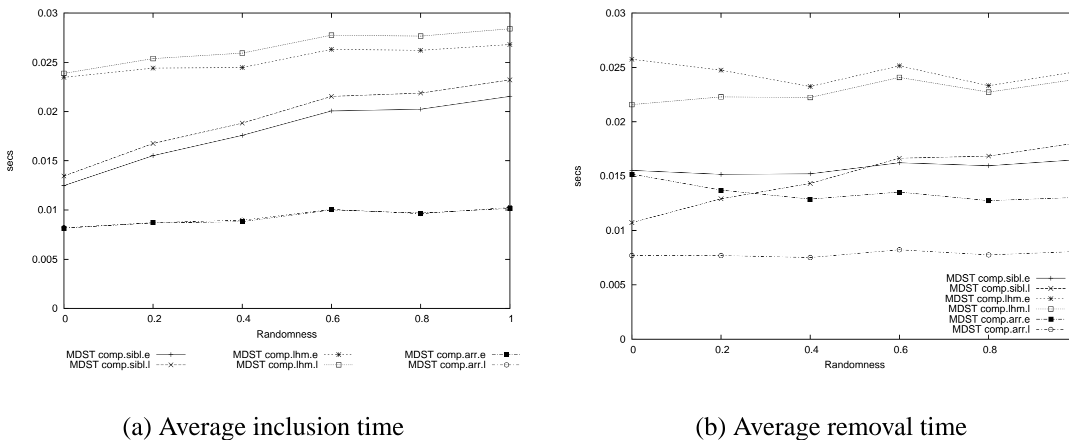
as it did not give much space gain.

5.3.2 Varying text randomness

The data in the former test was generated from words in a zipf distribution. Below follows a test of how the models react on varying randomness in the data. The parameters are given in the column “random” in figure 5.1. A portion of the words in the documents are taken from a zipf distribution, while some are random words.

Figure 5.3(a) shows that the sibling lists perform worse on document inclusion with more random data. This is probably due to the cost of child traversal. On random data the average out-degree of nodes at the top of the tree is higher. The cost of child traversal with child arrays depends less on the number of children in practise, because of caching effects.

We see that the difference between lazy and eager update on document inclusion is visible for the sibling lists, but not for the child arrays. This is because of a small implementation detail. With the sibling lists, the first character on child edges are not stored in the children themselves. To extract this character, the document ID of the child must be checked, and if it refers to an undeleted document, a new document ID and head position must be inherited from a node in the subtree below. In the child arrays, the first character on the outgoing edges are stored within the array. It is not certain why inclusion time differs for eager and lazy update with the linked hash map. It might have to do with with more expensive child access on updates.



(a) Average inclusion time

(b) Average removal time

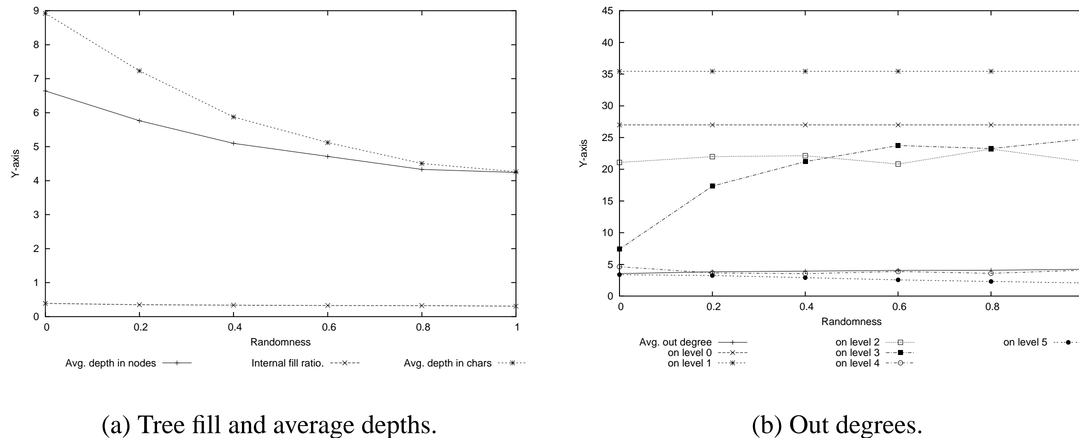
Figure 5.3: Varying text randomness.

In figure 5.4 you see some statistics for the suffix trees in the given run: The average out degree of internal nodes, the average depth of internal nodes measured in symbols, the average depth of internal nodes measured in nodes, and the fill in the tree. The latter is a percentage of the theoretical maximum number of internal nodes. You see that as the randomness increases, the average depth in chars approaches the average depth in nodes, which is expected.

The out degree of nodes is also measured separately in levels of the tree. The level is the distance

from the root measured in nodes. The reason there are more nodes on level 1 than on level 0 is the split document end markers, explained in section 3.6.3.2. An end marker always starts with a unique character, binary zero, but might contain any binary character. You see that the tree is almost always full on level 0, 1 and 2. On level 3, the out degree increases with the randomness. On level 4, the opposite happens. The average depth decreases, and the splits move up in the tree.

The internal fill ratio in figure 5.4(a) is inversely proportional to the average out degree in figure 5.4(b). It seems to be almost constant. Characters in a random string play the role of words in strings from a dictionary. The reason the internal fill is not entirely constant, but seem to decrease slightly with increasing randomness, is that the words in the zipf distribution were created with a first order Markov model. Which character follows another is not random. This means nodes will have fewer children, and that more nodes are needed to make the N paths to leaf nodes.



(a) Tree fill and average depths.

(b) Out degrees.

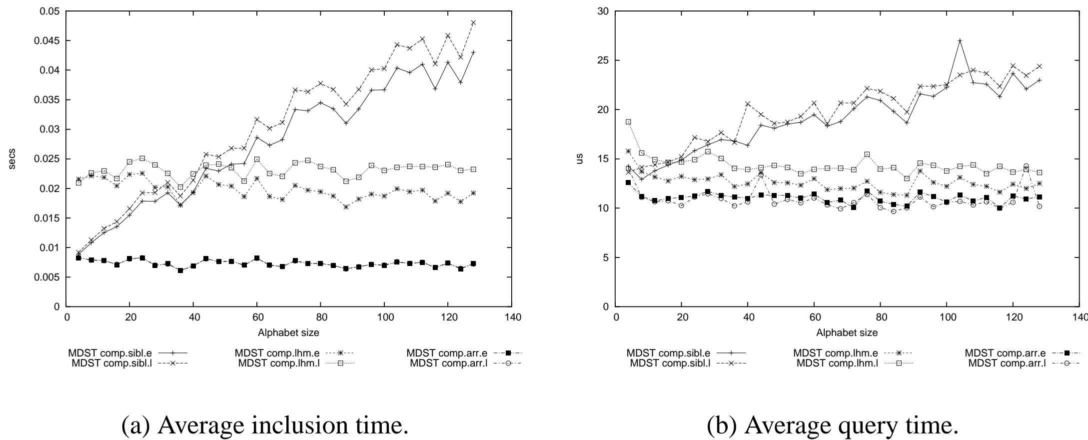
Figure 5.4: Statistics for varying text randomness.

When running a test with the 32nd Fibonacci string, which is about $3.4 \cdot 10^6$ characters long, the internal fill was measured to exactly 1. The average depth in characters was $9.3 \cdot 10^5$, and the average depth in nodes 30.1.

5.3.3 Varying alphabet size

This test measures how the alphabet size impacts the performance of the MDST variants. The parameters for the test are given in the column "alphasize" in figure 5.1. Random data is used. In theory, the sibling list model should have a linear time dependency on the alphabet size, and the linked hash map model should be independent of it.

As you can see in figures 5.5(a) and 5.5(b), the dependency of the alphabet size is as expected for the sibling lists and the linked hash map. The child arrays seem independent of the alphabet size. This is due to cache effects.



(a) Average inclusion time.

(b) Average query time.

Figure 5.5: Varying alphabet size, random data.

Remember that this is an artificial test. Since we have random data, the average LCP is very short, and the nodes at the top of the tree have a very high out-degree. For all models, the query cost goes down from $\sigma = 4$ to $\sigma = 20$. This is because with a smaller alphabet, the average LCP is higher, and the number of nodes you must traverse to find a string is higher. Traversing many nodes is more expensive in practise than traversing long edges. This test runs long queries, reporting only one hit. If more hits were reported, sibling lists would have performed better.

5.3.4 Varying number of hits

Below in figure 5.6 follows a test for reporting a variable number of hits. As it should be, the cost per hit reported seem to be linear above a certain number, for all models tested. The child arrays and the sibling lists are much simpler, and the traversal of the children of a node requires fewer operations. Hence they are faster than the linked hash maps.

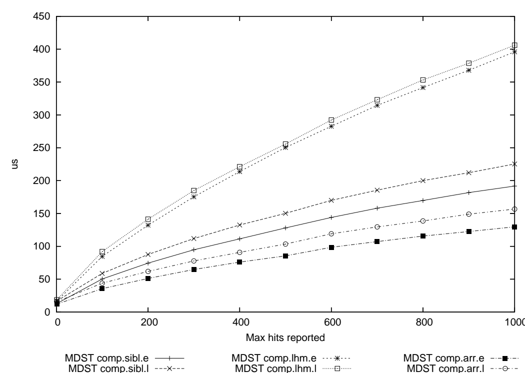


Figure 5.6: Average query time, varying number of hits.

5.3.5 Real world data

Below in figures 5.7, 5.8 and 5.9, follow the results for a test on some documents from the Canterbury Corpus [BP]: `E.coli` (4.5 MB), `bible.txt` (3.9MB) and `world192.txt` (2.4 MB). We test the inclusion of a single document, and running 1000 queries of length 10 to 30, reporting all hits. For reference, Stefan Kurtz' suffix tree implementation from the MUMmer software package [MUM] is included. It is described in [Kur99]. The BPR suffix array construction algorithm [SS05] is also included, where a simple binary search with no additional data structures is used.

	E.coli	bible.txt	world192.txt
MDST struct.sibl.e	5.916	5.347	3.125
MDST struct.sibl.l	5.299	4.581	2.548
MDST comp.sibl.e	5.886	5.055	2.894
MDST comp.sibl.l	6.532	5.631	3.295
MDST comp.lhm.e	13.515	9.109	5.470
MDST comp.lhm.l	12.826	9.127	5.516
MDST comp.lhm.s	34.473	17.284	9.959
MDST comp.arr.e	3.958	2.560	1.345
MDST comp.arr.l	3.927	2.570	1.377
Kurtz	3.897	3.288	1.743
BPR hier m1 k2 d0 s0	1.330	1.352	0.748

Figure 5.7: Document inclusion time in seconds

The sibling list variants have roughly the same performance on document inclusion. Notice that the linked hash map variants perform worst on `E.coli`, which has a small alphabet of size 4. We see that Kurtz' implementation is about 50% faster than our sibling list implementation. There are probably two main reasons for this: The code is more optimised and it does less. Our child array implementation is fastest the trees, but BPR's suffix array sorting is faster.

	E.coli	bible.txt	world192.txt
MDST struct.sibl.e	5.8	9.5	14.5
MDST struct.sibl.l	5.5	8.7	13.6
MDST comp.sibl.e	5.1	8.3	13.2
MDST comp.sibl.l	5.4	9.1	14.7
MDST comp.lhm.e	7.2	12.4	23.2
MDST comp.lhm.l	7.9	12.9	23.8
MDST comp.lhm.s	8.5	14.0	28.2
MDST comp.arr.e	4.1	5.6	8.9
MDST comp.arr.l	4.2	6.0	10.0
Kurtz	5.0	12.2	15.4
BPR hier m1 k2 d0 s0	7.4	7.3	7.7

Figure 5.8: Average query time in μ-seconds

On queries, the linked hash map implementation is clearly slower, as child traversal when re-

porting hits is more expensive. On texts with bigger alphabets and/or more random text, the linked hash map performs good when reporting only one hit, as shown in figure 5.5(b) earlier.

In this test our sibling list code performs slightly better than Kurtz' on queries in `bible.txt` and `world192.txt`. Our code might be more optimised for enlisting many hits. The child arrays are again the fastest of the trees. The suffix array performs similarly on this test. It is relatively slow on finding a single hit, but fast on reporting many.

	E.coli	bible.txt	world192.txt
MDST struct.sibl.e	475	393	239
MDST struct.sibl.l	359	297	180
MDST comp.sibl.e	130	103	62
MDST comp.sibl.l	114	90	55
MDST comp.lhm.e	462	346	229
MDST comp.lhm.l	445	333	221
MDST comp.lhm.s	434	310	206
MDST comp.arr.e	141	112	68
MDST comp.arr.l	124	99	61
Kurtz	73	51	30
BPR hier m1 k2 d0 s0	28	24	15

Figure 5.9: Memory usage in megabytes

The lazy sibling list implementation with compact nodes is the most space efficient (figure 5.9), but it uses more space than Kurtz' implementation. This is because it does not use small nodes, and has more fields in the nodes. The child arrays does not use much more memory than the sibling lists. The suffix array is by far the most memory efficient.

5.3.6 MDST conclusion

The compact node tree with child arrays and lazy update seem to be a clear winner among the MDST models. It it just as fast or faster on all tests, and does not use much more memory than the most space effective model. It will be used in the comparisons with the static indexes.

5.4 Tuning of hierarchic models using BPR

Using static indexes to solve dynamic indexing problems was explored in section 2.4 and chapter 4.

Since we have found no freely available implementations of substring indexes for dynamic document sets, the MDST will be compared with a hierarchy of static indexes, which are partially rebuilt on document inclusion. The Bucket-Pointer Refinement method (BPR) of [SS05] will

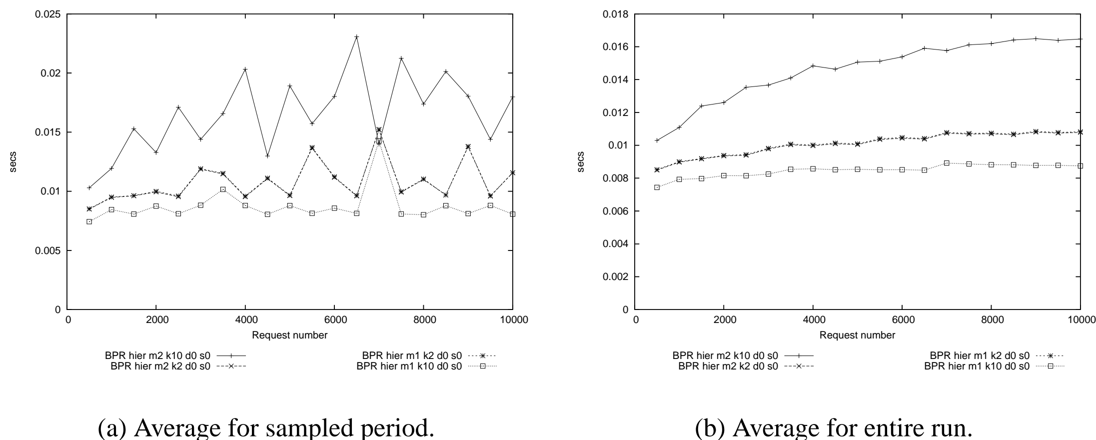
be used, as it is currently the fastest known suffix array construction algorithm. It is freely available from `http://bibiserv.techfak.uni-bielefeld.de/bpr/`, and is released under the GNU General Public Licence.

In the following tests, a configuration of a hierarchic index index using BPR is listed as "BPR hier m=(0|1) k=(2|10) d=(0|1) s=(0|1|2)". The parameter m tells which hierarchic model to use, k is the index size factor, d tells whether or not to use the document ID map, and s is the length of prefixes in the start index. All of these were explained in chapter 4.

5.4.1 Testing method 1 and method 2

For $k = 2$, method 1 and 2 are equal. Choosing which method you want to use, and which k, depends on whether you want to prioritise insertions or queries. For fast document insertions, you use method 1 with a high k, and for fast queries, you use method 2 with high k.

Average insertion times for the test "general" from figure 5.1 are shown in figure 5.10. The left plot gives the average of each sampled period, while the right shows the average for the entire run. The high averages come in periods where one of the bigger indexes in the hierarchies must be rebuilt. As in the theoretical case, method 1 and 2 are identical for $k = 2$.



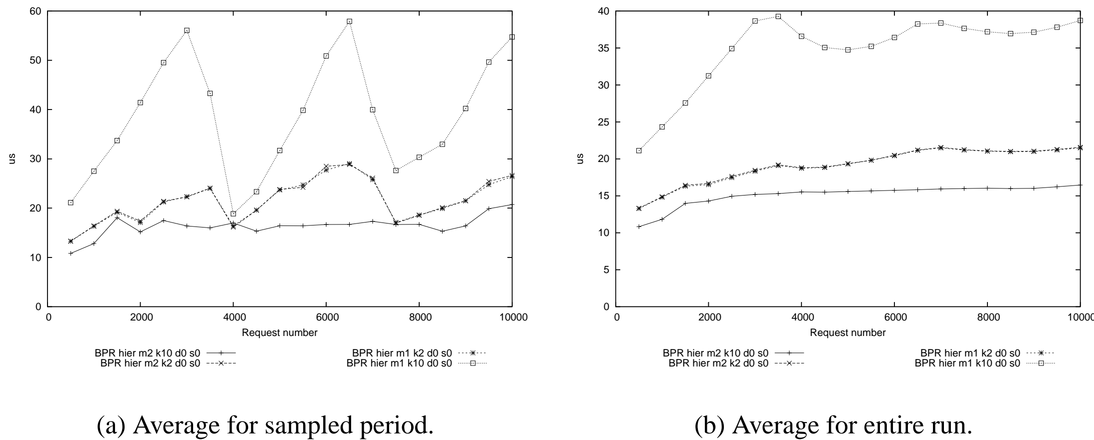
(a) Average for sampled period.

(b) Average for entire run.

Figure 5.10: Average document inclusion time.

The query cost when reporting one hit is given in figure 5.11. As expected, method 1 with $k = 10$ is the most expensive. Notice how the drops in query cost relate to the peaks in inclusion cost seen in figure 5.10(a). A large index is built, and all smaller indexes are merged into this. The query cost for one big index is much less than for many small.

Figure 5.12 shows the memory usage for the given run. Notice that the graph has some drops, even though more data is added to the index than removed. When a document is requested to be removed, it is left in its index. It is put in a blacklist, and the space will not be freed before the containing index is merged into a bigger index.



(a) Average for sampled period. (b) Average for entire run.

Figure 5.11: Average query time.

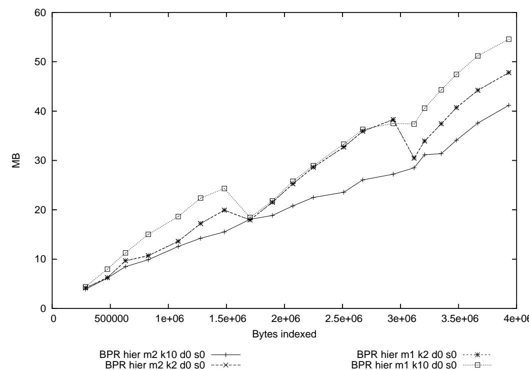
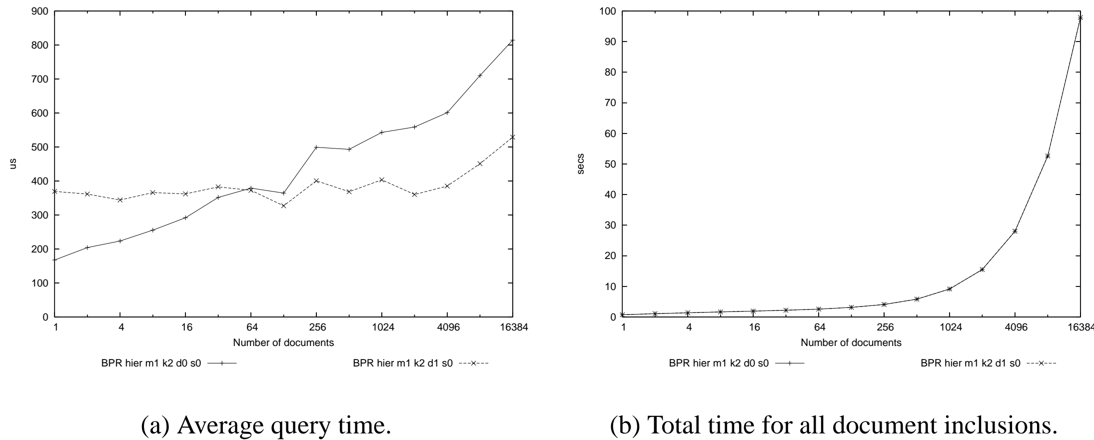


Figure 5.12: Maximum memory usage for sampled period, over total size of data indexed.

5.4.2 Testing docID map

How to map global string positions into document IDs and local position for a suffix array of concatenated strings was described in section 4.5. You either binary search for the document, or find it in a partial lookup array, mapping global position to document IDs. The local position is found in constant time by subtracting the starting position of the document. The following test varies the sizes of the documents, keeping the total data size fixed. When using the map, the price for finding the right document is theoretically constant, but when using binary search, it grows logarithmically on the number of documents, which is the inverse of the document size in this test.

Average query times with and without the document ID map are given in figure 5.13(a). The parameters for the test is given in the column “numdocs” in figure 5.1. You see that the binary search is cheaper than the direct lookup for a small number of documents. This is because of caching effects. The starting positions for all documents fit into a small number of cache lines, as they are stored subsequently. The entries in the document ID map are spread over a large array. As a query hits one portion of it, another portion might be flushed from the cache.



(a) Average query time.

(b) Total time for all document inclusions.

Figure 5.13: Increasing number of documents, constant data size.

This test runs very short queries with many hits, which means the listing of hits dominates over the m and $\log n$ terms. For other types of queries, the effect of the document ID map would be less significant. As you see in figure 5.13(b), there is no noticeable cost for building it.

The conclusion of this test is that the document ID map should be used as long as the number of documents is not very small.

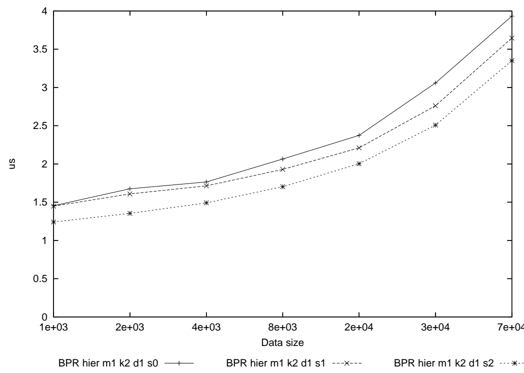
5.4.3 Testing binary search start index

Using a start index for the left and right border in the binary search for queries was described in section 4.6. The cost and effect of this index is dependant on the data size, and the length of the prefixes indexed, s. The time cost of building the index is linear on the size of the data and has an additional $\Theta(\sigma^s)$ space and time cost.

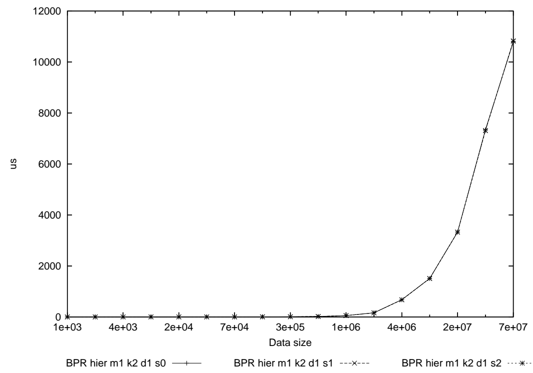
In figure 5.14 you see the query speedup of using the start index. Parameters for this test are given in the column “datasize” in figure 5.1 Two plots are shown, one for small data sizes, and one for larger. Remember from section 4.6 that the savings on using the start index is independent of the data size in theory. With a given s, you save $s \log_2 \sigma$ steps in the binary search on average for random data. In the test, the difference in query cost seems to be approximately constant for the different s on varying data size, as we go above a certain level.

Remember that this is a very artificial test, only one hit is reported per query. With many hits, the binary search would account for less of the time cost, and the advantage of the start index would be even smaller. The speedup is also related to the alphabet distribution. It is highest with a large alphabet with a uniform distribution.

The build cost for the varying s is given in figure 5.15. The cost of building a start index seems to be around 5% for $s = 1$, and 15% for $s = 2$.

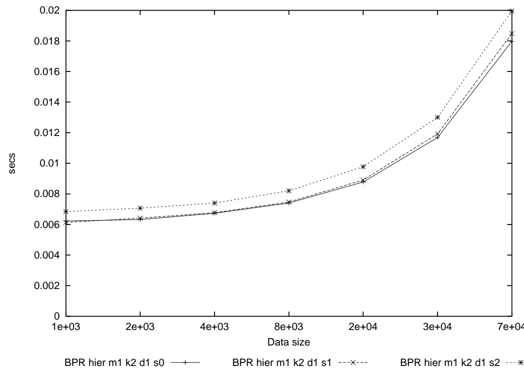


(a) On a small scale.

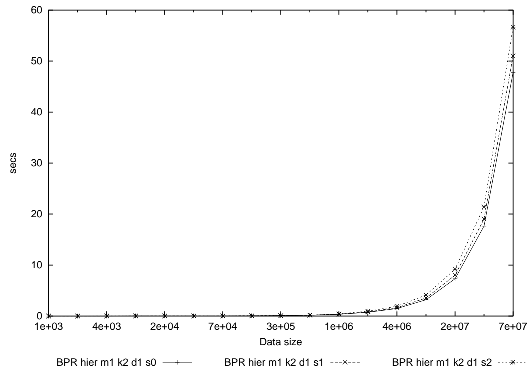


(b) On a larger scale.

Figure 5.14: Average query time. Varying data size.



(a) On a small scale.



(b) On a larger scale.

Figure 5.15: Inclusion time. Varying data size.

Whether or not using a start index is profitable, depends on the size of the data, the text distribution, and whether inclusions or queries are prioritised. As you see, it has little effect when the data is large. The search code is written for random access memory. When considering a non-uniform memory access model, possibly with magnetic disk, things might be different.

5.4.4 Hierarchic index conclusion

Using method 1 or 2 with $k = 2$ seems to be a good choice in the used setup, as it both has rather fast queries (figure 5.10) and document inclusion (figure 5.11). Using a document ID map is advantageous when there are more than just a few documents, while the start index does not give much effect. The variant marked as `BPR hier m1 s2 d1 s0` will be used in the following tests.

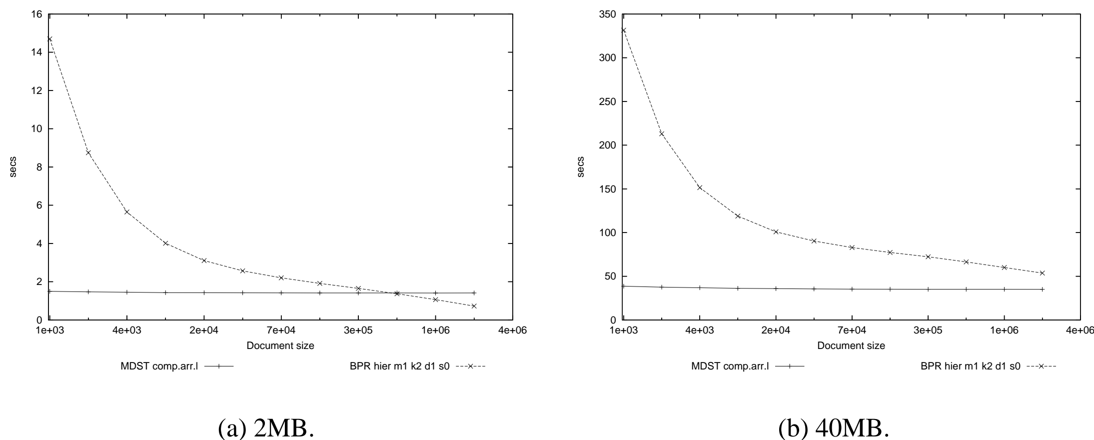
5.5 Comparison of static and dynamic indexes

This section compares the Multi-document Dynamic Suffix Tree with hierarchies of suffix-arrays created with the BPR algorithm.

5.5.1 Varying document size

The major difference between the MDST and the hierarchic suffix arrays, is that the MDST has construction time $\Theta(N)$, where N is the total size of the data indexed, while hierarchic method 1 needs $O(k \log_k N)P_S(N)$ time, and method 2 needs $O(\log_k N)P_S(N)$ time. The upper bound for $P_S(N)$ for BPR in [SS05] is $O(N^2)$, but experiments in [SS05] show it is just as fast as any known $\Theta(N)$ method, even for artificial data. In practise, its running time can be viewed as linear. As seen in figure 5.7, it is about 4 times faster than the MDST.

The following tests add a constant amount of data to the index, varying the document size. The test parameters can be found in the column “docsize” in figure 5.1. Both a test of 2MB, and a test of 40MB are given in figure 5.16, where the total time spent on document inclusions is shown.



(a) 2MB.

(b) 40MB.

Figure 5.16: Total inclusion with varying document size, constant data size.

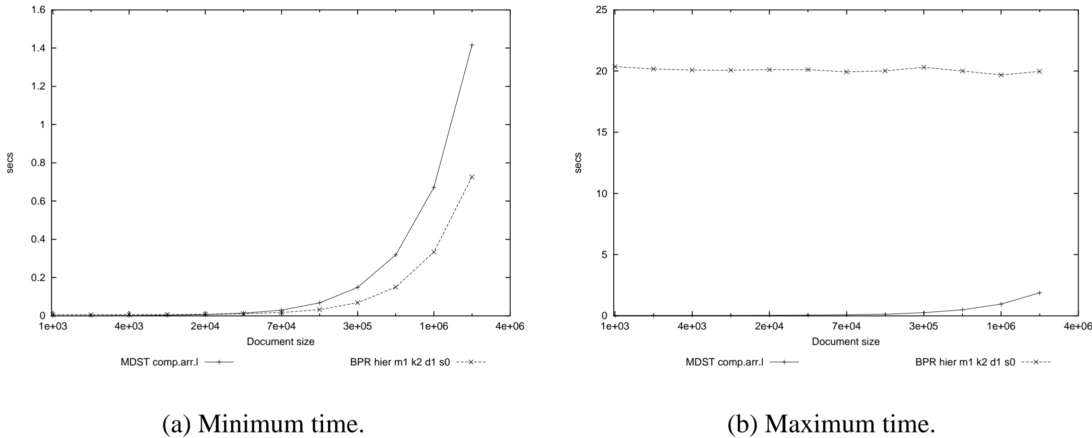
The MDST is faster in this test. For the inclusion of a total of 2MB, the hierarchic indexes are faster from file size 512KB and up, where the number of files is from 4 to 1. This makes sense, as BPR is 2-4 times faster than the MDST on a single document. When the total data amount is increased to 40MB, the number of files at 2MB is 20, and the MDST is faster.

For the hierarchic indexes, if the document size is bigger than k^i, you will not have to rebuild indexes of sizes k^0 to k^i. If there is only one document, there is no overhead with the hierarchic indexes. For a given total amount of data, the hierarchic indexes clearly prefer large documents.

You also see a slight decrease in the total cost for the MDST when there are larger but fewer

documents. This has to do with document management.

The figures 5.17(a) and 5.17(b) give the minimum and maximum time for document inclusion. There might not be many, if any, applications for substring indexes where the time deviation is important. In a real application, the big indexes would be rebuilt in a separate thread. Queries would be given to the old indexes and a set of small fresh indexes, until the new big indexes are finished.



(a) Minimum time. (b) Maximum time.

Figure 5.17: Minimum and maximum inclusion time, 40MB total data.

Memory usage for the methods is shown in figure 5.18. As can be seen, both methods use an amount approximately linear on the total data size, and independent of the document sizes. The space needed for document management is insignificant. The MDST uses about three times the memory of the hierarchic suffix arrays.

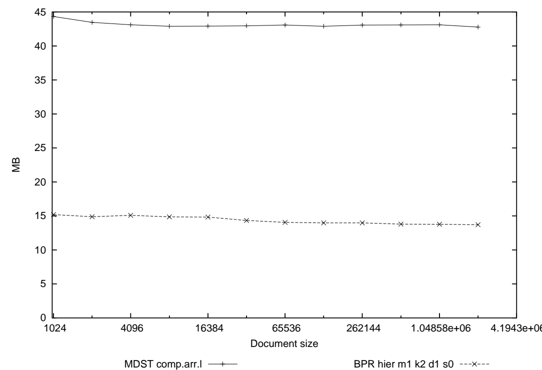


Figure 5.18: Memory usage

Queries was not tested here, because many parameters influence which method is better. Various tests for queries are given later in this chapter.

Most of the rest of the tests are run on 4KB files. Originally this was chosen because this was the point at which the MDST with sibling lists had the same performance as the hierarchic indexes in this test. The child arrays were implemented last, and turned out to be much more

effective. But it would not make sense to change the rest of the test to use 10MB files. 4KB is representative for many applications. On the web, most documents are between 1KB and 10KB, when HTML tags are removed. Therefore, the 4KB tests are kept as they were.

5.5.2 Varying freshness requirement

In most real-life cases, it is not necessary to index a document the moment it is received. The inclusion may be postponed for a given time, or the index is updated after some best effort scheme. Since our tests are not run in a "real-time" environment with incoming documents, the freshness requirement is given as "how many documents can we postpone adding before we have to add them".

Figure 5.19 shows average document inclusion times for a varying freshness requirement. Parameters for the test are given in the column "freshness" in figure 5.1. When you compare with the test of variable file sizes in figure 5.16, you see that more slack on the freshness requirement is the same as indexing bigger files given the same amount of data. It is not certain why there is a peak in the plot for the hierarchic indexes.

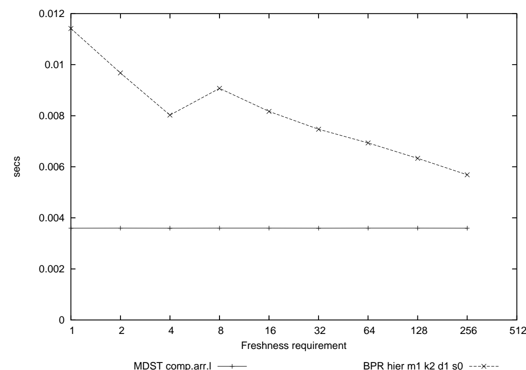


Figure 5.19: Average inclusion time for varying freshness requirements

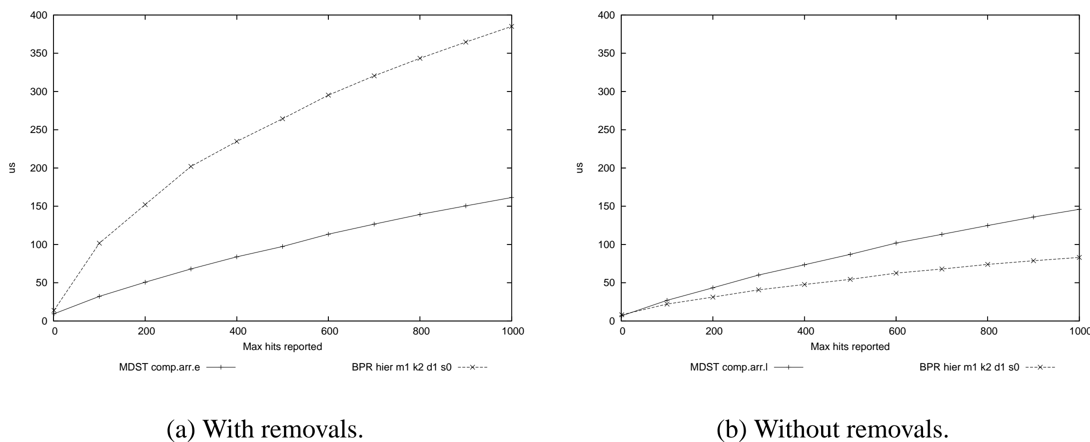
This test concludes that hierarchic indexes should be avoided if you have an absolute freshness requirement. Note that this is for the given setup of substring indexes. The results might be different for another update model of the hierarchic indexes.

5.5.3 Varying number of hits

The cost of reporting a large number of hits has the same complexity for a suffix tree and a suffix array when the $\log n$ term does not dominate. Reporting occ occurrences costs $\Theta(occ)$ time in addition to the cost of locating one hit. But in practise the cost is not the same. In a suffix tree, you must traverse the entire subtree below a given point, which has bad memory

locality. In a suffix array, you just read a range of consecutive memory locations, which gives a good memory locality and cache utilisation.

Interesting behaviour was seen when experimenting with this test. In the first setup, the hierarchic indexes performed unexpectedly bad (figure 5.20(a)). It turned out this was because the were many requests for document deletion given to the indexes. Parameters are given in figure 5.1. A large portion of the documents reside in the largest index in the hierarchy, which is not rebuilt before all indexes are full. So if there are a lot of deletions, there will be a lot of false hits which have to be traversed before the requested number of hits are found. Because of this, a similar test without document removals was run. The results can be seen in figure 5.20(b). The averages given in both figures are measured for the last 10% of the requests. As you can see, the hierarchy of suffix arrays is much faster when there are no deletions. Hit reporting is also faster for the lazy MDST, as no document ID cleanup must be performed.



(a) With removals. (b) Without removals.

Figure 5.20: Reporting a variable number of hits for a set of documents.

To speed up queries in the hierarchic indexes, one should consider rebuilding indexes when they contain to many removed documents. This is not done in the tested implementation. More rebuilds would give a higher maintenance cost.

5.5.4 Varying query length

In figure 5.21 follows a test measuring lookup time for varying query length, reporting 1 hit. The query length is varied between 1 and 50. Only one hit is reported. For the hierarchic indexes, the cost increases greatly up to a query length of 10. Queries are sampled evenly from the data, so common words in the distribution are seen often when the queries are short. This means the same jumps will be performed in the binary search often, meaning more accesses to CPU cache, and less to main memory.

It is not certain why we do not see the same effect with the MDST. The plot shows that the cost of finding the right position in the tree is low compared to the overhead of processing the query.

The cost is almost constant. It might mean that the suffix tree has better memory locality than the virtual tree seen in the binary search of the suffix array.

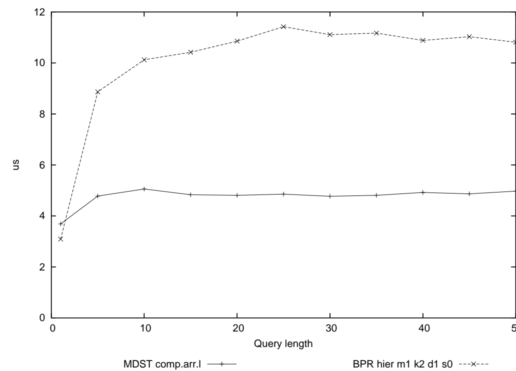


Figure 5.21: Reporting one hit, varying query length.

This test might not be very relevant in practise, as searches with a single hit is not very common.

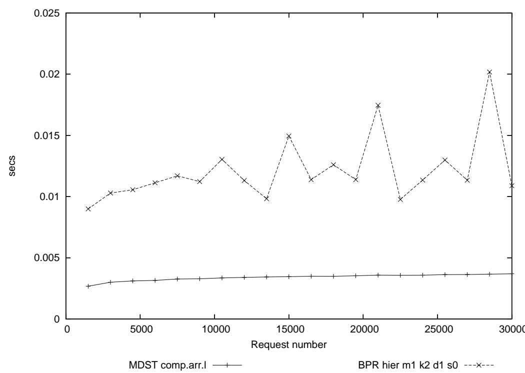
5.5.5 Increasing total data size

Below follows a test of increasing total data size. Documents of 4KB are added to the indexes. Inclusions and queries run mixed, and average times are collected periodically, both for the sampled period and the entire run. The parameters are taken from the column "general2" in figure 5.1.

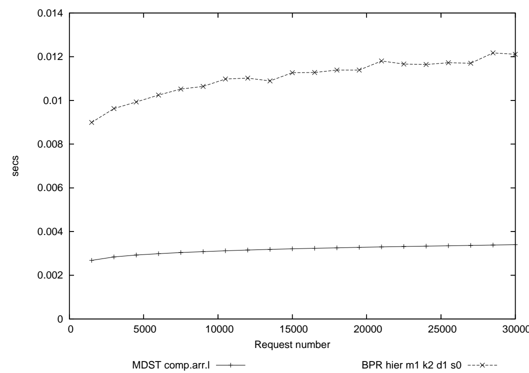
As can be seen in figure 5.22, the MDST is about three times faster than the hierarchic indexes on document inclusion, and has a much lower time deviation. Figure 5.22(d) shows that the maximum inclusion time is about 400 times the average for the hierarchic indexes, 4.8 versus 0.012 seconds. Note that this deviation could be hidden by running the rebuild of the big indexes in separate threads. The minimum inclusion time is the same in all sampled periods for the hierarchic indexes. It is the cost of building the smallest index. If all documents have the same size, this happens in every second inclusion. For the MDST, the minimum and maximum are very close to the average.

In figure 5.23 you see the average and maximum query times seen in the sampled periods. A short query is given to the index, and all hits are returned. The two methods perform roughly the same, and query time seems to grow linearly. There are very many hits for some of the queries, and the time to report these clearly dominates. As many as 150000 hits were reported at the most.

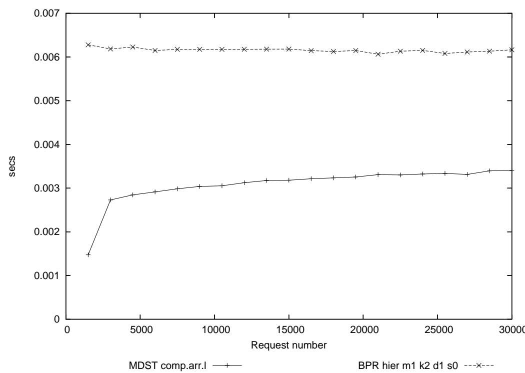
Figure 5.24(a) shows the average removal times. For the hierarchic indexes, it is very low, while for the MDST, it is close to the average inclusion time. In the MDST, postponing the removal of a document does not give any advantage. The cost is always linear on the size of the document. For the hierarchic indexes, the cost of fully removing a document is dependant on the total data size in the worst case, and blacklists are much preferred. If you just used a blacklist in the



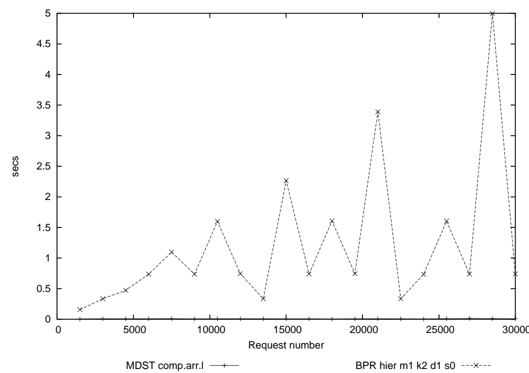
(a) Average for sampled period.



(b) Average for entire run.



(c) Min. for sampled period.



(d) Max. for sampled period.

Figure 5.22: Inclusion time. Increasing data size.

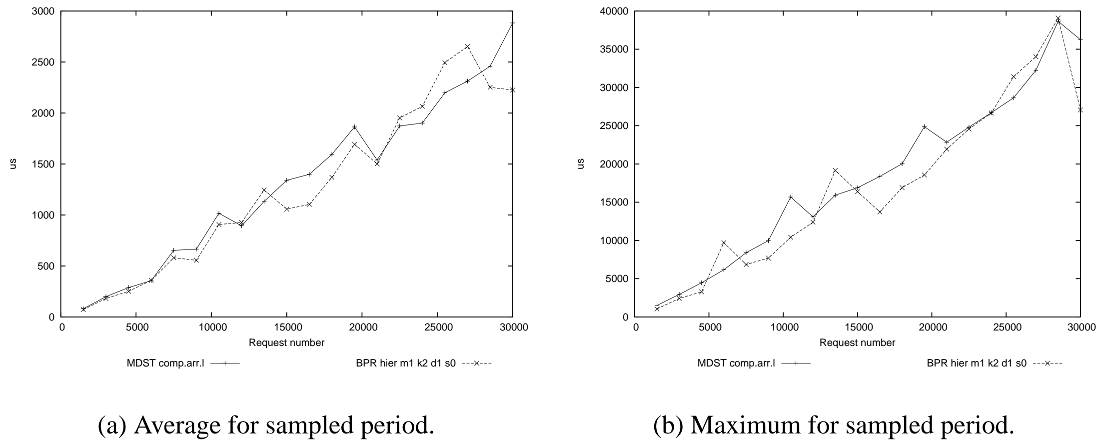
MDST, and left data there, the tree would get a higher fill, and document inclusion would be slightly slower. Hit reporting would also be more expensive.

Figure 5.24(b), shows the average for all operations: Inclusions, removals and queries. The increase in query time as the data grows makes up for most of the increase in the total average. We see that the MDST is still slightly faster than the hierarchic indexes.

Figure 5.25 shows the memory usage for this test. The values would be lower for the hierarchic indexes if there were no document deletions. Remember from figure 5.9 that they use about one fourth of the memory of the MDST with child arrays.

5.6 Critique of the tests

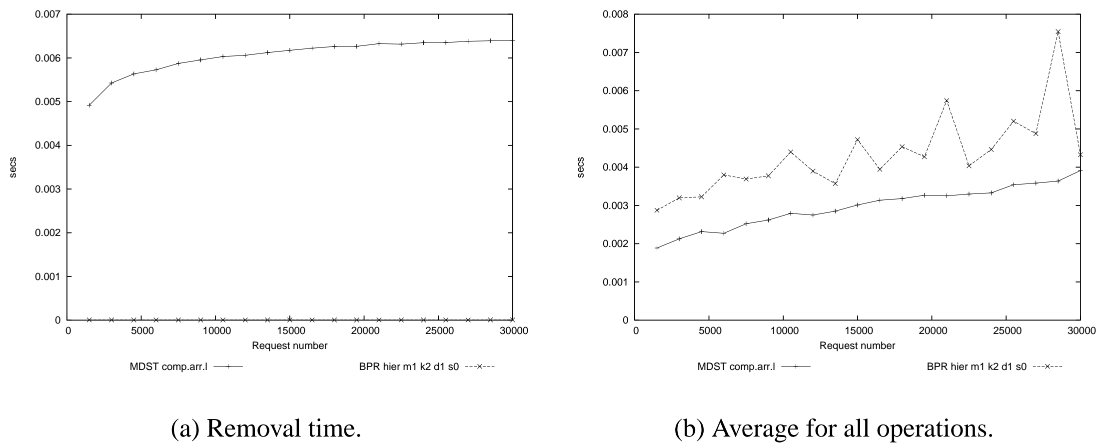
The tests run in the experiments in this report are artificial. The results might have more theoretical than practical validity, but the give many clues as to how real-world systems could be



(a) Average for sampled period.

(b) Maximum for sampled period.

Figure 5.23: Query time. Increasing data size.



(a) Removal time.

(b) Average for all operations.

Figure 5.24: Average for sampled period. Increasing data size.

made, and especially how to set up experiments for real systems.

The test environment is a bit different from most real situations. The requests are given sequentially to the system, and are performed one by one. In most systems indexing large amounts of data, inclusions and queries are run in separate threads. This is essential for hierarchic indexes. The system cannot wait while the biggest indexes are rebuilt. Therefore, the big indexes are rebuilt as a new copy, while the old indexes, plus the small fresh indexes are queried. When the build is done, the old index is replaced with the new. When this is run in separate threads, the long latencies are hidden.

In most systems, unique documents found are returned from queries, not all pairs of documents and local positions. This adds some complexity, but it would be the same in both the tested MDST and in the hierarchies of suffix array. For inverted file word indexes, reporting only documents is a cheaper problem.

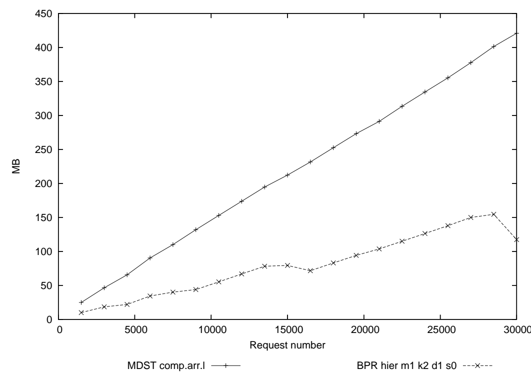


Figure 5.25: Memory usage.

Keeping data and indexes in RAM is also not possible in most applications. The amount of data is usually far too large. It would be very interesting to run the same experiment on similar structures on disk.

There are two main reasons the test environment and tools were designed as they were. Firstly, completing the project had to be feasible within the given time frame. Secondly, this artificial environment gives very good insight in the behaviour of the data structures in different situations, as the test parameters are adjusted very easily.

Chapter 6

Conclusion

The child array model for the MDST was implemented during the last weeks of writing this report, and it changed many of the conclusions drawn. It was conceived and implemented because effects of memory caching was seen in many of the tests. Especially notable was the difference between the binary search and the direct lookup in figure 5.13(a). The author has not seen child arrays used in articles describing suffix trees before, probably because a doubling factor of 2 would make them space inefficient. But a doubling factor of 1.1 worked very well here.

Before the child arrays were implemented, the lazy update sibling lists were compared with the hierarchic index. The latter were faster for documents larger than 4KB in the test shown in figure 5.16. It was concluded that the MDST was only better for small documents. It is of course possible that the hierarchic indexes could see a similar speedup, and that the conclusions would change again.

6.1 Use cases

If the tests in this report are representative, we can draw some conclusions about in what cases you should choose what type of index. If the size of your document set is large, but there is little change, then the hierarchies of suffix arrays are the better choice. They use one third of the memory, and report hits faster when there are few document removals.

If you have small documents, many updates, and an absolute fresness requirement, then the MDST is clearly better. The hierarchy spends a lot of time rebuilding many small indexes. Note that this is for the compared hierarchic index using method 1 with $k = 2$. For a higher k with method 1, inclusion costs would be much lower, at the cost of slower queries.

The MDST should also be chosen if queries need to be fast, and you have a rapidly changing document set. In our setup this would mean that the ratio of deletions to inclusions is high. The given implementation of hierarchic indexes did not do very well on queries if there had been a

lot of deletions in the document set. Note that this could be different in another implementation.

All this is under the assumption that the environment is comparable to our artificial testing environment.

6.2 Further work

The most important step after these experiments would be to run a similar experiment on disk based substring indexes. This would not be an easy step. There is currently a lot of research being done on disk resident suffix structures, and much work would have to be done to become familiar with this. Both suffix trees and suffix arrays work very badly on disk in their basic form, as they have bad spatial locality. Disk based indexes also makes the implementation much more complex, as data must be stored and accessed in a clever order, and multi-threading must be used to get good performance. Creating an efficient disk based substring index for dynamic document sets would be a very important achievement, both scientifically and commercially.

A simpler thing that could have been done is to set up experiments more similar to a real life scenario, where queries and new documents come in through different pipelines, and indexes are built in the background. This would hide the great time deviation for document inclusions with the hierarchic indexes. If you had such a setup, you could also tune the factor k in your hierarchy implementation to minimize the load on your system, based on the amount of inclusions versus the amount of queries.

Bibliography

[BP] Tim Bell and Matt Powell. The canterbury corpus.

[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.

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

[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.

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

[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.

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

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

[MM91] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. 1991.

[Mor68] Donald R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MUM] The Institute for Genomic Research MUMmer. Webpage. *http://www.tigr.org/software/mummer/.*

[OvL80] Mark H. Overmars and Jan van Leeuwen. Some principles for dynamizing decomposable searching problems. Technical Report RUU-CS-80-1, Rijksuniversitet Utrecht, 1980.

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

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