Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Departamento de Informática

An Implementation of Dynamic Fully Compressed Suffix Trees

Miguel Filipe da Silva Figueiredo

Dissertação apresentada na
Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa
para obtenção do grau de Mestre em Engenharia Informática

Orientador: Prof. Dr. Luís Russo

19 de Fevereiro de 2010

Resumo

Esta dissertação estuda e implementa uma árvore de sufixos dinâmica e comprimida. As árvores de sufixos são estruturas de dados importantes no estudo de cadeias de caracteres e têm soluções óptimas para uma vasta gama de problemas. Organizações com muitos recursos, como companhias da biomedicina, utilizam computadores poderosos para indexar grandes volumes de dados e correr algoritmos baseados nesta estrutura. Contudo, para serem acessíveis a um publico mais vasto as árvores de sufixos precisam de ser mais pequenas e práticas. Até recentemente ainda ocupavam muito espaço, uma árvore de sufixos dos 700 megabytes do genoma humano ocupava 40 gigabytes de memória.

A árvore de sufixos comprimida reduziu este espaço. A árvore de sufixos estática e comprimida requer ainda menos espaço, de facto requer espaço comprimido optimal. Contudo, como é estática não é adequada a ambientes dinâmicos. Chan et al. [3] descreveram a primeira árvore dinâmica comprimida, todavia o espaço usado para um texto de tamanho n é $O(n \log \sigma)$ bits, o que está longe das propostas estáticas que utilizam espaço perto da entropia do texto. O objectivo é implementar a recente proposta por Russo, Arlindo e Navarro[22] que define a árvore de sufixos dinâmica e completamente cumprimida e utiliza apenas $nH_k + O(n \log \sigma)$ bits de espaço.

Abstract

This dissertation studies and implements a dynamic fully compressed suffix tree. Suffix trees are important algorithms in stringology and provide optimal solutions for myriads of problems. Suffix trees are used, in bioinformatics to index large volumes of data. For most aplications suffix trees need to be efficient in size and funcionality. Until recently they were very large, suffix trees for the 700 megabyte human genome spawn 40 gigabytes of data.

The compressed suffix tree requires less space and the recent static fully compressed suffix tree requires even less space, in fact it requires optimal compressed space. However since it is static it is not suitable for dynamic environments. Chan et. al.[3] proposed the first dynamic compressed suffix tree however the space used for a text of size n is $O(n \log \sigma)$bits which is far from the new static solutions. Our goal is to implement a recent proposal by Russo, Arlindo and Navarro[22] that defines a dynamic fully compressed suffix tree and uses only $nH_0 + O(n \log \sigma)$ bits of space.

Contents

1 Introduction **6**

1.1 General Introduction 6

1.2 Problem Description and Context 7

1.3 Proposed Solution 10

1.4 Main Achievements 10

1.5 Simbols and Notations 12

2 Related Work **14**

2.1 Text Indexing 14

2.1.1 Suffix Tree 14

2.1.2 Suffix Array 17

2.2 Static Compressed Indexes 20

2.2.1 Entropy 20

2.2.2 Rank Select 21

2.2.3 FMIndex 27

2.3 Static Compressed Suffix Trees 30

2.3.1 Sadakane Compressed 30

2.3.2 FCST 32

2.3.3 An(Other) entropy-bounded compressed suffix tree . . 36

2.4 Dynamic Compressed Indexes 38

2.4.1 Dynamic Rank and Select 38

2.4.2 Dynamic compressed suffix trees 43

2.4.3 Dynamic FCST 49

3 Design and Implementation **56**
3.1 Design . . . 56
3.2 DFCST . . . 57
3.2.1 Design Problems . . . 58
3.2.2 DFCST Operations . . . 60
3.3 Wavelet Tree . . . 65
3.3.1 Optimizations . . . 65
3.3.2 Operations . . . 67
3.4 Bit Tree . . . 69
3.4.1 Optimizations . . . 69
3.4.2 Buffering Tree Paths . . . 70
3.5 Parentheses Bit Tree . . . 75
3.5.1 The Siblings Problem . . . 77

4 Experimental results **80**
4.1 Bit Tree . . . 81
4.1.1 Determining the arity of the B+ tree . . . 82
4.1.2 Performance . . . 84
4.1.3 Comparing Results . . . 86
4.1.4 Dynamic Environment Results . . . 88
4.2 Parentheses Bit Tree . . . 88
4.3 Wavelet Tree . . . 91
4.4 DFCST . . . 92
4.4.1 Space used by the DFCST . . . 92
4.4.2 The DFCST operations time . . . 92
4.4.3 Comparing the DFCST and the FCST . . . 93

5 Conclusions and Future work **95**
5.1 Conclusions . . . 95
5.2 Future work . . . 97

List of Figures

2.1 Suffix tree for the text "mississippi". 15
2.2 A sub-tree of the suffix tree for string "mississippi". 17
2.3 Suffix array of the text "mississippi". 18
2.4 *Rank* of of size four bitmaps with two "1"s. 23
2.5 Static structure for rank and select with superblocks, blocks and bitmaps. 24
2.6 Bitcoding for the text "mississippi". 25
2.7 The wavelet tree for the text "mississippi". 25
2.8 Matrix M of the Burrows Wheeler Transform. 28
2.9 Matrix M of the Burrows Wheeler Transform with operations computed during a backward search. 30
2.10 Parentheses representation of the suffix tree for "mississippi". . 31
2.11 Sampled suffix tree with suffix links. 34
2.12 A LCP table and the operations required to perform $SLink$(3,6). 37
2.13 A binary tree with p and r values at the nodes and bitmaps at the leaves. 40
2.14 Matching parentheses example. 45
2.15 Computing *opened* values for blocks. 46
2.16 Computing *closed* values for blocks. 46
2.17 Computing *closed* and *opened* values for several blocks. 47
2.18 Computing LCA of two nodes. 48
2.19 Reversed tree of the suffix tree for "mississippi", also the sampled suffix tree of "mississippi". 52

3.1 Classes diagram. 57

3.2 The suffix tree structure and the DFCST structure. 58
3.3 A sampling creates a sampled node with the wrong string depth. 59
3.4 Leaf insertion creating a sampling error. 60
3.5 Weiner node computation. 61
3.6 Computation of LCA. 62
3.7 Child computation with a binary search. 64
3.8 The wavelet tree data structure using a binary tree. 65
3.9 The wavelet tree data structure using a heap. 66
3.10 Compact wavelet tree data structure with a binary tree. . . . 67
3.11 Sorted access implementation for the wavelet tree. 68
3.12 Compact wavelet tree data structure with a heap. 68
3.13 The bit tree buffering. 72
3.14 The redistribution operation of a critical bitmap block. 74
3.15 Compacting the parentheses tree. 77

4.1 The space ratio for bit trees with arity 10, 100 and 1000. . . . 82
4.2 The build time for bit trees with arity 10, 100 and 1000. . . . 83
4.3 The time used to complete a batch of operations for bit trees with arity 10, 100 and 1000. 83
4.4 The space ration for the RB bit tree and the B+ bit tree. . . . 84
4.5 The build time statistic for the RB bit tree and the B+ bit tree. 85
4.6 The time statistic for completion of a batch of operations with the RB bit tree and the B+ bit tree. 85
4.7 Bit trees compared over three criteria. 86
4.8 Comparing the bit tree normal build and the alternated deletion build. 88
4.9 Comparing the space used per node of normal trees and parentheses trees. 89
4.10 Comparing wavelet trees space. 91
4.11 Comparing wavelet trees speed. 91
4.12 DFCST space use results. 93
4.13 DFCST speed results. 93

List of Tables

1.1 The table shows notations used. 12
1.2 The table shows notations used. 13

2.1 The table shows time and space complexities for Sadakane static CST and Russo et al. FCST. 36
2.2 The table shows time and space complexities for Chan et al. dynamic CST and Russo et al. dynamic FCST. 55

4.1 Test machine descriptions. 80
4.2 DFCST and FCST operations time. 94
4.3 DFCST, FCST and CST space. 94

Chapter 1

Introduction

1.1 General Introduction

The organization of data and its retrieval is a fundamental pillar for the scientific development. In the 20Th century the science of information retrieval has matured, its early days may be traced to the development of tabulating machines by Hollerith in 1890. From the 1960s through to the 1990s several techniques were developed showing that information retrieval on small documents was feasible using computers. The Text Retrieval Conference, part of the TIPSTER program, was sponsored by the US government and focused on the importance of efficient algorithms for information retrieval of large texts. The TIPSTER created a infrastructure for the evaluation of text retrieval techniques on large texts, furthermore when web search engines were introduced in 1993 the area of data retrieval continued to prosper. Search engines relish on information retrieval and are strong investors in such technology.

We are interested in bio-informatics, this area has been around since the beginning of computer science in the mid 20th century. Bio-informatics takes

advantage of the latest developments on databases, algorithms, computational and statistical techniques to study problems that are otherwise too complex in today's computers. Exact matching is a sub-task used in several problems dealing with DNA, RNA, processed patterns or amino acid rings. As such developments in this area provide benefits to a wide range of bio-informatics applications.

1.2 Problem Description and Context

Searching for genes in DNA is a common problem in bio-informatics. Finding a specific gene sequence within the human genome is a big task and feasible only with the most advanced computers. The results are otherwise prohibitively time consuming. Several problems arise when searching for damaged or mutated genes and the algorithms used can consume a lot of time to process these queries. Problems related to finding a specific string within other string are referred to as exact string matching. Finding a string with some form of errors is a inexact string matching. String matching is not limited to DNA, in fact these problems are of great use for other scientific areas, for example it is essential for large Internet search engines, since information grows exponentially yearly.

The digital search tree called trie was defined by Ed Fredkin in 1960 [6]. Storing suffixes description through the tree path and text position at leaves. The trie is also called a prefix tree because all node descendants share a common prefix. In 1973 Weiner introduced the *position tree*[25], known today as suffix tree. It was described by Donald Knuth as "the algorithm of the year 1973"[10]. The solution for a variety of problems with exact and inexact string matching can be solved with a suffix tree in optimal or near optimal time. It is also possible to store the indexes in linear, $O(n \log n)$ bits, space. Other solutions envolve linear search over a large database and are not good for large problems, for example the human DNA spans 700 megabytes

which yields an extremely large suffix tree. Suffix trees have problematic space requirements. Uncompressed suffix trees can use from 10 to 20 times the size of the original text. A simple implementation of a suffix tree for the human genome consumes 40 gigabytes, as a consequence this tree requires secondary memory. In this case operations will slow down so extremely that it renders the structure useless[10]. Hence several techniques were created to save space and represent suffix trees in a more practical size.

Suffix arrays were developed by Eugene Myers and Udi Manber and in parallel by Baeza Yates and Garton Gonet. These data structures are based on suffix trees but can be stored in much less space[16]. They consists in a array with the starting positions of all suffixes in the the text arranged by lexicographical order. Suffix arrays are used to locate suffixes within a string and can be extended to determine the longest common prefixes of any two suffixes, however they can support only a limited subset of the operations provided by the suffix tree, still by adding extra structures it is possible to simulate a suffix tree algorithm[2].

Another classical attempt to reduce space is the directed acyclic graph. It represents strings and supports constant time search for suffixes. During construction isomorphic sub trees are detected and it generates a acyclic directed graph instead of a tree.

The main achievement and difference between tries and dags is the elimination of suffix redundancy in the trie. Both tries and dags eliminate prefix redundancy but only dags eliminate both prefix and suffix redundancy.

The classical online solutions for pattern matching have linear time algorithms. The most well known linear approaches are the Boyer Moore, Knuth Morris Pratt and Aho Corasick algorithms. These algorithms are well documented hence it is expected that suffix trees have more room for research. If size of the text is n, and the size of the patterns is m each and the number of patterns is α, the Boyer Moore and Knuth Morris Pratt use linear time to preprocess the patterns and linear time to search the text, $O(\alpha \times n + m)$.

Suffix trees solve the pattern matching problem in the same worst case linear time $O(n+\alpha\times m)$. However because the preprocessed string is the text, and with a larger α, time is linear on the size of the patterns which normally is much smaller. When there are several patterns to search the speed advantage of reusing the preprocessed suffix tree is obvious. The linear algorithms will always spend time preprocessing the patterns and running the text in linear time while suffix trees only need to preprocess the text once and run the patterns in linear time.

The Aho Corasick achieves a time similar to suffix trees as it matches a set of patterns with a text. However the largest speed advantage of suffix trees over classical linear algorithms appears in problems that are more complex than exact string matching. For example the longest common substring problem can be solved in linear algorithms by suffix trees and there are no other known linear time algorithms. The linear algorithms will always search for a exact string at every run, however the suffix tree has a structure that allows it to find all substrings in the same run with little time cost.

Suffix arrays can be enhanced to support all functions that suffix trees do with similar times and additional structures. Since suffix trees solve different classes of problems thanks to some properties(suffix links, bottom up and top down traversal) the enhanced suffix arrays add structures that simulate all those properties. Though using less space than conventional suffix trees the overall space cost is high.

The compressed suffix tree with full funcionality presented by Sadakane in 2007 improves on the space used by suffix trees from $O(n\log\, n)$ bits down to $O(n\log\,|\sigma|)$ bits[24]. Albeight a significant progress, these results are still far from the desired space restrictions of a minimal suffix tree. Makinen et al. [15] developed an entropy bounded compressed suffix tree which improves the result presented by Sadakane[24]. However that suffix tree does not yet achieve optimal compressed space as the one presented by Russo, Arlindo and Navarro [12]. The fully compressed suffix tree presents the smallest

space result for a compressed suffix tree. It discards the space used by the compressed suffix tree of Sadakane and achieves a smaller space than the entropy bounded compressed suffix tree[24]. It achieves a compressed space suffix tree for the first time, the times of most operations are logarithmic or near logarithmic. The implementation of this structure recently presented by Russo has proven this result. This dissertation compares results with the implementation by Russo.

The recent space reduction by Russo, Arlindo and Navarro takes static suffix trees down to compressed space[12]. Although this is a important result, this suffix tree as well as other implementations of similar sizes, is statical. The lack of dynamic insertions and deletions, makes these implementations unsuited for problems within a dynamic environment. This has a significant impact in the space that is used at build time. During this initial fase the suffix tree needs space that is discarded once construction is complete.

1.3 Proposed Solution

A solution proposed by Russo, Arlindo and Navarro[22] solves both the dynamic suffix tree problem and the construction space problem. There is a cost of adding dynamic capabilitie to the fully compressed suffix tree, which is a logarithmic slow down on the operations. This dissertation proposes to implement this dynamic fully compressed suffix tree.

1.4 Main Achievements

Our main achievements are a library with a fully compressed dynamic suffix tree and experimental tests with a large text. Hence we obtained a small dynamic suffix tree with efficient operations. Hence we are able to build a

compressed suffix within reasonable space not overcoming the final space. We obtain the following results:

- An implementation of a dynamic suffix tree.
- A small dynamic suffix tree, not larger than twice the text.
- Efficient operations whose results are predicted by Russo, Arlindo and Navarro[22].

1.5 Simbols and Notations

Table 1.1: The table shows notations used.

Notation	Definition
n	size of a text
m	size of a pattern
α	size of a text
δ	the sampling
σ	the size of a alphabet
T	a suffix tree
T^{-1}	the reversed suffix tree
i	a index position
v	a node
$SLink(v)$	the suffix link of node v
$SDep(v)$	the string depth of node v
$TDep(v)$	the tree depth of node v
$LCA(v, v')$	the lowest common ancestor of node v and v'
$LETTER(v, i)$	the i-th letter in the path label of v
$Parent(v)$	computes the parent of node v
$excess$	number of opened parentheses minus the number of closed parentheses in a balanced sequence of parentheses
$minexcess(l, r)$	position i between l and r such that $excess(l, i)$ is the smallest in the range (l, r)
SA	suffix array
CSA	compressed suffix array
CST	compressed suffix tree
FCST	fully compressed suffix tree
FMIndex	full text index in minute space

Table 1.2: The table shows notations used.

Notation	Definition
BWT	Burrow Wheeler transform
LF	last to first mapping
M	matrix with the cyclic shifts of a text
L	the last comumn in M
F	first column in matrix M
LCP	computes the longest common prefix over a range of suffixes
$\psi(v)$	psi link of node v
s	a string
c	a caracter
S_j	superblock number j
B_k	block number k
G_c	the table of class c
C_c	computes the number of characters lexicographically smaller than c in the text
$	a caracter that does not exist in the text
opened	unmatched opened parentheses over a balanced sequence of parentheses
closed	unmatched closed parentheses over a balanced sequence of parentheses
I	integers sequence
$Occ(c, k)$	counts the number of occurrences of character c before position k in column L
$WeinerLink(v, c)$	computes the weiner link of node v using letter c
A	a bitmap
$Rank_c(s, i)$	counts letters 'c' in s up to position i
$Select_c(s, i)$	computes the index position of the i'Th occurrence in s
n_c	number of occurrences of c

Chapter 2

Related Work

2.1 Text Indexing

In section 2.1.1 we describe the suffix tree data structure which, like other full text indexes, is an important tool for substring searching problems. However suffix trees tend to be excessively large, ie 10 to 20 times the text size. Since suffix trees are very functional hence reducing their space requirements is crucial. In section 2.1.2 we describe suffix arrays, which are more space efficient full text indexes.

2.1.1 Suffix Tree

The suffix tree was presented in 1973[25] and its potential was noticeable, Donald Knuth referred to it as the algorithm of the year. The suffix tree saved several string problems in optimal time. Such a promising discovery would be expected to had a wider audience but the first academic papers were hard to understand and therefore its dissemination stalled. However

the suffix tree is not complicated although it is not trivial either.

A suffix tree for a text T of size n is a rooted directed tree with n leaves. Each leaf has a distinct value from 1 to n. All tree edges have a non empty label with a string. Every path from the root that ends on a leaf i describes a suffix starting at i. The concatenation of the labels down to the leaf describe the suffixes so finding a pattern is done with comparisons between the labels and the pattern. Every internal node except the root must have at least two children. No two children out of a node may start with the same letter.

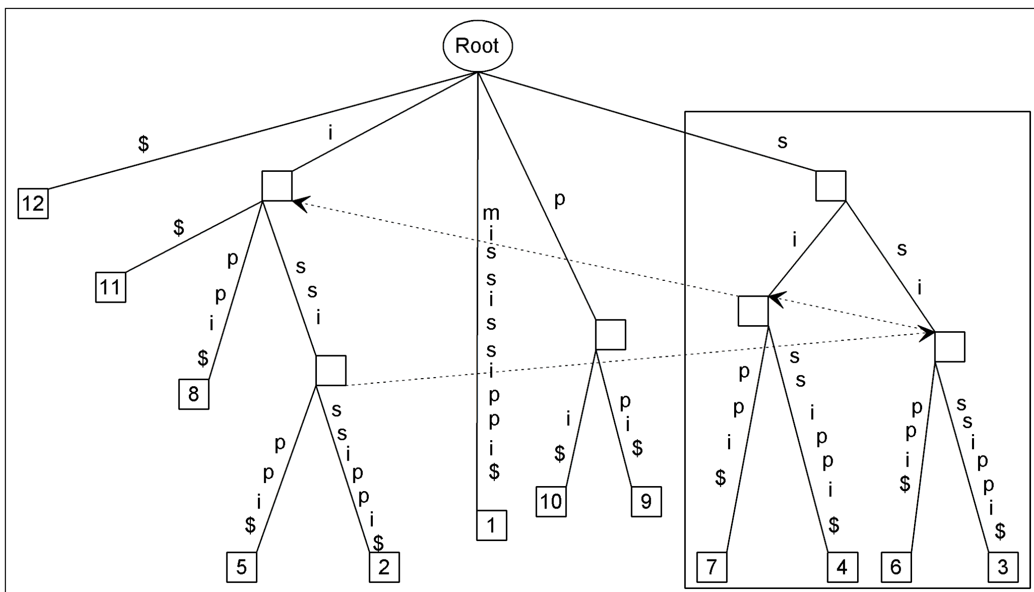


Figure 2.1: Suffix tree for the text "mississippi". The values at the leaves are the initial positions of a suffix in the text. To the right the sub-tree containing leaves 3, 4, 6 and 7 is in a box. The suffix links of the internal nodes are shown with dotted arrows.

To perform exact pattern matching, for example, find "ssi" in "mississippi" using the suffix tree. Start at the root and search for the path with a label that starts with the first character in the pattern. That is the fourth path from the left with label "s". The next node has only two paths, one has a label "si" and the other has "i". Since "s" is already used the text "si" is still missing. Now choose the node whose label "si" matches the current pattern. At this point all characters of "ssi" are found. The sub-tree at the end of the

pattern matching contains all the occurrences of the pattern, each leaf of the sub-tree represents one occurrence. For the example with "ssi" there are two occurrences at position 5 and 2, which are respectively the suffixes "ssippi" and ssissippi". If the pattern was "kitty" the process fails to find a label in the suffix tree that matched the pattern and conclude that no occurrences exist.

The suffix link for a node v with path $x.\alpha$ is usually denoted as $SLink(v)$. The suffix links are defined for nodes and exist from node v to v', path label of v' is α. In Figure 2.1 there are four suffix links represented as arrows between internal nodes. For example the node with path "issi" has a suffix link leading to the node with path "ssi". An important case is the suffix links on the leaves, they are defined as ψ and can be computed in the compressed suffix array which are explained in section 2.1.2. The weiner link is the opposite operation of suffix link as it points in the reverse direction. Likewise in this dissertation the LF mapping is referred as the reverse of ψ because it will only exist for the leaves.

Two important concepts are $SDep$ and $TDep$, they are resp string depth and tree depth of a node. For example in the suffix tree of "mississippi" the node with path "issi" has $SDep$ 4 because that is the string size of the path. The $TDep$ of the same node is 2, because it is the second node from the root. The LCA is the lowest common ancestor between two nodes. In the suffix tree of "mississippi" the LCA of the node with path "sissippi$" and the node node with path "ssi" is the node with path "s" which corresponds to the largest common prefix between the two strings. The operation $LETTER(v, i)$ refers to the i-th letter in the path label of v. The operation $LAQ_T(v, d)$, tree level ancestor querie of node v and height d, returns the highest ancestor of node v that has a tree depth smaller or equal to d. The $LAQ_S(v, d)$ is similar but uses the string depth instead of tree depth.

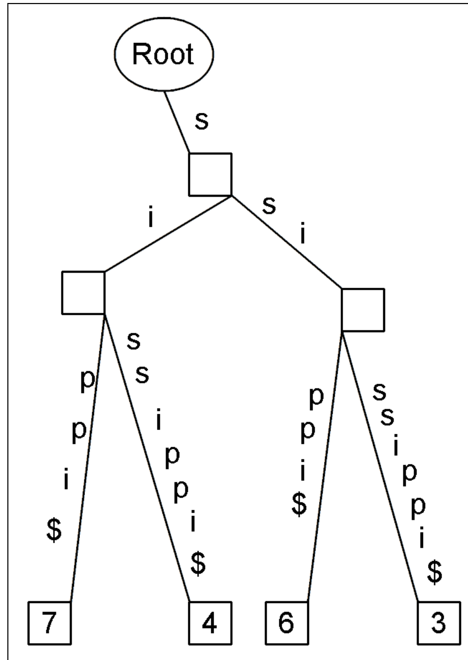


Figure 2.2: A sub-tree of the suffix tree for string "mississippi". The LCA of nodes 3 and 5 is node 1. For nodes 2 and 1 the LCA is node 1. The $SDep$ and $TDep$ of node 5 is resp 3 and 2. While $SDep$ and $TDep$ for node 4 is resp 9 and 3.

2.1.2 Suffix Array

Suffix arrays were defined in 1990 by Manber & Myers[16]. It is a data structure that contains the suffixes of the text that and occupies less space than a suffix tree. A suffix arrays for a text T, of size n, is an array SA of integers in the range 1 to n containing the start location of the lexicographically sorted suffixes of T. Every position in SA contains a suffix and $SA[1]$ stores the lexically smallest suffix in T. The suffixes grow lexically at every consecutive position of $SA[i]$ to $SA[i+1]$.

The suffix array has no information about tue string depth or tree structure. Therefore suffix arrays can be stored in very little space, i.e. an n sized array with n computer words. To find a pattern within the text a binary search is computed on the array and get the result in $O(m \log n)$ time. To report all

the α occurrences it uses $O(m \log n + \alpha)$ time [13].

	SA	*LCP*	Suffixes
1	12	0	$
2	11	0	i$
3	8	1	ippi$
4	5	1	issippi$
5	2	4	ississippi$
6	1	0	mississippi$
7	10	0	pi$
8	9	1	ppi$
9	7	0	sippi$
10	4	2	sissippi$
11	6	1	ssippi$
12	3	3	ssissippi$

1	2	3	4	5	6	7	8	9	10	11	12
M	I	S	S	I	S	S	I	P	P	I	$

Figure 2.3: Suffix array of the text "mississippi". The figure shows the index positions (leftmost), the SA values(second column) and the list of suffixes. At the right and horizontally is the text "mississippi$" with the index positions.

A important operation that can be implemented with suffix arrays is the longest common prefix of a interval of suffixes, LCP. The longest common prefix of two suffixes in the tree is the common path the two suffixes share in the top down tree traversal. For example to determine the LCP of the suffix "ississippi" and the suffix "issippi", see Figure 2.1. Start at the root then follow the path with "i", then choose the path with "ssi", at this point the suffixes path diverge and the pattern "issi" is determined as the longest common prefix. Note that a node in the suffix tree corresponds to an interval in the suffix array, therefore finding the LCP of such an interval corresponds to computing the $SDep$ of that node. The structure used in suffix arrays for this operation is the LCP table, it stores for each index the length of the LCP between the current index position and the previous position, i.e. $LCP[i] = \text{Longest Common Prefix}\{SA[i], SA[i-1]\}$.

This table is used to compute the LCP of a larger interval[1]. For a sequence of integers, as the LCP table, the range minimum querie, RMQ, uses two index positions (i, j) to return the index of the smallest integer between i

and j. For example compute the $SDep$ of the sub-tree in Figure 2.2. The v_l and v_r are respectively 9 and 12, the smallest integer from index 9 to index 12 in the LCP table computing $RMQ(v_l{+}1,\ v_r,\ LCP) = 1$. Therefore the $SDep$ of the sub-tree is 1, the LCP of the suffix array index positions 9 to 12 is "s". In figure 2.3 the positions 9 through 12 correspond to suffixes 7, 4, 6 and 3, therefore the common prefix of the suffixes in this sub-tree is "s".

A form of compacting the suffix array was researched in 2000 by Mäkinen[14]. In parallel a concept for compression of a suffix array was found in 2000 by Grossi & Vitter[9]. These two ideas of compact and compressed suffix arrays represent notable advances towards space reduction. The compact suffix array uses the self repetitions in the SA to store it in less space. It was later proved by Mäkinen and Navarro[13] that the size of the compact suffix array is related to the text entropy and therefore it is a compressed index.

The compressed suffix array by Grossi & Vitter is based on the idea of allowing access to some position in SA without representing the whole SA array. The algorithm uses the notion of function ψ which is the suffix link operation for leaves to decompose the SA.

Compressed suffix arrays can normally replace the text, i.e. they become a self indexes, i.e. it is possible to extract a part of the text of size l from the compressed suffix array, this is the $Letter$ operation. Mäkinen et al.[13] describes a compressed compact suffix array that finds all occurrences of a pattern in $O((m + \alpha) \log n$ time. Compressed suffix array can also compute $Locate(i)$, i.e. the value of $SA[i]$.

For compressed suffix arrays that support the LF operation, such as the FMIndex, it is possible to compute the $WeinerLink(v, s)$ for a string s. This operation is supported by some suffix arrays and returns the suffix tree node with path $s.v[0..]$.

2.2 Static Compressed Indexes

The entropy of a string, described in section 2.2.1, is an important concept to understand the space requirements of compressed data structures. Section 2.2.2 explains the *Rank* and *Select* operations, which are important subroutines in state of the art compressed indexes. Section 2.2.3 provides a simplified description of the Full-Text in minute space (FM-)index. We also refer to the FM-Indexes as CSA's since they provide the functionality of compressed suffix arrays, even though we do not describe other compressed suffix arrays.

2.2.1 Entropy

The compressibility of a text is measured by its entropy. The k-th order entropy of a text, represented as H_k, is the average number of bits needed to encode a symbol after seeing the k previous ones. The 0-Th order entropy, represented as H_0, is the weakest entropy because it will not look for repetitions, only for symbols frequencies. H_k is the strongest and in fact it is the application of H_0 to smaller contexts.

The k-th order entropy for finite text was defined by Manzini [17] in 2001. $\mathrm{T}_{1,n}$ is a text of size n, σ is the alphabet size and n_c is the number of occurrences of symbol c in T. The 0-Th order entropy of T is defined as[10]:

$H_0(T) = \sum_{c\in\sigma}(\frac{n_c}{n}\log\frac{n}{n_c})$

Symbols that do not occur in T are not used in the sum. Then sum for each symbol c, every occurrence of c in the text,n_c, and multiply the size in bits, $\log\frac{n}{n_c}$, used to represent each c. T^s is the sub-sequence of T formed by all the characters that occur followed by the context s. The k-th order entropy of T is defined by Manzini et. al[10]:

$H_k(T) = \sum_{c \in \sigma^k} (\frac{|T^s|}{n} H_0(T^s))$

For example it is showed how to compute the first order entropy of "mississippi". To achieve a greater compression using the first letter that precedes each symbol. For "i", "m", "p" and "s" compute the strings, resp, "mssp", "i", "pi" and "sisi". Then encode each of these strings with the 0-th order entropy and obtain a entropy of 0,9.

$H_1(T) = \frac{4}{11} H_0("mssp") + \frac{1}{11} H_0("i") + \frac{2}{11} H_0("pi") + \frac{4}{11} H_0("sisi")$

Given a text and a pattern, full text indexes are commonly used for three operations. Detecting if the pattern occurs at any position of the text, computing positions of the pattern in the text and retrieving the text. Our goal is to obtain smaller indexes while maintaining optimal or near optimal speed. The main achievement, in this area, is an index that occupies space close to the entropy of the text and operations such as insert, delete and consult close to $O(1)$.

2.2.2 Rank Select

Suffix arrays and FMindexes need two special operations called *Rank* and *Select* over a sequence of symbols. Suffix arrays are a element of modern compressed suffix trees and these operations are essential for performance. We will start by the case when the alphabet is binary and then extend *Rank* to arbitrary alphabets. They are simpler to understand than other solutions and offer constant time while using space near the k-th order entropy of the array.

Given an array of bits, bitmap A, and a position i, *Rank* of "1" over A up to i is, $Rank_1(A, i)$, the number of "1"'s from A start until position i. Notice that for bitmaps $Rank_0(A, i) = i - Rank_1(A, i)$. We assume that the positions start at 1 and for $Rank_1(A, i)$ the caracter at position i also adds to rank.

For example rank of position 5 and caracter "1" is $Rank_1(0110101, 5) = 3$. For caracter 0 and position 5 use $Rank_0(0110101, 5) = 5$ - $Rank_1(0110101, 5)$ which means there are 2 caracters "0" up to position 5. In this section we explain how it is possible to compute *Rank* in constant time.

Likewise the dual operation $Select_1(A, j)$, returns the position in A of the j-th occurrence of 1. *Select* must be implemented for both "1" and "0" as there is no way to obtain one from the other. For example to select the third caracter "1" is $Select_1(0110101, 3) = 3$ which returns 5.

The complete representation of binary sequences presented here was proposed by Raman [21], it solves the *Rank* and *Select* problem in $O(1)$ time and $nH_0 + o(n)$ bits. The representation is based on a numbering scheme. The sequence is divided into several chunks of equal size. Each piece is represented by a pair (c, i) where c is the number of bits set to 1 in the chunk and i is the identifier to that particular chunk among all chunks with c "1"-bits. The number of bits set to "1", c, is also refered to as the class of the chunk. Notice that this grouping will allow shorter identifiers for pieces with assimetrical number of "1"'s and "0"'s and hence obtaining 0-th order entropy.

The idea is to divide the sequence A into superblocks S_i of size $(\log^2 n)$ bits. Each superblock is divided into $2 \log n$ blocks B_i of size $\frac{\log n}{2}$ bits and each block belongs to a class c. Notice that for every class there are several possible combinations of bits. Each class has a table with all rank answers for its possible bit combinations.

A block is described with the number of bits set to 1(the class number) and its position within the class table. A superblock contains a pointer to all its blocks and the answer to rank at the start of the superblock. It also stores the relative answer to *Rank* of each block relative to the start of the superblock.

This structure is enough to answer *Rank* in constant time. Consider for example $Rank_1(A, i)$. First we calculate the superblock S_j to which i belongs.

Then the block number B_k within S_j and using the class number c and the index position l we consult the class table of c and position l. The *Rank* will be calculated adding the superblock *Rank* plus the relative *Rank* of the block plus the *Rank* of the relative position of i within the class table.

Combinations of class 2 with 4 bits
0011
0101
1001
0110
1010
1100

index \ positions	1	2	3	4
1	0	0	1	2
2	0	1	1	2
3	1	1	1	2
4	0	1	2	2
5	1	1	2	2
6	1	2	2	2

Figure 2.4: To the left is a table with the possible combinations of size 4 and class 2. The right table has all *Rank* answers for each position of blocks corresponding to the 6 indexes.

We show table G_c for $c = 2$ with blocks of size 4. The $c = 2$ means that all blocks have two "1"'s and therefore all 6 possible blocks have the rank calculated for at each position. For this class and index 2 we compute rank for every position of the block "0101". $Rank_1$ of the first position is 0, then for each position, the rank is the previous position rank plus one if there is a "1" at that position in the block or zero otherwise.

For example, for a bitmap A of size 250 number of superblocks will be $4 = \lceil \frac{250}{\log^2 n} \rceil$. The extra 6 bits are due to the round up operation and are padded with zeros. There will be $16 = 2\lceil \log n \rceil$ blocks with $4 = \lceil (\log n)/2 \rceil$ bits each. To find $Rank_1(A, 78)$ compute the superblock with $86/64 = 1$ remainder 14, therefore superblock S_2. The block $14/4 = 3$ remainder 2, therefore block B_4. Retrieving the class $c = 2$ the block and the index $i = 5$ we consult the table $G_2(2, 5)$. Now $Rank_1(A, 78)$ is obtained adding the rank of S_2 plus the relative rank of B_4 plus the rank returned by table G_2.

To compute *Rank* and *Select* over arbitrary alphabets we use the wavelet tree which were developed by Grossi in 2003[8]. A balanced binary wavelet

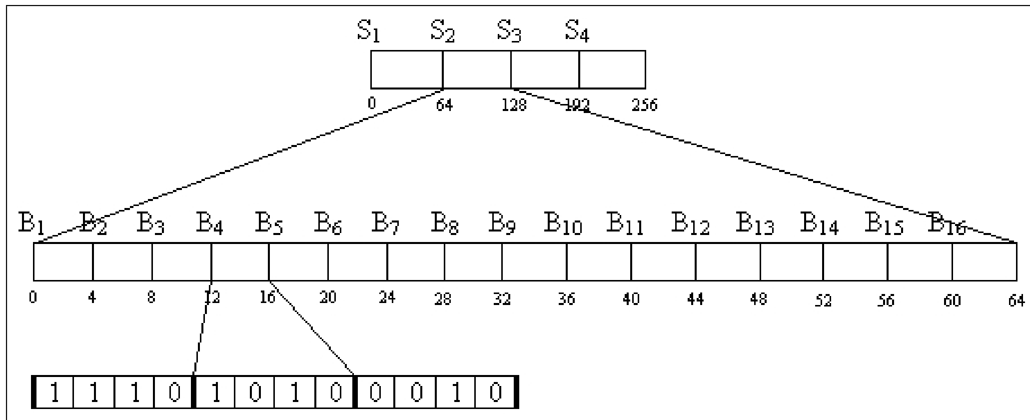


Figure 2.5: The four superblocks are on top, then the 16 blocks corresponding to the second superblock and at the bottom the binary representation of the third, fourth and fifth blocks.

tree is a tree whose leaves represent the symbols of the alfabet. Given a position i in sequence T the algorithm will travel through the nodes until it reaches a leaf and discovers the corresponding alphabet symbol. This process will allow us to compute *Rank* and *Select*.

The root is associated to the whole sequence T= $t_1 ... t_n$. The left and right child of root will each have a part of the sequence associated to a half of the alphabet. This is done by dividing the alphabet of size σ in $\sigma/2$, the left child will have a sequence W with the symbols with value smaller or equal to $\sigma/2$ and the right child larger than $\sigma/2$. A position in the left child sequence is given by concatenating all $t_i < \sigma/2$ in T. Notice that for every node v in the tree, the sequence associated to a child of v is complementary to the sequence associated to the other child. In practice the sequence is not the concatenation of caracters but the r-th bit from caracters in the order as they appear in T. The r-th bit is incremented with the tree depth, for the root r is one. This divides the alphabet at every node. As it descends it will reach a point where the alphabet is reduced to one symbol. At that point it forms a leaf. The leaf has information about the total number of caracters which it represents.

$	i	m	p	s
0	0	0	0	1
0	0	1	1	0
0	1	0	1	0

m	i	s	s	i	s	s	i	p	p	i	$
0	0	1	1	0	1	1	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	0	0
0	1	0	0	1	0	0	1	1	1	1	0

Figure 2.6: For example the text "mississippi", has a alphabet bit coding showed on the left. That bit coding generates the two dimensional bit array shown on the right.

We can use the bitmap at a node and the *Rank* and *Select* operations to map to a child node. To compute the generalized *Rank*, resp *Select* we iteratively apply *Rank*, resp *Select*, travelling through the wavelet tree. To compute *Rank* we descend from the root to a leaf. To compute select we move upwards from a leaf to the root. To get the caracter at position i, t_i, we travel the tree by going left or right depending on the value of the bit vector of the node. If the position i has 0 in node's v bitmap, W, the t_i is on the left child, else it is on the right child. If the left is chosen we should update $i \leftarrow Rank_0(W, i)$. Else if we go to the right and update $i \leftarrow Rank_1(W, i)$.

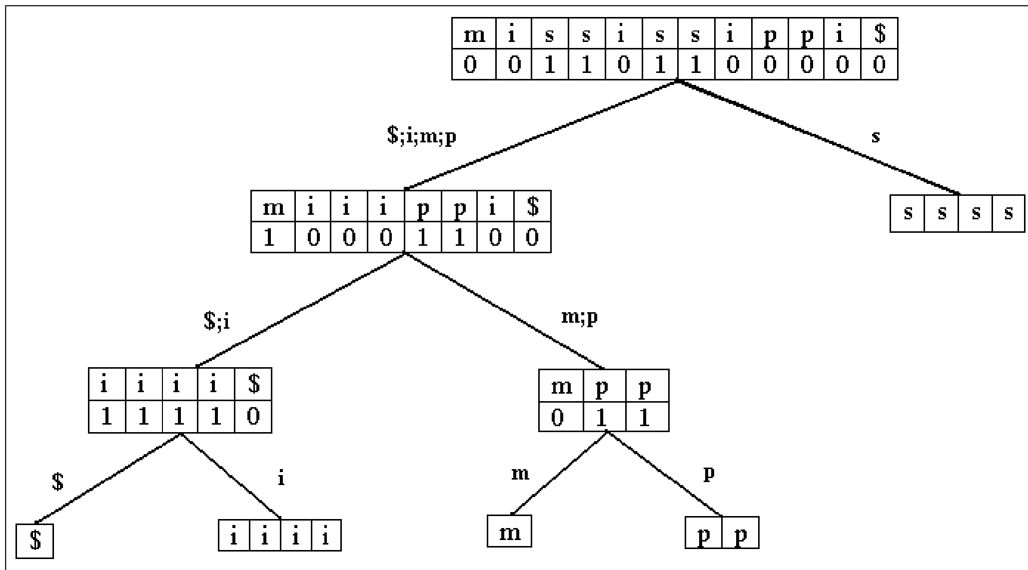


Figure 2.7: The wavelet tree for the text "mississippi".

For example we show how to compute *Rank* of caracter "p" in $position = 11$ of T. Note that the representation of "p" is "0,1,1", therefore we will compute $Rank_0$, $Rank_1$ and $Rank_1$ in succession for each node visited. The first bit in the binary representation of "p" is 1, therefore we should go to the left child and $position \leftarrow Rank_0(001101100000, 11)$ results in $position = 7$. The second bit for caracter "p" in the alphabet bitmap is "1", therefore we will go to the right child and update $position \leftarrow Rank_1(10001100, 7)$ which computes $position = 3$. Finally the third bit for caracter "p" is "1", therefore we should go to the right child and $position \leftarrow Rank_1(011, 3)$ results in $position = 2$. The result of $Rank_p(mississippi\$, 11)$ is found as we reach the leaf. There the current $position = 2$ indicates the rank of the symbol "p" at 11.

To compute $Select_c(T, position)$ the algorithm will start at the leaf of c. Since the representation of "i" is "0,0,1" we will compute $Select_1$, $Select_0$ and $Select_0$ successively. We will climb the tree up to the root. At each node v it updates $position \leftarrow Select_b(W, position)$ (b is 0 or 1 depending on the corresponding bit in the representation of "i"). For example finding the position of the second "i" in T is $Select_i(mississippi\$, 2)$. First we travel to the leaf of symbol "i", this can be done with a direct mapping from a array of the alphabet to the leaves. Since the current node is the right child of its parent $position \leftarrow Select_1(11110, 2)$. We travel to the parent and now $position = 2$. The next upward climb is from a left child and therefore $position \leftarrow Select_0(10001100, 2)$ so $position = 3$. Reaching the root again from a left child $position \leftarrow Select_0(001101100000, 3)$ computes the final position $position = 4$.

Operations are done over bitmaps and with the structure explained before, are solved in constant time. The space required for the bitmaps is not greater than the original string of "*mississippi$*". The bits were rearrenged in the tree like structure and therefore all added space is due to a structure to organize the tree.

2.2.3 FMIndex

The FMIndex is a full text index which occupies minute space. Since it compresses and respresents the text the FMIndex is a compressed self index. The strengh of this index relies on the combination of the Burrows Wheeler compression algorithm with the wavelet tree data structure.

The Burrows Wheeler Transform (BWT) is a reversible transformation of a text. A text T of size n is represented in a new text with the same caracters in a different order but normally with more sequential repetitions and therefore easier to compress. The stages of the BWT are explained in three steps.

1. A new caracter with lexicographical value smaller than any other in T is append at the end. Let it be "$".
2. Build a matrix M such that each row is cyclic shift of the string T$, then sort the rows in lexicographic order.
3. The text L is formed by the last column of M and is the result of the BWT.

The Figure 2.8 ilustrates the Burrows-Wheeler transformation of a text. The text "mississippi" becomes "mississippi$" after step 1. Using cyclic shifts the matrix on the left side of figure is generated. Sorting those rows by lexicographic order creates the matrix on the right. L is a permutation of the original text T and so are all other columns in M. Column F is a special case because it is the lexicopraphically ordered caracters of T$. The relation between matrix M and suffix arrays becomes evidente if we notice that sorting the rows of M is sorting the suffixes of T$.

Operation $C(c)$ will report the number of occurrences of caracters lexicographically smaller than c. $Occ(c, k)$ will report all occurrences of caracter c before position k in column L. Notice that for any row in M all caracters

mississippi$
ississippi$m
ssissippi$mi
sissippi$mis
issippi$miss
ssippi$missi
sippi$missis
ippi$mississ
ppi$mississi
pi$mississip
i$mississipp
$mississippi

F		L
$	mississipp	i
i	$mississip	p
i	ppi$missis	s
i	ssippi$mis	s
i	ssissippi$	m
m	ississippi	$
p	i$mississi	p
p	pi$mississ	i
s	ippi$missi	s
s	issippi$mi	s
s	sippi$miss	i
s	sissippi$m	i

Figure 2.8: To the left of the figure is the matrix with successive cyclic shifts of text "mississippi$". On the right of the figure is the matrix after lexicographical row ordering with the last and first collumns in boxes.

in L precede the caracters in F. A important function is the last to first column mapping (LF). LF describes a way to obtain the caracter position in F corresponding to a given position in L with functions C and Occ [19].

$$LF(i) = C(L[i]) + Occ(L[i], i)$$

For example the "p" in mississippi is at position 7 of L. We wish to know $LF(7)$ so we calculate $C(”p”) = 6$ and $Occ(”p”, 7) = 2$. Finally $LF(7) = 6+2$ shows that the result of moving caracter "p" from L to F results in row 8. The next operation is fundamental to understand how LF mapping generates and returns the string T in reverse order. If T is the ith caracter in L then the caracter at position $k - 1$ is at the end of the row returned by LF(i).

$$T[k - 1] = L[LF(i)]$$

For example suppose we want the caracter before the "s" in position 3 of the text "mississippi". Notice that in L the first "s" in the text is represented by position 10. $LF(10) = 8 + 4$, the LF mapping indicates row 12. The last

caracter in row 12 is the first "i" in the text. Iterating over $L[LF(i)]$ the FMIndex returns the full text from the compressed representation of L.

Based on this idea Ferragina and Manzini [4] proposed the backward search procedure. The backward search finds a pattern $P[1,n]$ within the text finding caracters of the pattern from right to left. This is usefull for the FMIndex since the LF mapping returns caracters right to left. In matrix M notice that all answers to a particular pattern are lexicographically similar and are put in sequential rows. These rows are delimited by the *sp* and *ep* indexes, *sp* indicates (in lexicographic order of M) the first row with the pattern and *ep* the last row.

At the start of the search *sp* is the position of the first lexicographic occurrence of the last caracter of pattern. Since ep points to the last row in the sequence it should be the row before the next lexicographical caracter. Therefore $sp = C(P[m]) + 1$ and $ep = C(P[m] + 1)$. The backward search starts with the character in position $i = m$. The algorithm will use a caracter in position $P[i]$ at each step until it reaches the start of pattern and $i = 1$ and returns the interval $[sp, ep]$.

The cicle that moves $P[i]$ from $i = m$ down to $i = 1$ and updates *sp* and *ep* uses the number of occurrences in L of caracter $P[i]$ up to position *sp* or *ep*. The function for *sp* is $sp = C(c) + Occ(c, sp - 1) + 1$ and for *ep* is ep= $C(c) + Occ(ep)$.

For example we will search for pattern "ss", see Figure 2.9 within "mississippi" with backward search. First $c = P[m]$ so $c = "s"$, hence $sp = C("s") + 1$ so $sp = 9$, $ep = C("s" + 1)$ so $ep = 12$. Now we proceed to the begining of the pattern as $c = [m - 1]$ so $c = "s"$. We update sp and ep, $sp = C(c) + Occ(c, 9 - 1) + 1$ so $sp = 8 + 2 + 1$, $ep = C(c) + Occ(c, 12)$ so $ep = 8 + 4$. As c is at the beggining of P the backward search stops and returns $[sp = 11, ep = 12]$. The interval describes the only two existing suffixes that start with pattern "ss" in "mississipi".

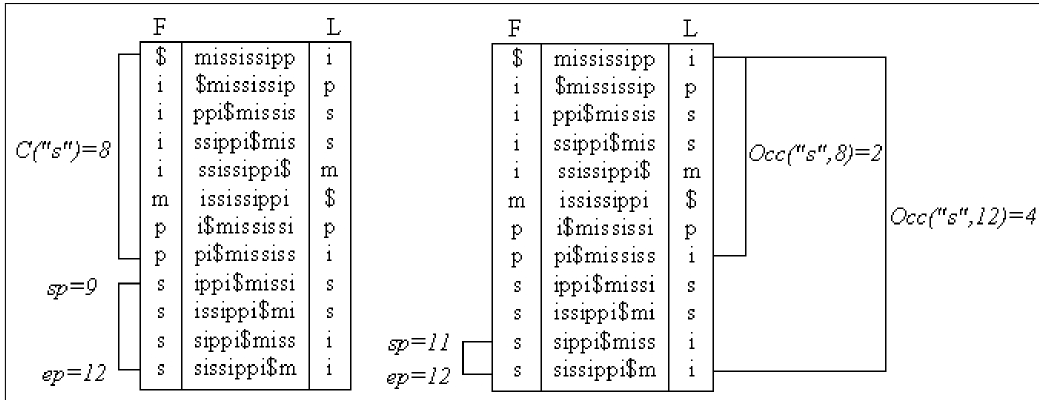


Figure 2.9: Matrix M of the Burrows Wheeler Transform with operations computed during a backward search.

The space required by FMIndex is $nH_k+o(n\log\sigma)$ bits, with $k \geq \alpha \log_\sigma \log n$. The time to count the number of occurrences is $O(m)$ and time to return l letters is $O(\sigma(l + \log^{1+\epsilon} n))$, ϵ is any constant larger than 0 [19].

2.3 Static Compressed Suffix Trees

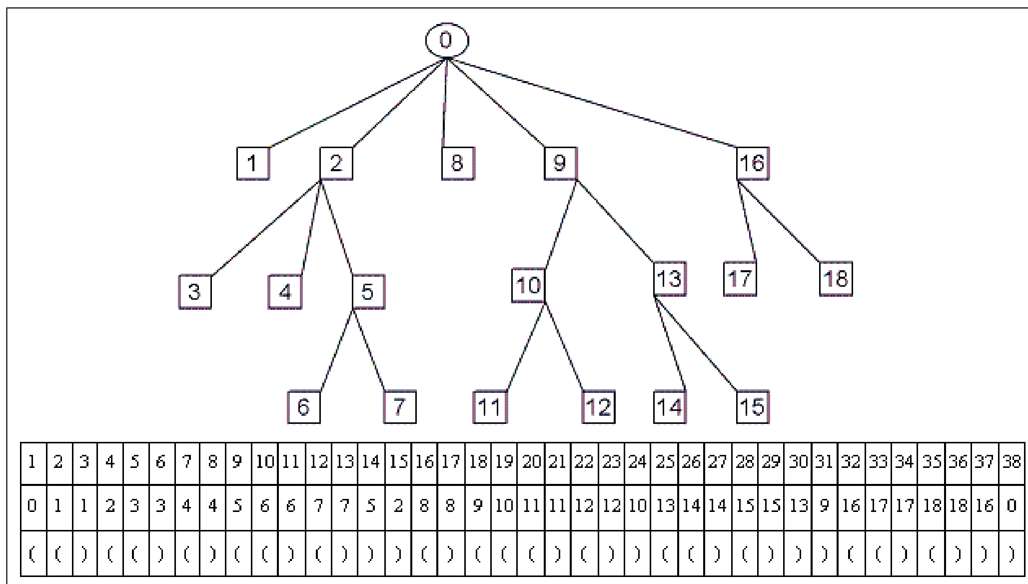
Three recent compressed and static suffix trees are discussed in this chapter. These are different approaches and studying them is important to understand why the dissertation approached this work. First we present the CST proposed by Sadakane et al [24], section 2.3.1. Section 2.3.2 presents the FCST proposed by Russo, Arlindo and Navarro [12] which is compared with the CST described in 2.3.1. The compressed suffix tree by Fischer et al [5] is presented in section 2.3.3.

2.3.1 Sadakane Compressed

Sadakane in 2007 proposed the first compressed suffix tree that uses linear space $6n + nH_k + o(n\log\sigma)$ bits [24]. Former suffix trees were represented in

$O(n \log n)$ bits which compared n with the size of a alphabet shows a huge achievement. For example DNA has a alphabet of size 4, the size of the human genome is 700 megabytes, hence in this case $\log n/\log \sigma$ is at least 14,5. A classical suffix tree for such a problem would span a impressive 40 gigabytes [10].

The problem addressed by Sadakane[24] is to remove the pointers from the representation. As for every pointer it is necessary $\log n$ bits such a result uses at least $O(n \log\ n)$. To reduce space use Sadakane replaced the commonly used tree structure with a balanced parentheses representation of the tree. For a tree with n nodes the parentheses representation uses $2n+o(n)$ bits[18]. A suffix tree with text of size n has at most n leaves, $n-1$ internal nodes for a total $2n-1$ nodes. Therefore a tree can be represented in $4n+o(n)$ bits with the parentheses notation.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	1	1	2	3	3	4	4	5	6	6	7	7	5	2	8	8	9	10
(	(	)	(	(	)	(	)	(	(	)	(	)	)	)	(	)	(	(

20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
11	11	12	12	10	13	14	14	15	15	13	9	16	17	17	18	18	16	0
(	)	(	)	)	(	(	)	(	)	)	)	(	(	)	(	)	)	)

Figure 2.10: At the top of the figure is a representation of depth first ordering of the suffix tree for "mississippi". The table at the bottom represents the order of parentheses for this tree. The first row is the index of the array, the second is the node number in the suffix tree and the third is the paretheses representation.

The parentheses tree is built with a ordered tree traversal, the first time a

node is visited add a left parentheses, then visit all the nodes in the sub-tree, after all sub-nodes are visited add a right parentheses[18]. The nodes in the tree are represented by a pair of parentheses. This representation can be stored by using a bit per parentheses. Therefore the parentheses tree representation is stored in a bitmap. This is a significant improvement provided the usual navigational operations are supported.

The total space for this suffix tree is $nH_k + 6n + o(n)$ bits. The nH_k accounts for a compressed suffix array which is necessary to compute $SLink$ and read edge-labels, the remaining space is for auxiliary data structures such as the parentheses representation and a range minimum query data structure. Interestingly in a note for future work Sadakane referred to the $6n$ space problem in the structure. That $6n$ problem was addressed by Russo, Arlindo and Navarro[12].

2.3.2 FCST

The fully compressed suffix tree proposed by Russo, Arlindo and Navarro [12] uses the less space to represent a suffix tree while loosing some speed. There is an implementation by Russo that achieves optimal compressed space for the first time.

The FCST is composed of two data structures. A sampled suffix tree S and a compressed suffix array CSA. The sampled suffix tree plays the same role in the FCST as Sadakane's parentheses tree in the CST. The reason why the sampling is used instead of storing all the nodes is that suffix trees are self-similar acording to the following lemma:

Lemma 1 $SLink(LCA(v, v')) = LCA(SLink(v), SLink(v'))$

To understand this lemma assume that the nodes v and v' have a path label $X.\alpha.Y.\beta$ resp $X.\alpha.Z.\beta$. Both have equal prefixes $X.\alpha$ therefore the LCA of

both nodes is reached with $X.\alpha$, $LCA(v, v') = X.\alpha$. The SLink is applied to v, v' and $LCA(v, v')$and obtain resp $\alpha.Y\beta$, $\alpha.Z.\beta$ and α. Notice that $LCA(\alpha.Y\beta,\ \alpha.Z.\beta) = \alpha$ and therefore
$SLink(LCA(v, v')) = LCA(SLink(v), SLink(v'))$.

The sampled tree explores this similarity, it is necessary that every node is, in some sense, close enough to a sampled node. This means that if the computation starts at a node v and follow suffix links successively, i.e. apply $SLink$ on the result of $SLink$ of v several times, in a maximum of δ steps the computation reaches a node sampled in the tree. This is an important property for a δ sampled tree. Also because the $SLink$ of the root is a special case that has no result, the root must be sampled. The nodes picked for sampling are those that $SDep(v) \equiv_{\delta/2} 0$ such that exists a node v and a string $|T'| \geq \delta/2$ and $v' = LF(T', v)$, i.e. the remander of $SDep(v)$ and $\delta/2$ is 0.

The sampled suffix tree allows the reduction of space usage on the total tree. A suffix tree with $2n$ nodes with a implementation based on pointers uses $O(n \log n)$ bits. A sampled tree requires only $O(\frac{n}{\delta} \log n)$ bits, to use only $o(n \log \delta)$ bits of space and in this dissertation $\delta = \lceil (\log_\sigma \log n) \log n \rceil$.

Sadakane uses a CSA [9] that requires space of $\frac{1}{\epsilon} n H_0 + O(n \log \log \sigma)$ bits. In the FCST the CSA is an FM-index[7]. It requires $nH_k + o(n \log \sigma)$ bits, with $k \geq \alpha \log_\sigma \log n$ and constant $0 < \sigma < 1$. Note that although Sadakane´s CSA is faster it would use more space than is desirable.

It is important to map the information from the sampled tree to the CSA and vice-versa. For this goal the operations in the sampled suffix tree include $LCSA(v, v'), LSA(v)$ and $REDUCE(v)$. These operations are explained with detail in section 2.4.3. To obtain the interval over the CSA that corresponds to a sampled node it is enough to store a pair of integers in the sampled tree. In the other direction, however, a injective mapping does not exist, instead it is used the lowest sampled ancestor, LSA.

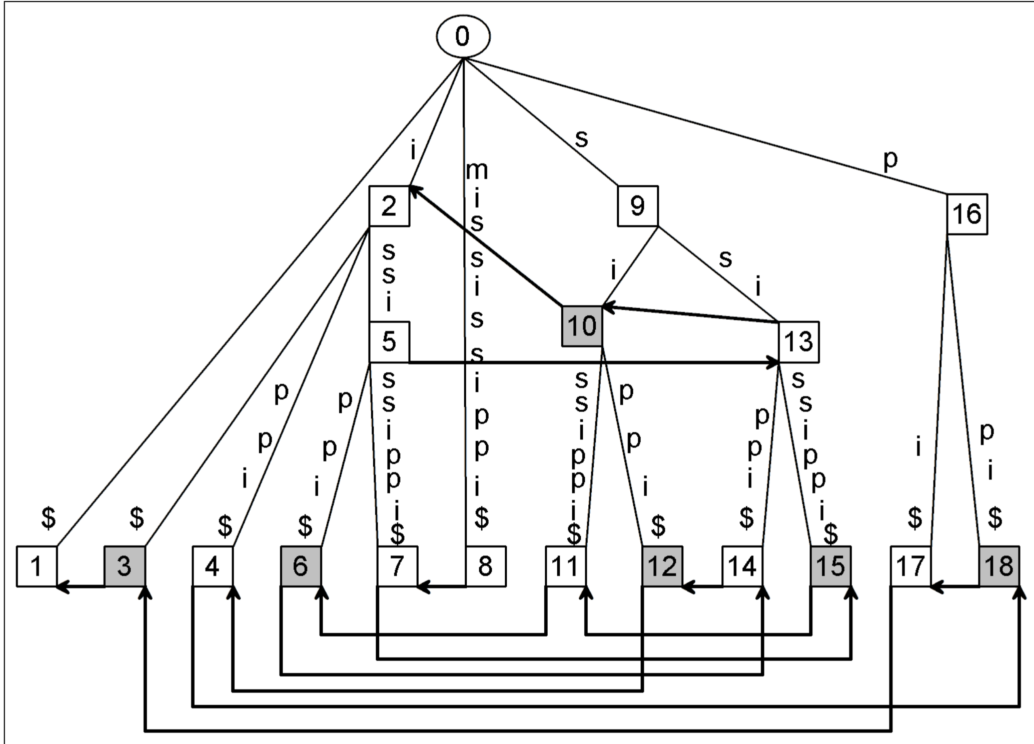


Figure 2.11: The figure shows the suffix tree of the word mississippi. Nodes filled in gray outline are sampled due to the number of suffix links and to the string depth. The sampling chosen is 4 so nodes are sampled if $SDep$ is multiple of 2, and if exists a suffix link chain of length multiple of two. The thick arrows between leafs are suffix links.

It is now explained how the FCST computes its basic funcion. If v and v' are nodes and $SLink^r(LCA(v, v')) = ROOT$, $d = min(\delta, r + 1)$:

Lemma 2

$$SDep(LCA(v, v')) = max_{0 \leq i \leq d}\{i + SDep(LCSA((SLink^i(v), SLink^i(v')))\}.$$

The operation $LCSA$ is supported in constant time for leaves. $SDep$ is only applied to sample nodes so its information is stored in the sampled nodes. The other operation needed to implement the previous lemma is $SLink$.

Sadakane proved that $SLink(v) = LCA(\psi(vl), \psi(vr))$ with $v \neq ROOT$,

where v_l and v_r are the left adn right extremes of the interval that represents v [3]. This is extended to $SLink^i(v) = LCA(\psi^i(vl), \psi^i(vr))$. Remember that all ψ answers are computadle in constant time. A lemma proded by Russo et al. concludes:

Lemma 3

$LCA(v, v') = LCA(min\{vl, vl'\}, max\{vr, vr'\})$

From the previous lemma, the definition of ψ and lemma 2 concludes:

$SDep(LCA(v, v')) =$

$max_{0 \leq i \leq d}\{i + SDep(LCSA((\psi^i(min\{vl, vl'\}), \psi^i(max\{vr, vr'\})))\}$

Therefore $SLink$ is not necessary to compute LCA. Hence it is also concluded, using the i from lemma 2 :

Lemma 4:$LCA(v, v') = LF(v[0..i-1], LCSA(SLink^i(v), SLink^i(v'))$

Therefore it can be solved with the same properties that solved lemma 3.

$LCA(v, v') = LF(v[0..i-1], LCSA(\psi^i(min\{vl, vl'\}), \psi^i(min\{vr, vr'\}))$

The operation $LETTER$ in FCST is solved with the following:
$LETTER(v, i) = SLink^i(v)[0] = \psi^i(v_l)[0]$.

Operation $Parent(v)$ returns the smallest between $LCA(v_l - 1, v_r$ and $LCA(v_l, v_r + 1)$. This works because suffix trees are compact. Child of a node is computed directly over the CSA. The time complexity of all these operations is shown in Table 2.1.

Table 2.1: The table shows time and space complexities for Sadakane static CST and Russo et al. FCST. The first row has space use and the remaining rows are time complexities. In the left collumn are operations, the middle column has time complexities for Sadakane static CST and the right column has FCST time complexities.

	Sadakane CST	Russo et al. FCST
Space in bits	$nH_k + \mathbf{6n} + o(n \log \sigma)$	$nH_k + o(n \log \sigma)$
SDep	$\log_\sigma(\log n) \log n$	$\log_\sigma(\log n) \log n$
Count/Ancestor	1	1
Parent	1	$\log_\sigma(\log n) \log n$
SLink	1	$\log_\sigma(\log n) \log n$
SLink^i	$\log_\sigma(\log n) \log n$	$\log_\sigma(\log n) \log n$
LETTER(v, i)	$\log_\sigma(\log n) \log n$	$\log_\sigma(\log n) \log n$
LCA	1	$\log_\sigma(\log n) \log n$
Child	$\log(\log n) \log n$	$(\log(\log n))^2 \log_\sigma$
TDep	1	$((\log_\sigma(\log n)) \log n)^2$
WeinerLink	1	1

2.3.3 An(Other) entropy-bounded compressed suffix tree

Fischer et al. [5] presented a compressed suffix tree with sub-logarithmic time for operations and consuming less space than Sadakane's compressed suffix tree, detailed in section 2.3.1. They used two ideas to achieve theses results, first reducing space used for LCP information and secondly discarding the suffix tree structure using the suffix array intervals to represent tree nodes and using the LCP information to navigate the tree.

Sadakane's CST space complexities has a term with $6n$ bits of space, Fischer et al. replaces this term with a smaller factor. The $6n$ term contains information for LCP queries and the suffix tree structure using the parentheses representation. Notice that $i + LCP[i]$ in consecutive positions of the text is non decreasing [5]. Sadakane et al. defines table Hgt to store only the differences in unary and thus reduce space to $2n$. Fischer et al. replaced Hgt by U, observing that the table U has the number of 1-bit runs bounded to the

number of runs in ψ. Encoding this information with additional structures and reducing U they obtain $nH_k \times (2 \log \frac{1}{H_k} + \frac{1}{\epsilon} + O(1))$ bits to store the LCP information, where ϵ is a constant $0<\epsilon<1$.

They define the next smaller querie, NSV and the previous smaller querie PSV. For a sequence, I, of integers the NSV of position i returns j such that $j > i$, $I[j] < I[i]$ and no position between i and j has a smaller integer in I. The PSV is identical to NSV with $j < i$. Remember the RMQ is two index positions (i, j) and I to return the index of the smallest integer between i and j $argmin_{i \leq k \leq j} I[k]$.

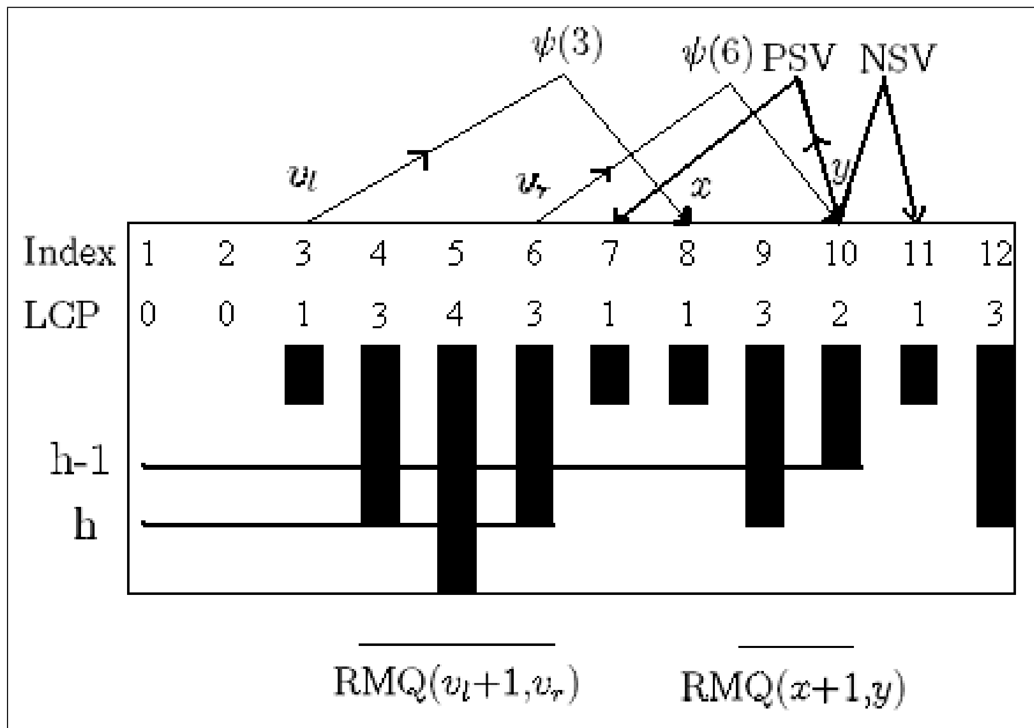


Figure 2.12: The figure shows a LCP table and the operations required to perform $SLink$(3,6) with RMQ and PSV, NSV.

The RMQ together with ψ, NSQ and PSQ is enough to navigate the suffix tree. For an example see Figure 2.12, given a node $v(v_l, vr)$ the suffix link is computation is shown. First notice that RMQ in the interval $[v_l, vr]$ will return the smallest LCP value in that range, let it be h. Next apply the ψ

function, available in the CSA, to v_l and vr] and obtain $[x, y]$. Now find k with a LCP(k) = h-1 using RMQ in the interval $[x, y]$, finally to find the node's right and left limits $[v'_l, v'r]$ apply PSV to x and NSV to y.

They achieve a total space of $nH_k \times (2 \log \frac{1}{H_k} + \frac{1}{\epsilon} + O(1)) + o(n \log \sigma)$ bits of space while FCST uses $nH_k + o(n \log \sigma)$ bits. The extra factor tends to zero if nH_k is close to zero, however it is not common for the entropy to be close to zero. This solution presents a middle point between FCST and Sadakane's CST in both speed and space. Moreover this solution is static, i.e. it cannot be updated whenever the text changes.

2.4 Dynamic Compressed Indexes

Dynamic FCST's use dynamic bit sequence as a auxiliary structures. One such dynamic bit sequence, proposed by Makinen and Navarro [15], is presented in section 2.4.1. Section 2.4.2 presents the dynamic CST by Chan et. al [3], which is a alternative to the dynamic FCST proposed by by Russo, Arlindo and Navarro [22]. The dynamic FCST is described in section 2.4.3 which ends with the comparison of the dynamic FCST and the dynamic CST.

2.4.1 Dynamic Rank and Select

A structure is dynamic if it supports the insertion and removal of text from a collection. Makinen and Navarro obtained a dynamic FMIndex by first presenting a dynamic structure for *Rank*, *Select* and using a wavelet tree over the BWT[15]. They show how to achieve $nH_0 + o(n)$ bits of space and $O(\log(n))$ worst case for *Rank*, *Select*, insert and delete.

To solve *Rank* and *Select* the approach presented is similar to the one discussed previously as a static solution. The structure consists of superblocks

and blocks, while the superblocks are in the leaves of a tree the blocks are arranged in the superblocks.

The tree used to store the superblocks is a binary tree, a red black tree, with additional data in the nodes to compute operations such as *Rank* and *Select* while traversing the tree. Consider a balanced binary tree on a bit vector $A = a_1...a_n$, the left most leaf contains bits $a_1 a_2 ... a_{\log n}$, the second left leaf $a_{\log n+1} + a_{\log n+2} ... a_{2(\log n+1)}$ through to the last leaf. Each node v contains counters $p(v)$ and $r(v)$ resp counting the number of positions stored and the number of bits set to "1" in the subtree v. This tree with $\log(n)$ size pointers and counters, requires $O(n)$ bits of space[15].

The superblocks contain compacted bit-sequences but we will explain the operations as if they are not compacted. To compute $Rank(A, i)$ we use the tree to find the leaf with position i. We use a variable *rankResult* that is initially set with value 0. We travel the tree downwards to the leaves, we use the value of $p(left(v))$, if it is smaller than i we go to the left subtree of v. Otherwise we descend to the right node, in which case i and *rankResult* must be updated as $i = i - p(left(v))$ and $rankResult = rankResult + r(left(v))$. The desired leaf is reached in $O(\log(n))$ time and $Rank(A, i)$ uses extra $O(\log(n))$ time to scan the bit sequence of the corresponding leaf. When the leaf is reached the result of scanning the bit sequence for *Rank* is added to *rankResult*. Select(A,i) is similar but we switch the $r(v)$ and $p(v)$ roles.

As an example we will compute $Rank_1(A, 10)$, see Figure 2.13. Since $8 = p(left(root))$ is smaller than 10 we descend to the right child of the root and update $rankResult = r(left(root))$ and $i = 2 = 10 - p(left(root))$. The left child of the current node has $p = 4$ which is larger than i, therefore we descent to the left child. The current node is a leaf and we scan the bitmap to find the local rank of position $i = 2$. The local rank plus *rankResult* gives a total rank of 4.

Consider that the leaves do not contain superblocks but simple bitmaps. We will now explain insertions and deletions for that situation and later detail the

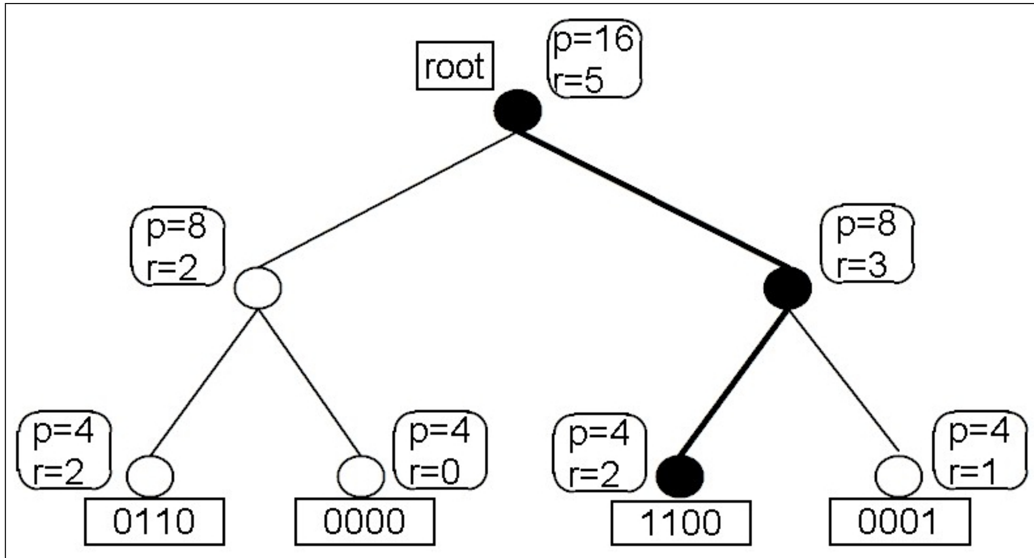


Figure 2.13: The figure shows a binary tree with p and r values at the nodes and bitmaps at the leaves. The path in bold is used to compute *Rank* of position 10.

superblock operations. To find where to insert or delete a bit we navigate the red black tree down to the leaf, like in *Rank*, and update the bit-sequence by performing the necessary changes. The next operation is updating the $p(v)$ and $r(v)$ functions in the path from the leaf up to the root. Eventually insert and delete will generate overflow or underflow. If we insert a bit in a leaf the block is bitwise shifted and the bit inserted. This however will make a bit fall of the end of the block which has to be inserted on the next block. The underflow problem is similar and both overflow and underflow are discussed further on. After these split and merge operations the tree must be updated with new values for $p(v)$ and $r(v)$ as well as rebalancing the tree. If the bitmaps are compacted the underflow and overflow are handled differently.

This structure with bitmaps on the leaves uses $O(n)$ bits, applying the superblocks hierarchy reduces it to $n + o(n)$ bits. In this case leaves contain superblocks, i.e. they contain about $\log^2 n$ bits. The superblocks are structured to support dynamic operations so it is different from the static case explained earlier. Each superblock has $2(\log n)$ blocks and each block has

$\frac{\log n}{2}$ bits. The universal table R in the superblock computes the *Rank* values for each block, therefore *Rank* in the superblock is computed with the help of table R.

To compute *Rank* in a superblock we scan through the blocks and table R adding each *Rank* until we are at the block with the query position. The bits within the block are scanned until we reach the desired position and *Rank* is computed in $O(\log n)$ time. *Select* is similar to *Rank* because the universal table indicates the number of bits within a block. Therefore we travel the blocks to the block that contains the position querie. At that block we scan the bits and retrieve *Select*.

In the structure presented by Veli and Navarro the block and superblock have no wasted bits, therefore whenever a bit is inserted or deleted a overflow or underflow problem arises. Overflow propagation to the adjacent leaves may not be fixed with a constant number of block splits. We will now discuss the solution to the underflow problem.

In this structure whenever a bit is inserted a bit shift occurs. A block were a bit is inserted will have a block overflow due to the extra bit that needs to be inserted in the next block. This propagates through all the blocks in the superblock and eventually reaches the end of the superblock causing a superblock overflow. To limit the propagation of overflow we will add a partial superblock at every $O(\log n)$ superblocks. This superblock uses $O(\log n) \log n$ bits but might be partially full. It also permits a underfilled block at it´s end (underfilled because it has less than $\frac{\log n}{2}$ bits. The partial block needs to be managed with care. It must be padded with dummy bits to obtain a representation in R and care is needed to notice its real length during operations. Partial superblocks can waste $O(n/O(\log n))$ bits, but ensure that we never traverse more than $O(\log n)$ superblocks in the overflow propagation, a density of partial superblocks with at least $O(\log n)$ distance among them. First we check if a partial superblock exists in the next $2O(\log n)$ superblocks. If we find one, we carry out the propagation

until we reach it. If there is no partial superblock we propagate through $O(\log n)$ superblocks and create a partial superblock at this location. In both situations we have to travel $O(\log n)$ superblocks and guarantee that every superblocks is at least $O(\log n)$ distant from other superblocks. However the partial superblocks may overflow, in which case they are no longer partial. We create a new partial superblock after it and the partial superblock that overflows becomes a normal superblock. When a partial superblock overflows it will in some cases have a partial block at its end. They solve this by simply moving this block to the new partial superblock end. Other overflow blocks will fill the rest of the partial superblock.

Another operation is the removal of one bit that causes underflow. We ensure that the superblocks are always full. If some underflow happens in the end of the superblock, we use the next superblock and move some blocks back. This propagation is similar to overflow propagation. If we reach a partial superblock the problem is solved and propagation stops. If the search for a partial superblock exceeds $2O(\log n)$ steps we allow the underflow in the $O(\log n)$ superblock and it becomes a new partial superblock. If a partial superblock becomes empty it is removed from the tree.

Insertion and deletion of bits will require the update of $p(v)$ and $r(v)$ values from the leaf up to the root. However the propagation problem affects only $O(\log n)$ superblocks. When we find the leaf that we wish to create or delete, the red-black tree uses constant time to rebalance, this will add $O(\log n)$ time per insertion or deletion. When propagating the coloring of the red-black tree and updating the $p(v)$ and $r(v)$ values the $O(\log n)$ blocks are contiguous, therefore the number of ancestors does not exceed $O(\log n) + O(\log n) = O(\log n)$. The overall work needed for this maintenance is $O(\log n)$.

Veli and Navarro achieve a structure that manages a dynamic bit sequence in $nH_0 + o(n)$ bits and logarithmic time for insert, delete, *Rank* and *Select*[15]. This structure is a important background to our dynamic FCST, because it supports a dynamic FMIndex in $nH_k + o(n \log \sigma)$bits.

2.4.2 Dynamic compressed suffix trees

Chan et al. proposed, in 2004, a dynamic compressed suffix tree that uses $O(n \log \sigma)$ bits of space[3]. They use a mixed version of a CSA plus a FMIndex to speed up their updates, at the time CSA and the FMIndex were used to provide complementary operations. However new versions of the FMIndex can also compute the ψ function hence replacing the CSA[3]. The structures used are named COUNT, MARK and PSI respectively related to the LF, the SA and the ψ functions. The MARK structure computes SA[i], to do this it stores some values from the SA array and determines the other values with the COUNT structure[3]. The COUNT and PSI structures are supported by an FMIndex that supports insertions and deletions of texts T' in $O(|T'| \log n)$.

Recall that $Occ(c, i)$ returns the number of occurrences of symbol "c" up to position i of the BWT. For example, a bitmap of size n for character "c" with each occurrence in the text is computable, notice that $Rank_1(i)$ over this bitmap will return $Count(c, i)$. These bitmaps can be stored in the structures presented in the previous section. To compute MARK they use two RedBlacks that store values explicitly. Adding all the red blacks the total space is $O(n \log \sigma)$ bits. The insertion and deletion of a character from the text uses $O(\log\ n)$ time while finding a pattern of size m uses $O(m \log\ n + occ \log^2 n)$.

This approach is one of the few dynamic compressed suffix trees available and therefore is a tool to judge our own dynamic CST performance. Chan et al.[3] CST uses $O(n \log \sigma)$, however the dynamic FCST uses $nH_k + o(n \log \sigma)$, which is much smaller in general.

Dynamic Parentheses Representation

Chan et al.[3] proposed a CST in 2007, in this dissertation there is interest in the approach to the LCA problem. They proposed a way to store the topology of a suffix tree in $O(n)$ bits of space. The parentheses representation of the tree topology creates a bitmap of $2n$ bits that is processed to find matching and enclosing parentheses. This is done with two structures that complement each other and answer LCA queries. The two structures are dynamic, the first supports delete and insert in $O(\frac{\log n}{\log \log n})$ time, the second supports these operations in $O(\log n)$.

The first structure computes matching parentheses. It is a B-tree with the parentheses bitmap divided in blocks of size from $\frac{\log^2 n}{\log \log n}$ to $2\frac{\log^2 n}{\log \log n}$. The bitmap is distributed on the leaves of the tree, i.e. and concatenating the leaves in order returns the original parentheses bitmap.

The second structure determines the LCA, witch is the same as double enclosing parentheses, it is a red black tree with the parentheses bitmap divided in blocks of size from $\log n$ to $2 \log n$. The bitmap is distributed over the leaves of the tree. Concatenating the leaves in order returns the original parentheses bitmap and find the nearest enclosing parentheses using auxiliary structures in the nodes of the red black.

Matching Parentheses

The matching parentheses of a position i in the parentheses representation is found consulting position $i + 1$ and if necessary computing the nearest enclosing parentheses.

For example, the computation of the matching parentheses of two index positions in a parentheses representation of a suffix tree is shown in Figure 2.14. The index position 18+1 corresponds to a opened parentheses therefore

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
DF numbering	0	1	1	2	3	3	4	4	5	6	6	7	7	5	2	8	8	9	10
Parentheses	(	(	)	(	(	)	(	)	(	(	)	(	)	)	)	(	)	(	(
Variation				-1	-1	+1	-1	+1	-1	-1	+1	-1	+1	+1					
Maching 15				-1	0	1	0	1	0	1	2	1	2	1					
Variation																			+1
Matching 18																			1

Index	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
DF numbering	11	11	12	12	10	13	14	14	15	15	13	9	16	17	17	18	18	16	0
Parentheses	(	)	(	)	)	(	(	)	(	)	)	)	(	(	)	(	)	)	)
Variation																			
Maching 15																			
Variation	+1	-1	+1	-1	-1	+1	+1	-1	+1	-1	-1	-1							
Matching 18	2	1	2	1	0	1	2	1	2	1	0	-1							

Figure 2.14: The figure represents the computation of the matching parentheses for index position 15 and position 18 over the parentheses representation of the suffix tree for "mississippi". The first row is the index position of the bitmap. The second row is the depth first numbering of the nodes and the third is the parentheses represented by the bitmap. The fourth and fifth rows are the steps used to compute the matching open parentheses and the sixth and seventh rows are the computation of the matching closing parentheses.

search incrementing the index position to find the corresponding enclosing parentheses. For each index position visited add 1 to a counter if it is a opened parentheses and subtract 1 if it is a closed parentheses. In this example in Figure 2.14 travel from index 19 to index 31, until our counter reaches -1. Therefore the matching parentheses of 18 is 31.

The index position 15 corresponds to a closed parentheses, therefore search backwards for the corresponding enclosing open parentheses. For each index position visited add 1 to the counter if it is a closed parentheses and subtract 1 otherwise. In this example in Figure 2.14 travel from index 14 to index 4 until the counter reaches -1, therefore the matching parentheses of 15 is 4.

The structure presented by Chan et al. [3] proposes that for each node v of the B-tree, information is stored for the computation of $size, closed$, $opened, nearOpen$, $farOpen, nearClose$ and $farClose$. $size$ stores the number of parentheses in the sub-tree of v, $closed$ and $opened$ store the total of closed and opened parentheses in the sub-tree. The structures $nearOpen$ stores the number of opened parentheses whose match can be found in the sub-tree of the B-tree and the $farOpen$ stores the number of opened paren-

theses is not found in that sub-tree, the $nearClose$ and $farClose$ are identical to $nearOpen$ and $farOpen$ but for the closed parentheses.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
Parentheses	(	(	)	(	(	)	(	)	(	(	)	(	)	)	)	(	)	(	(	(	)	(	)	)	(	(	)	(	)	)	)	(	(	)	(	)	)	)
Computing steps	1	2	1	2	1	0	1	0	1	2	1	2	0	0	0	1	0	1	2	3	0	1	0	0	1	2	0	1	0	0	0	1	1	0	1	0	0	0
block *opened*	opened = 2				opened = 0				opened = 2				opened = 1				opened = 3				opened = 0				opened = 1				opened = 1				opened = 0				opened =0	

Figure 2.15: The figure presents the computation of *opened* values for bitmap blocks. The example shown is computed over the parentheses bitmap of the suffix tree of "mississippi". Notice that the *opened* values are computed from left to right. The first row is the index position of the bitmap. The second row is the parentheses representation. The third row is the values computed for each index position during the computation of the *opened* values for each block. The fourth row is the result of *opened* for each block.

Computing the opened parentheses over blocks of bitmaps is done from left to right over the index. Start from the left of each block with a counter value at 0. For example in Figure 2.15 starting at the first index, and for blocks of size 4, add 1 for each index positions"1", "2" because they have opened parentheses, then subtract 1 because index 3 has a closing parentheses. Then add one for index position "4" which has a opened parentheses, therefore this block has opened value 2. The counter is never less than 0.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
Parentheses	(	(	)	(	(	)	(	)	(	(	)	(	)	)	)	(	)	(	(	(	)	(	)	)	(	(	)	(	)	)	)	(	(	)	(	)	)	)
Computing steps	0	0	1	0	0	1	0	1	0	0	1	0	3	2	1	0	1	0	0	0	2	1	2	1	0	0	1	0	3	2	1	0	0	1	0	1	2	1
block *closed*	closed = 0				closed = 0				closed = 0				closed = 3				closed = 0				closed = 2				closed = 0				closed = 3				closed = 0				closed =2	

Figure 2.16: The figure presents the computation of *closed* values for bitmap blocks. The example shown is computed over the parentheses bitmap of the suffix tree of "mississippi". Notice that the closed values are computed from right to left. The first row is the index position of the bitmap. The second row is the parentheses representation. The third row is the values computed for each index position during the computation of the *closed* values for each block. The fourth row is the result of *opened* for each block.

The closed parentheses computation is symmetrical to the computation of the opened parentheses. Therefore it is done from right to left over the index.

Start by the last index of each block and a counter with value 0. For example in 2.16 starting at index 4, and for blocks of size 4, do not subtract "1" for the open parentheses in index "4" because subtracting would make the counter negative. Add "1" to the counter for index position 3 and subtract one for index 2. At index 1 do nothing and the final *closed* value is 0.

Block Index	1	2	3	4	5	6	7	8	9	10
Block values	opened = 2 closed = 0	opened = 0 closed = 0	opened = 2 closed = 0	opened = 1 closed = 3	opened = 3 closed = 0	opened = 0 closed = 2	opened = 1 closed = 0	opened = 1 closed = 3	opened = 0 closed = 0	opened =0 closed =2
Values over 2 blocks	opened = 2 closed = 0		opened = 1 closed = 1		opened = 1 closed = 0		opened = 1 closed = 2		opened = 0 closed =2	
Values over all blocks	opened = 0 closed = 0									

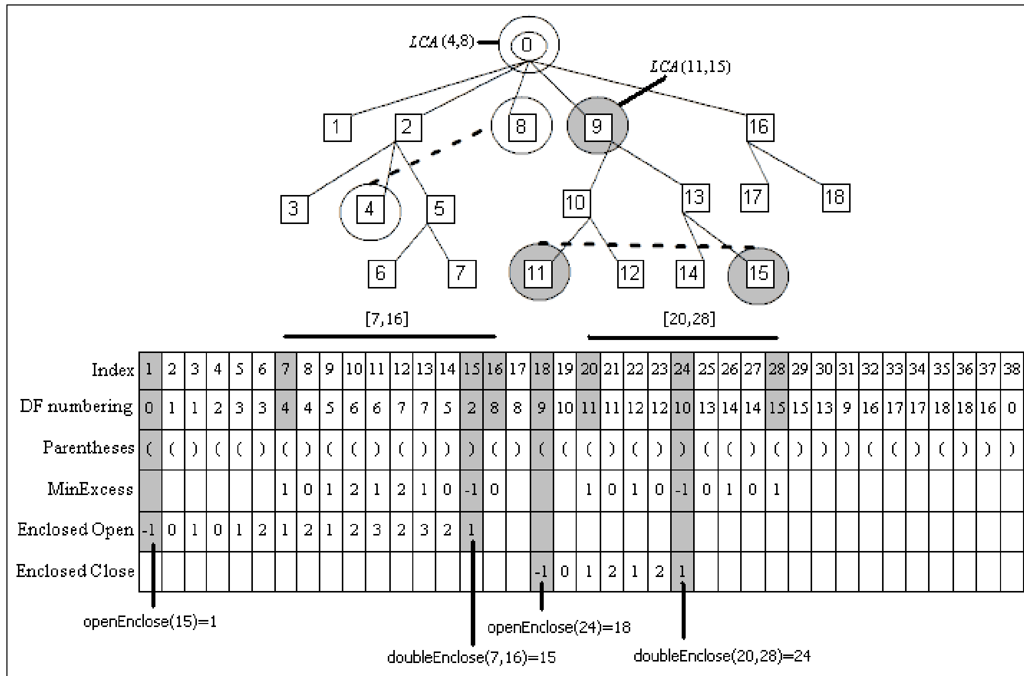
Figure 2.17: The figure presents the computation of *closed* and *opened* values for large ranges. The example shown is computed over the parentheses bitmap of the suffix tree of "mississippi". Notice that the closed values are computed from right to left. The first row is the index position of the blocks. The second row is the values *opened* and *closed* computed for each block. The third row is the result of *opened* and *closed* for two adjacent blocks and the fourth row is the total *opened* and *closed* for the bitmap.

The idea to compute *opened* and *closed* is extended to larger blocks using the results of each block. For example to compute the *opened* of blocks 3 and 4, Figure 2.17, use the *opened* of block 2 and subtract the *closed* of block 4 witch is less than zero, therefore retain zero. Now add the *opened* of block 4 and obtain the *opened* value 1. The symmetrical is done for *closed* and iterating this rule for a larger range will also compute the *opened* and *closed* values.

Enclosing Parentheses

The second structure computes the double enclosing of two parentheses, i.e. the index of a pair of parentheses that contains two given index positions, (l, r) in the parentheses representation. The $excess(l, r)$ operation computes the number of opening parentheses minus the number of closing parentheses for the range (l, r). Notice that unlike the values computed for *opened*

and *closed* these values can be negative. The operation $minexcess(l, r)$ determines the index position in the range (l, i), $i \leq r$, with the smallest $excess(l, i)$ and smallest i. Then compute the enclosed parentheses of this index position with the B-Tree described earlier to find the LCA of index positions (l, r).



Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
DF numbering	0	1	1	2	3	3	4	4	5	6	6	7	7	5	2	8	8	9	10
Parentheses	(	(	)	(	(	)	(	)	(	(	)	(	)	)	)	(	)	(	(
MinExcess							1	0	1	2	1	2	1	0	-1	0			
Enclosed Open	-1	0	1	0	1	2	1	2	1	2	3	2	3	2	1				
Enclosed Close																		-1	0

Index	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
DF numbering	11	11	12	12	10	13	14	14	15	15	13	9	16	17	17	18	18	16	0
Parentheses	(	)	(	)	)	(	(	)	(	)	)	)	(	(	)	(	)	)	)
MinExcess	1	0	1	0	-1	0	1	0	1										
Enclosed Open																			
Enclosed Close	1	2	1	2	1														

Figure 2.18: The figure shows the tree topology of the suffix tree for the text "mississippi". The numbers in the tree nodes represent the numbering in depth first of the suffix tree, in white and dark circles are the nodes used for resp LCA(4,8) and LCA(11,15).

The LCA operations, in Figure 2.18, are computed over the parentheses bitmap with the double enclose operation. The double enclose returns the right parentheses of the son of LCA(4,8), therefore the enclose of the that position obtains $LCA(4, 8)$. Figure 2.18 also shows the parentheses representation of the tree with the computation of the lowest common ancestor of nodes (4,8) and nodes (11,15), resp index positions (7,16) and (20,28). The operations enclose and double enclose are used to compute the LCA, in the case (7,16) the $minexcess(7, 16) = 15$, which is computed in figure 2.18. The open enclosed parentheses of 15 which is node 0 at index 1. In the second

example (20,28) the $minexcess(20, 28)$=24. The open enclosed parentheses of 24 is at position 18, which corresponds to node 9.

2.4.3 Dynamic FCST

The static fully compressed suffix tree, FCST, cannot be used in a dynamic environment. As such there is a need to add a dynamic funcionality. Recently a dynamic version of FCST was proposed by Russo, Arlindo and Navarro[22]. A limiting factor to build the static FCST is the need for the uncompressed suffix tree, which spawns a large amount of space. Due to this requirement the FCST uses a large amount of space at build time. A dynamic FCST however can be constucted in optimal space, i.e. the construction process does not need more space than the final tree.

The dynamic FCST has much smaller space requirements than other implementations of compressed suffix trees[22]. The tradeoff is more time for most operations but space can be down to as much as a quarter of Chan et al.[15] space for DNA.

As promised in section 2.3.2 we will start by explaining the reduce operation. A bitmap B is mantained to compute the operation $Reduce(v)$. This bitmap starts with all positions set to "0", moreover for every sampled node $v = (v_l, v_r)$ positions $Select_0(B, v_l)$ and $(Select_0(B, v_r) + 1)$ are set to "1", figure 2.19. $Reduce(v)$ determines the position in the sampled tree where v should be, it finds the position of the paretheses to the left of the possible location of v, $Reduce(v) = Rank_1(B, Select_0(B, v + 1)) - 1$. The operation for the lowest sampled ancestor LSA does the mapping between the sampled tree and CSA. To solve LSA we compute $Reduce(v)$ and obtain the parentheses for v, if that parentheses is a "(" the LSA is at that position, if it is ")" LSA is the $Parent(Reduce(v))$.

For example we compute Reduce(10) in the suffix tree with the text "mis-

sissippi". In bitmap B of Figure 2.19 $Select_0(B, 10+1) = 22$, therefore $Rank_1(B, 22) - 1 = 10$. Consulting position 10 in the parentheses array we compute a closing, ")" parentheses. We also Compute Reduce(4), $Select_0(B, 4+1) = 10$, therefore $Rank_1(B, 10) - 1 = 4$. Consulting position 4 in the parentheses array we compute a opening, "(" parentheses.

We now compute the LSA of leaf 10 in figure 2.19. The operation $Reduce(10)$ indicates that leaf 10 should be after position 9, however returns the right parentheses, ")", of node 9. This indicates that $Parent(9)$ is the LSA of 10. We will also compute LSA of leaf 4 however in this case $Reduce(4)$ returns a "(" which indicates that we have found the sampled leaf node 4, as we can see in Figure 2.19.

The static FCST as well as the dynamic FCST composed of a CSA, a sampled suffix tree and mapping between these structures. Operations such as $SLink$, $LETTER$, $SDep$, LCA, LSA, $LCSA$, $Parent$ are supported by the dynamic FCST. The CSA designed by Mäkinen et al.[15] is used in dynamic FCST and has polylogarithmic time for operations with optimal space complexity.

To achieve compression in construction time the dynamic FCST starts with a empty text collection and progressively adds elements to its collection. The static approach requires a uncompressed suffix tree so it can compute which are the sampled nodes and erase the nodes not required for sampling. Taking advantage of the dynamic operations allows the progressive construction of the sampled tree without full information.

With the simultaneous construction of the sampled tree and the CSA some operations in these structures can take avantage of each other. However the build steps of CSA and the sampled tree have to be similar. The dynamic FCST specified by Russo, Arlindo and Navarro[22] uses a dynamic CSA designed by Mäkinen et al.[15] which has polylogarithmic time for operations and optimal space complexity. The CSA inserts caracters from right to left. The Weiner algorithm[25] for suffix trees works in the same way. Specifically the CSA starts inserting the last caracter and then progresses backwards in

the text. For each caracter it will find the insertion position by using the LF mapping, it then adds the caracter to the CSA as needed. Notice that LF indicates a suffix of size $(t+1)$ that contains the current suffix of size t plus a caracter at the start. Remember that LF is the oposite of the suffix link operation. These operations are used to travel in oposite directions of the suffix tree. The Weiner algorithm uses $WeinerLinks$ [25], for an internal node v the $WeinerLink(c, n)$ corresponds to the point in the suffix tree whose path-label is $c.v$ as it is very similar to CSA construction algorithm. The data structure to help build the suffix tree is a CSA therefore synchronization with the CSA insertion algorithm will allow for CSA to provide Parent and WeinerLink.

To delete a text we will first locate the node that corresponds to the text. Then using $SLinks$ remove the following nodes that correspont to suffixes of the text. Doing this syncronized with CSA will keep it coherent and maintain the suffix tree as a support structure. The CSA goes from right to left while deleting with LF, this can be changed to use ψ and go left to right[15].

Correct sampling is defined as a maximum distance from one node to a sampled node. Distance is the number of suffix links needed to perform on v before finding a sampled node. The sampled tree T requires that for every node v there is a sampled $SLink^i(v)$ and $i < \delta$. Therefore maximum distance of δ will allow a worst case to reach a sampled node after $\delta - 1$ suffix links.

The reversed tree T^R of T is a important concept to understand how the sampling property works. Russo et al. defined the reverse tree T^R as a minimal labeled tree[23]. Such that all nodes v in T have a corresponding node v^R with the reverse path-label of v. For a node v with a path label "sppi$" the corresponding v^R will have "$ipps". The mapping between a node in T and T^R is done with function R. A important relation that comes from R is $SLink(v) = R^{-1}(Parent(R(v)))$, for every node v the parent of v^R is the suffix link of v.

The tree height of a node represents the distance from a node to its farhest

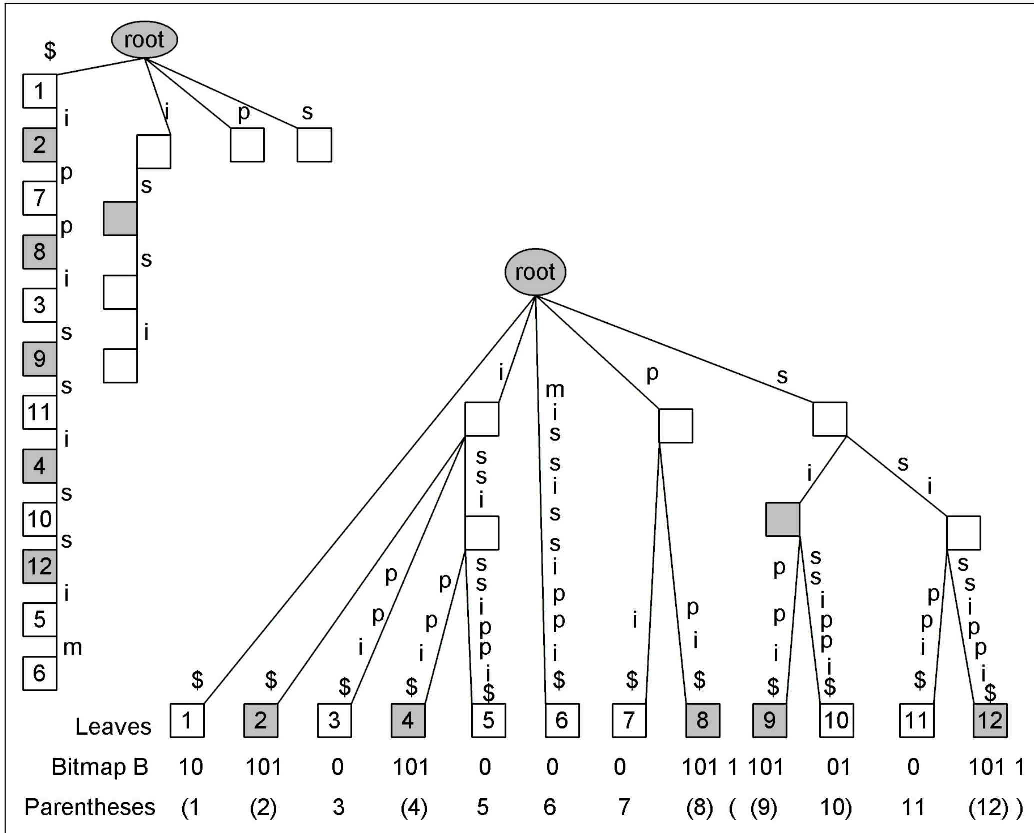


Figure 2.19: The figure shows to the left the reversed tree of the suffix tree for "mississippi". To the right is the sampled tree without suffix links and with the leafs sorted as they appear in the SA. Bitmap B bit values are distributed below the nodes to which they correspond in the suffix tree.

leaf and is represented as Height(v^R). we sample the nodes with Height(v^R)$\geq$ $\delta/2$ and $TDep(v^R) \equiv_{\delta/2} 0$. Note that because SLink removes one letter at a time the reverse tree is a trie, i.e. all edges in T^R are labelled with a single letter. So $SDep(v) = TDep(v^R)$.

An insertion or deletion of a node in the tree will not change the string depth of other nodes. As the $SDepth$ of T is the $TDepth$ of T^R this will not change either. All modifications to T^R will happen on the leaves, with elimination or insertion of new nodes. Another way to express this concept is that insertion or deletion in T will not break the suffix links chains. It will work at the ends of such a chain by inserting or deleting a node in T^R.

At insertion we need to check if some unsampled nodes need to be added to the sampled tree. Suppose Weiner algorithm subroutine determines insertion of a node $X.v$. To check the sampling consistency we will search for a ancestor in T^R whose tree heigth increased. This happens because a leaf $(X.v)^R$ is added as a descendant of v^R which may increase heigth. Checking is done by travelling upwards from v^R through T^R. We compute SDep because it is needed as $TDep^R$. If the distance from a node $(v')^R$ to $(X.v)^R$ is $\delta/2$ and $TDep((v')^R) \equiv_{\delta/2}$ the node v' meets the sampling condition and will be sampled.

For example we will compute the sampling caused when leaf 6 is inserted in the suffix tree of Figure 2.19. Assume this node does not exist, therefore node 6 in the reversed suffix tree does not exist and node 12 is not sampled. We insert the node 6 therefore the tree depth of T^R increases and we need to check the $\sigma = 4$ nodes above node 6. We reach height=2 at node 12, therefore it is a possible node for sampling however we need to check if there is another node at a distance grater than or equal to $\sigma/2$. Progressing up the tree we compute that node 4 is sampled and node 12 in T^R has tree height $\sigma/2$, therefore node 12 is added to the sampled tree.

Removing a node from T is the same as removing the correspondent leaf from T^R. Assume we remove the node $X.v$, to check consistency we will compute if some node in the sampled tree should be removed. In the same way as for insertion the reverse tree is scanned upwards and $SDep$ is checked. If v' is found such that$(SDep(v) - SDep(v')) < \delta/2$ then v' is a node that might need to be remove from sampling. But another path in the tree might need that node sampled, this happens if Height$(v'^R) \geq \delta/2$ is true for some other descendant. To control this the sampled tree will store in each sampled node the number of descendants with distance $\delta/2$. Whenever that count reaches zero, the node will be removed from sampling.

The operations Insert and Delete use $O((\log n + t)\delta)$ time, on top of that are some extra operations to manipulate the structure with the topology of S.

However these added operations do not overcome the $O((\log\ n + t)\delta)$ time. The dynamic operations on the CSA of FCST uses the result by Gonzalez et al.[7] which improves on Mäkinen et al.[15].

The dynamic sampled tree uses a parentheses tree structure S such as the one Chan et al.[15] describes. This structure, with a list of $O(n/\delta)$ nodes, uses $O(n/\delta)$ bits and spends $O(\log n)$ time for $FindMatch$, $Enclose$, $DoubleEnclose$, $Insert$ and $Delete$, 2.4.2.

To find the sampled parent of v, $Parent_S(v)$, the parentheses structure uses $Enclose(v)$. This operation finds the nearest pair of parentheses that contain v. For the $LCA(v_1, v_2)$ the $DoubleEnclose(v_1, v_2)$ will find the closest parentheses that contain both nodes. The Rank and Select operations are used over the parentheses sequence to manage additional information over nodes. A structure over a bitmap of n bits with Rank, Select, Insert and Delete uses $O(log\, n)$ time and $nH_o + O(n/\sqrt{log\, n})$ bits.

The dynamic FCST stores $SDep$ and a counter for each sampled node. The Weiner algorithm has a property that states that string depth does not vary, and makes changes on the $SDep$ unnecessary. Information for $SDep$ and the counter are kept in S. A sampling of node v means it is inserted in the structure, likewise removal if unsampling. The counters are saved in S but the values will change over insertions and deletions of nodes.

Table 2.2: The table shows time and space complexities for Chan et al. dynamic CST and Russo et al. dynamic FCST. The first row has space use and the remaining rows are time complexities. In the left collumn are operations, the middle column has the time complexities for Chan et al. dynamic CST and the right column has time complexities for the dynamic FCST.

	Chan et al. dynamic CST	Russo et al. dynamic FCST
Space in bits	$nH_k + On + o(n \log \sigma)$	$nH_k + o(n \log \sigma)$
SDep	$\log_\sigma(\log n) \log^2 n$	$\log_\sigma(\log n) \log^2 n$
Count/Ancestor	$\log n$	1
Parent	$\log n$	$\log_\sigma(\log n) \log^2 n$
SLink	$\log n$	$\log_\sigma(\log n) \log^2 n$
SLink^i	$\log_\sigma(\log n) \log^2 n$	$\log_\sigma(\log n) \log^2 n$
LETTER(v, i)	$\log_\sigma(\log n) \log^2 n$	$\log_\sigma(\log n) \log^2 n$
LCA	$\log n$	$\log_\sigma(\log n) \log^2 n$
Child	$\log_\sigma(\log n) \log^2 n$	$(\log_\sigma(\log n)) \log^2 \log \log_\sigma$
TDep	1	$((\log_\sigma(\log n)) \log n)^2$
WeinerLink	$\log n$	$\log n$

Chapter 3

Design and Implementation

This chapter presents the design of our dynamic data structures for the DFCST.

3.1 Design

The top abstraction of the software library is the run, config and tester files. However the FCST class has the functions of the dynamic suffix tree and is therefore relevant to present its description in section 3.2. The diagram in Figure 3.1 omits most data fields and shows the most relevant instance level relationships and class level relationships.

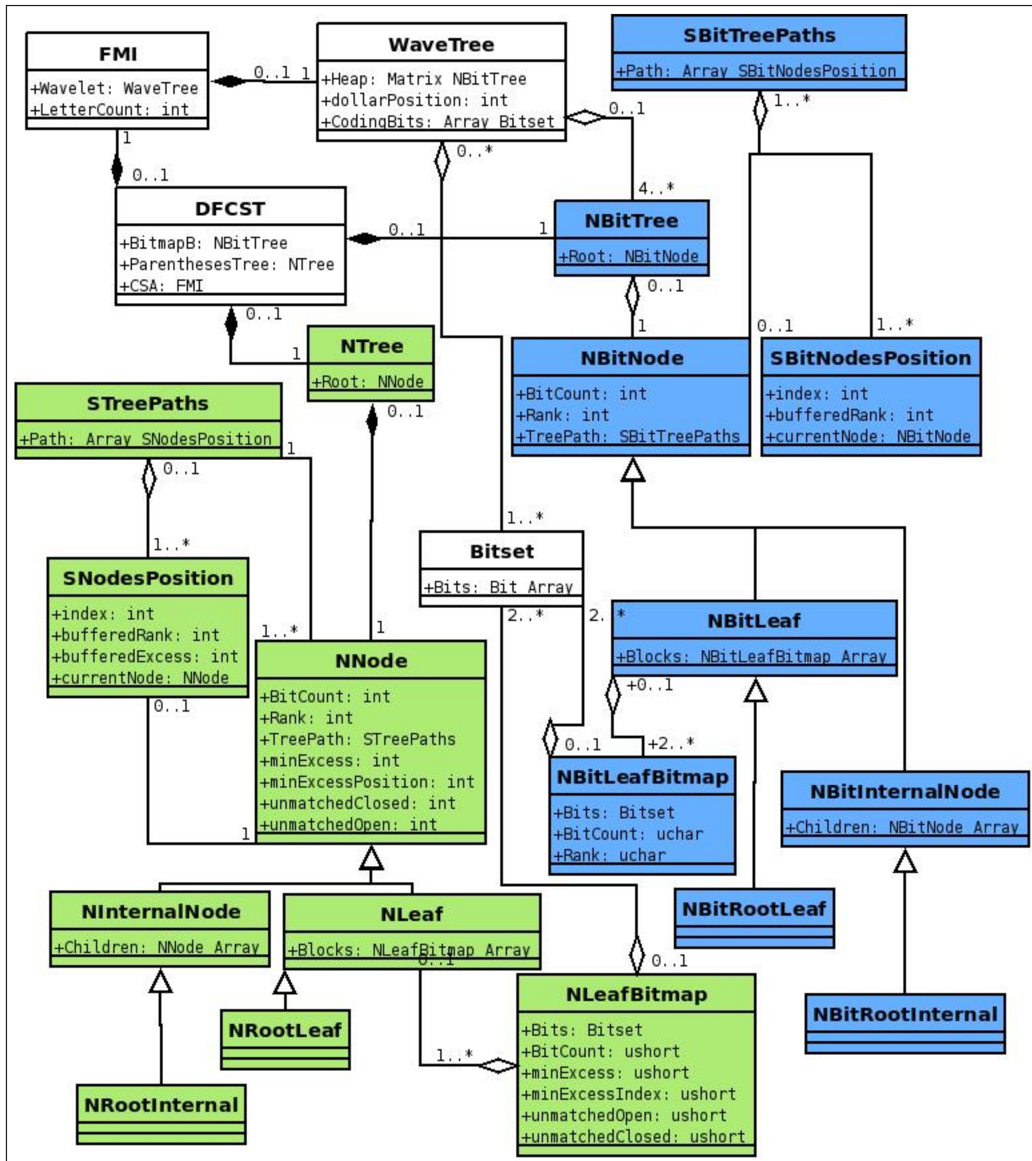


Figure 3.1: The figure shows the resumed class diagram for the implementation. In the pink circle are the higher level classes such as DFCST, FMI and the LSA. The green circle has the parentheses tree and in the blue circle is the bit tree.

3.2 DFCST

The dynamic FCST is formed by three data structures: these are the FMI; the B bitmap and the Sampled Tree. The data structure used by the dynamic

FCST is represented in Figure 3.2, the data structure is explained with detail and the suffix tree information is simulated.

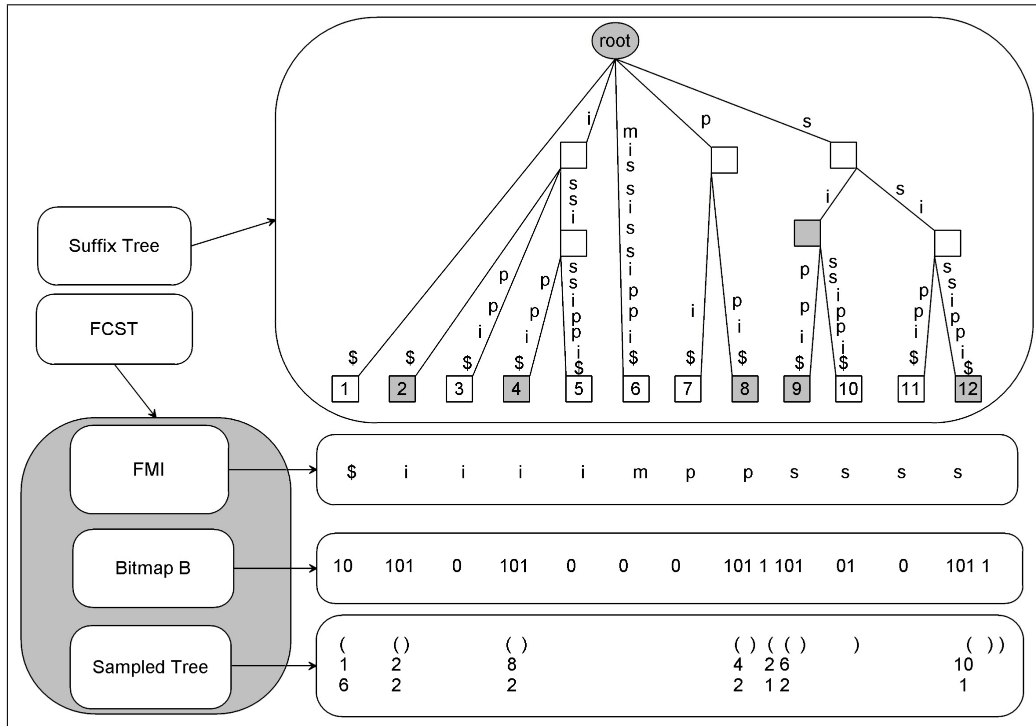


Figure 3.2: The suffix tree structure and the DFCST structure presented in parallel.

The main operations for this dissertation are the insertion and removal of a single letter from the text. This allows a dynamic suffix tree to function, grow and contract.

3.2.1 Design Problems

The implementation of a node insert in the dynamic FCST discovered a new problem, the degree of parenthood between the inserted node and other sampled nodes is unknown. This is solved determining the tree depth of the inserted node and its parent in the sampled tree. The difference between tree depths is the relative position of the new node among its brothers.

For example, when some internal node is inserted within the sampled tree the left and right child leaf of the node is known. However this information is not enough to determine where to insert the string depth in the sampled tree. In Figure 3.3 the case 1 and case 2 are both possible in the insertion of the node "A". Case 1 creates the inconsistent suffix tree because the string depth within the sampled tree was stored at the wrong position.

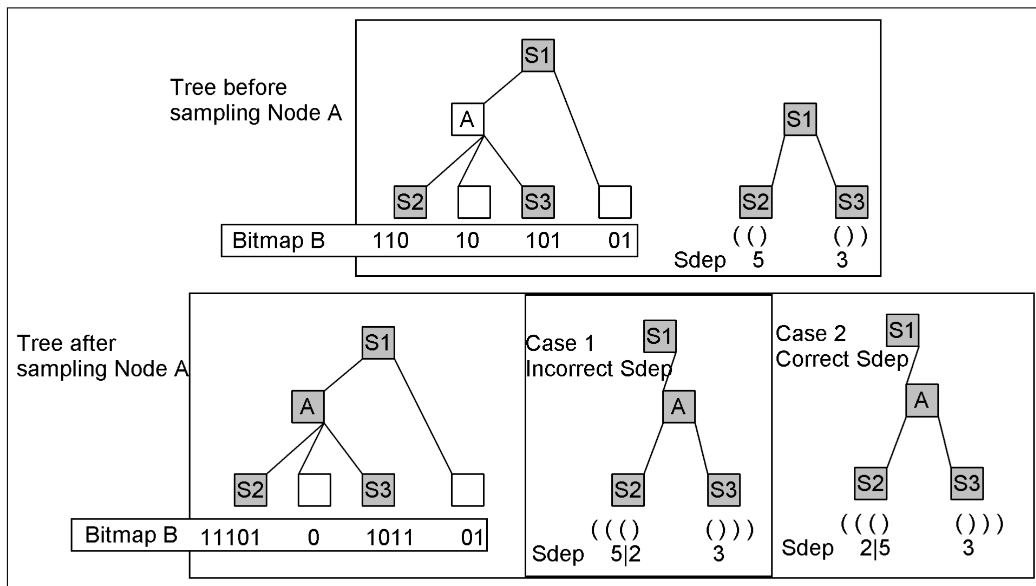


Figure 3.3: A sampling creates a sampled node with the wrong string depth. On top is the original suffix tree with the sampled tree. On the bottom is the suffix tree after node A becomes sampled and the two cases possible for the sampling.

Other situation arises when inserting the zero of a new leaf in bitmap B. In Figure 3.4 is the insertion of a left child to "S1". The middle image is the right solution where the "0" is inserted a "1" to the left of the "0" related to "S2". Notice that both cases are correct if we consider that the operation considers the correct position is to the left of "S2". However the lack of information to backtrack one "1" will cause the right tree in the image, which creates a inconsistent suffix tree.

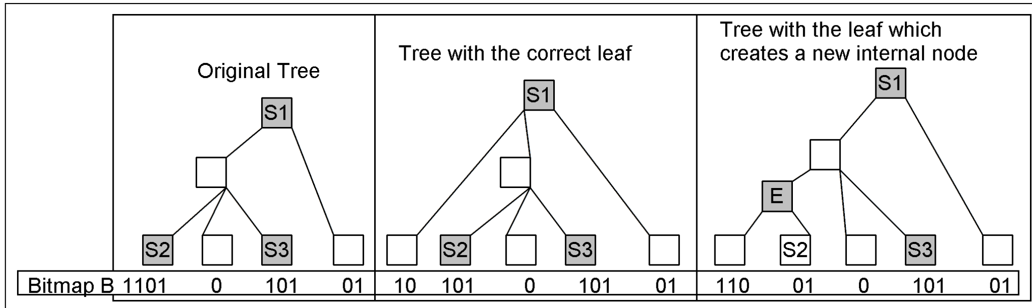


Figure 3.4: Leaf insertion creating a sampling error. On the left is the original tree. The center tree is the correct tree after inserting the left leaf. The right tree is the inconsistent tree that can be generated.

3.2.2 DFCST Operations

This section describes some critical operations of the DFCST, the operations are the insert and remove, the LCA, the Parent and Child.

Insert Letter

To insert a letter the operation first computes the weiner node of the new letter. The initial weiner node is the root, when inserting a letter the previous weiner node is used to compute the new Weiner node, the LF and Parent operations are iterated over the old weiner node until the root is found or LF is successful. If LF is successful it returns the new weiner node, otherwise the root is reached and the child, using the inserted letter, of the root node is the new weiner node. If child is unsuccessful the Weiner node is the root.

The new weiner node may require a new sampled node, therefore the insert operation computes the closest sampled node up to a maximum distance δ, a node A is distant by one unit from node B if SuffixLink(A) = B. If such distance is reached and no sampled node is found the node at distance $\delta/2$ requires sampling.

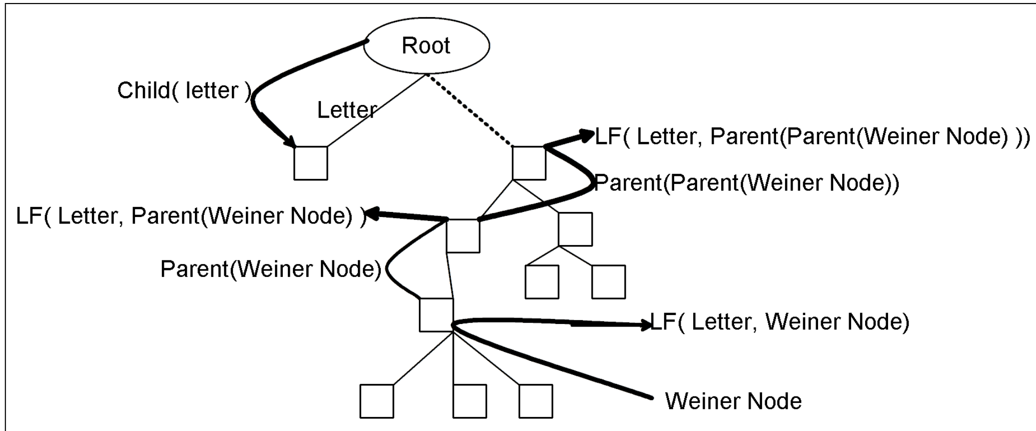


Figure 3.5: The weiner node computation over a suffix tree.

To sample a node, insert two "1"s in bitmap B, one at the left and one at the right of the "0"s which corresponds to the left and right limits of the node. To determine the position within the sampled tree compute the tree depth of the lowest common sampled ancestor of the node left and right. Next compute the tree depth of the left of the lowest common sampled ancestor. The difference between these two tree depths indicates the relationship between the new sampled node and the sampled brothers and the deviation of the insert position within the sampled tree.

The new leaf is computed when a new letter is inserted in the FMI since it corresponds to the dollar position in the BWT. Furthermore to store the leaf in bitmap B, the difference between the tree depth of the leaf lowest sampled ancestor and the Weiner node lowest sampled ancestor tree depth is computed to determine the new leaf position. This is due to the problem described in 3.4.

Remove Letter

The removable leaf is the largest suffix of the suffix tree. The remove operation computes the parent of this leaf to obtain the Weiner Node. Furthermore

it removes the sampling of the Weiner Node and the sampling of the leaf. The sampling removal and insertion are symmetric. Finally it removes the largest suffix position from the FMI which corresponds to the suffix leaf in bitmap B.

LCA

To compute the LCA of two nodes we iterate delta times the Suffix Link delta times, for each iteration compute the LSA string depth. Compute LF over the node with the largest string depth. This is the reverse operation of SuffixLink and therefore computing the reverse path the same number of times will return the LCA.

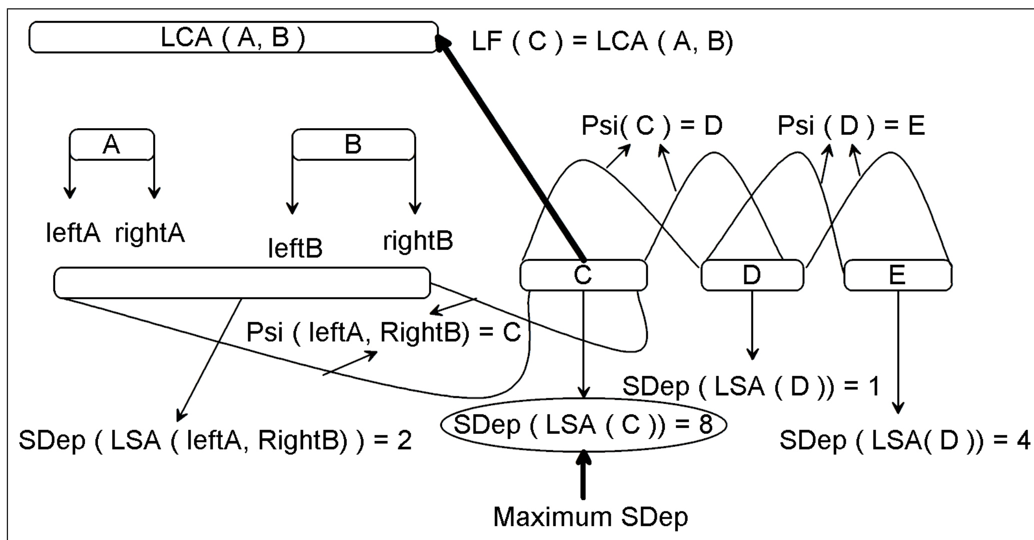


Figure 3.6: Computation of the LCA of two nodes A and B using the LSA, Psi and SDep.

Parent

Parent of a node is computed with the LCA operation over two nodes. To compute the parent of node $v(l, r)$, compute the LCA of $(l-1, r)$ and the

LCA of $(l, r+1)$. The minimum of the LCAs left sides is the Parent(v) left side, the maximum of the right sides is the right side of Parent(v) and the maximum LCAs string depth is the string depth of the Parent(v).

Child

The child operation is computed with a binary search over the range of a node. This binary search determines the sub interval of the node which has the letter desired for the child suffix. To determine letter, the binary search iterates Psi and Letter up to the string depth desired for the child.

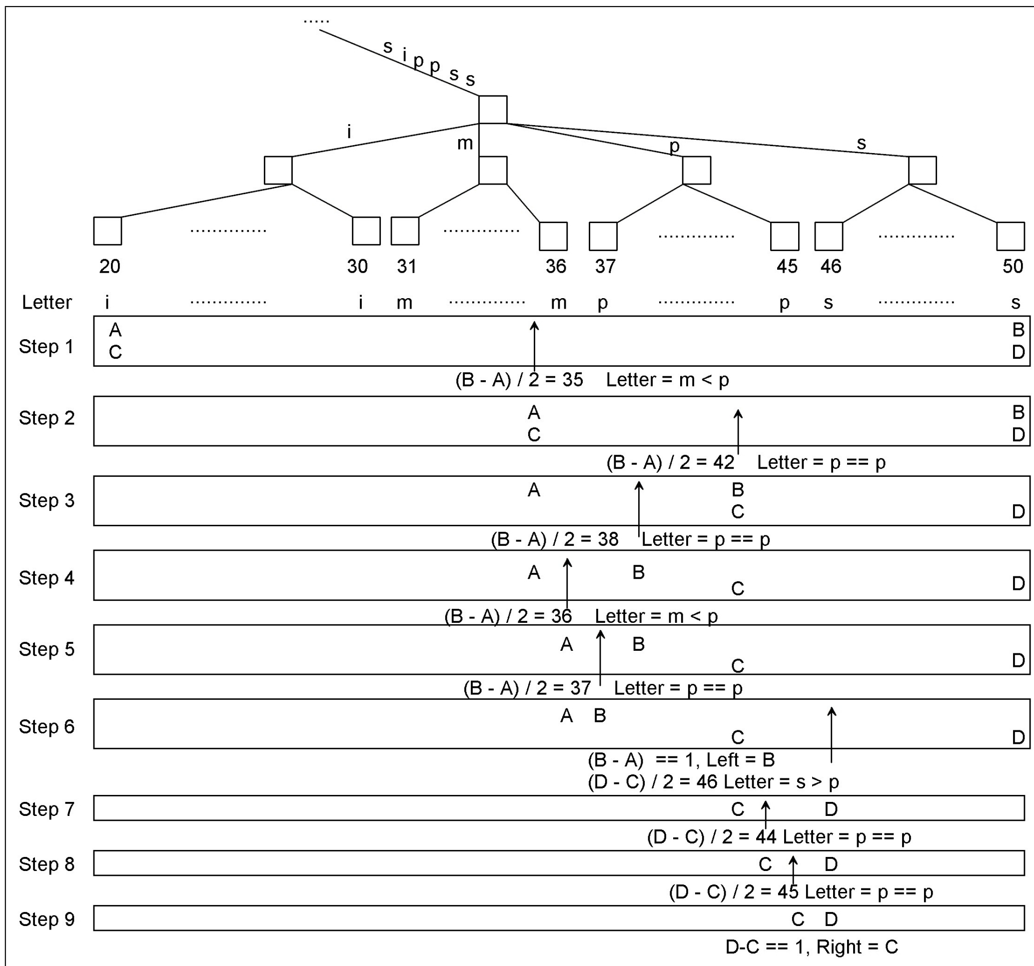


Figure 3.7: Child computation with a binary search. On top is a subtree of a suffix tree. In the rectangles are the positions of the indexes and between the rectangles is computation of the their positions in the next iteration of the binary search.

3.3 Wavelet Tree

The implementation of the wavelet tree in Figure 3.8 shows all the data required for the wavelet tree of the suffix tree with the text "mississippi$".

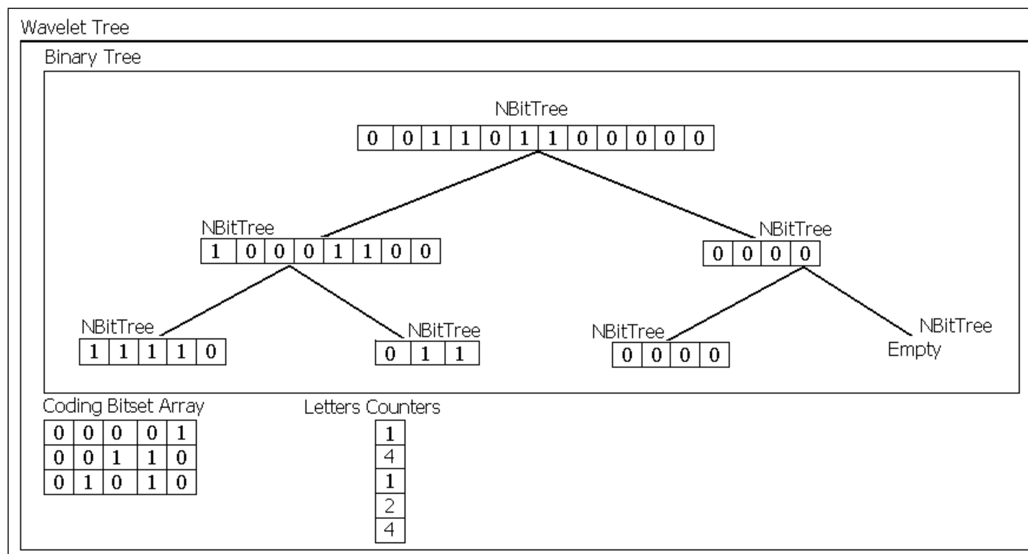


Figure 3.8: A implementation of the wavelet tree data structure using a binary tree.

It is possible to answer all queries over the wavelet tree with this binary tree and the coding bit set array. However notice that answering a query such as counting the number of letters "m" requires several operations over three bitmaps. Notice also that the binary tree is static, meaning it will never grow more nodes than in its initial size and it will not remove nodes.

3.3.1 Optimizations

One optimization is to transform the Binary tree in a Heap. This is implemented in the thesis and Figure3.9 shows the binary tree and the Heap structure of the text "mississippi$".

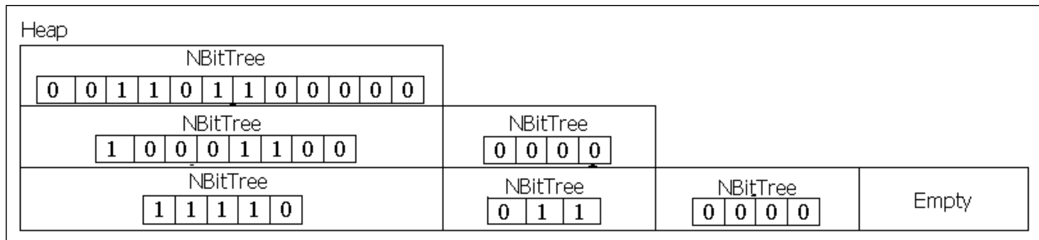


Figure 3.9: The wavelet tree data structure using a heap.

The Heap is a pointers matrix and all access operations dispense the operations related with travelling in the nodes of the binary tree, therefore this is faster than the previous proposed implementation. Furthermore, the algorithm determines if the NBitTree structure is used or not, in which case it dispenses its creation.

Compacting

The wavelet tree can be compacted, notice in Figure 3.9 there are three bitmaps with repeated information. This is because there are paths in the binary tree where the wavelet tree only stores repeated information. Therefore this information is replaced with counters and the changes necessary to deal with this new operation were introduced. The removal of unnecessary information can be seen as a tree trimming or a form of compacted wavelet tree.

The bitmap with the dollar sign is the left bottom bitmap, this bitmap only needs to store the amount of letters and the dollar position. Also notice that operations over these bitmaps no longer need to be computed within a bit tree and therefore cost constant time.

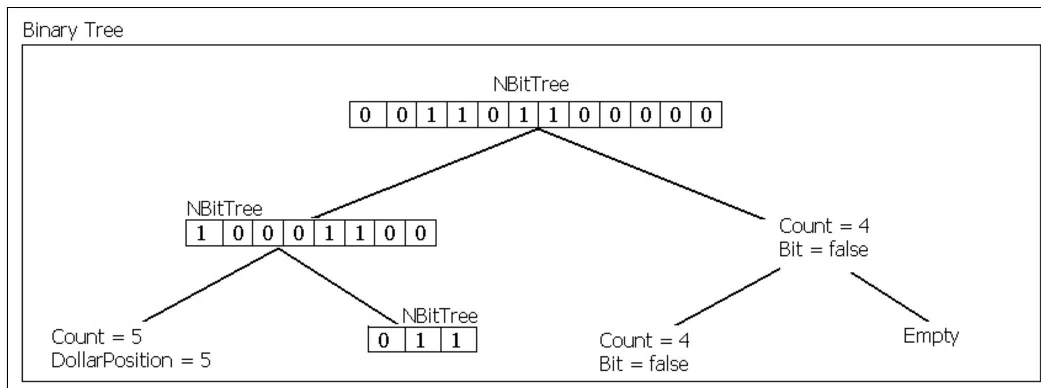


Figure 3.10: Compact wavelet tree data structure implemented with a binary tree and bitmaps stored in bit trees.

3.3.2 Operations

The operations *letterCount* and C require several operations over the bitmaps to answer very straight forward queries such as a letter count. Therefore it makes sense create a counter for each letter. With this counter these queries are answered in near constant time.

The operations for *insert*; *remove*; *access*; *sortedAccess*; *Rank* and *Select* are implemented as explained theoretically. However I choose to use a different algorithm to implement the *sortedAccess*. It is much more efficient in a real scenario than the theoretical proposal because it does not require rank or select over the wavelet tree. It returns the letter in near constant time.

The operation *sortedAccess*, as opposed to the *access* operation, consults a position in the wavelet tree and returns the letter inserted in order as it appears in the first column of the BWT, which is also the same letter in the suffix array.

The *sortedAccess* of position 10 is the letter 's', notice the number of letters from '$' through to 'm' is 8, therefore smaller than 10 and cannot be the letter in that position. However adding the number of letters 's' to the previous letters exceeds position 10. Therefore the *sortedAccess* of that position is

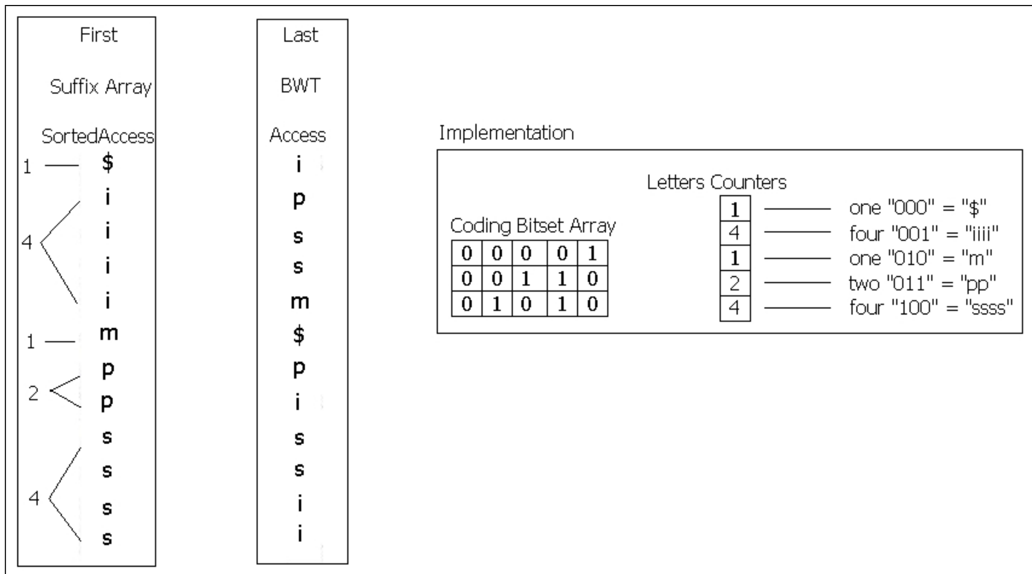


Figure 3.11: The left column is the lexicographically sorted text return by sorted access. The center column represents the BWT of the text, returned by the access function. The box to the right is the information stored to simulate the left column.

's'.

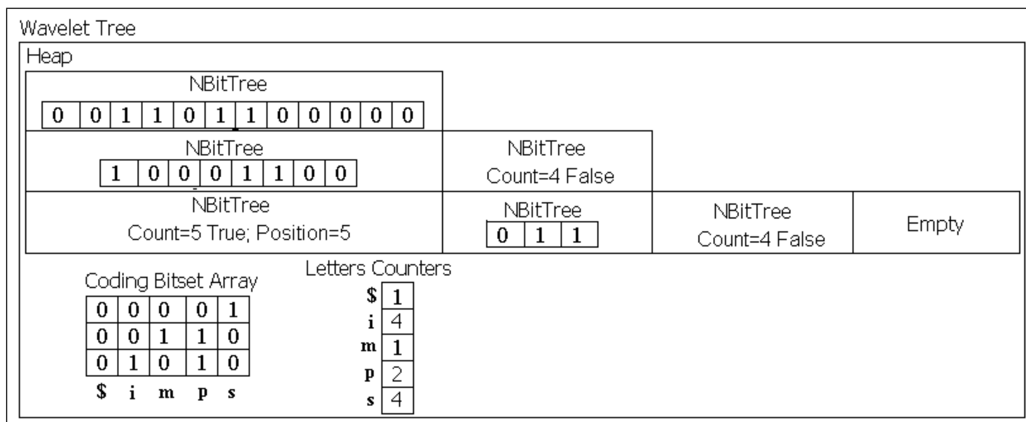


Figure 3.12: The implemented wavelet tree data structure. A heap with bit trees, with three bit trees replaced by a three pairs of booleans and integers.

3.4 Bit Tree

This dissertation was supposed to use a data structure developed by Wolfgang Gerlach and used by Veli Mäkinen in his CST et al [20] to solve the dynamic rank and select problem over a bitmap. However the dynamic FCST depends on this structure for space and for speed, therefore a very compact and fast structure was needed to solve this problem and the results obtained from the data structure by Gerlach would compromise the implementation.

However during the implementation of the parentheses tree based on the proposal by Chan et. al [3] and Veli Mäkinen et al [15], it became evident that it was also a bit tree for rank, with some other more complex data structures on it to support operations like enclosing parentheses and matching parentheses. With that in mind the select function was added and I removed all operations as well as data structures not essential or related to rank and select to create our bit tree prototype.

The bit tree created is a B+. The B+ does not spread the elements through the internal and leaf nodes, it stores the elements only in the leafs. This may seem as waste of space since some nodes are not used to store elements, however the loss of space is not significant. The B+ arity decreases the weight of the internal nodes and almost all space is used in the leafs. Therefore the programmer is able to concentrate efforts to reduce the space used in the leafs.

3.4.1 Optimizations

A very large global variable is used for operations management, this allows some important buffer features such as constant time access to any tree depth layer within the tree path.

- Removed the memory allocation calls for new bit sets in the NBitLeafBitmap and allocated large blocks of memory instead
- Removed the memory allocation calls for new NBitLeafBitmap in the NBitLeafBitmap and allocated large blocks of memory instead
- The tree redistributes the bitmaps when one bitmap is too full to prevent new memory allocations
- The tree redistributes the nodes when one is too full to prevent new memory allocations
- The dynamic pointers storing childs in internal nodes compensate the excessive memory allocation in the leafs. Allowing a very large trim in overall space because there are no unused allocated leafs
- The leaf does not need to have children node pointers like the internal nodes. This overhead of 8 bytes per bitmap block exists in the solution by Gerlach
- I used a very large global variable for operations management, so there is no need for father pointers
- Used only the essential structures as unsigned variables and as few bytes as I expect they will ever need

The removal of calls to "new" and "malloc" are significant improvements from my initial results because the memory manager of the operating system uses plenty of memory to store information on these two calls. Another significant improvement is the buffer system.

3.4.2 Buffering Tree Paths

Operations in the b+ tree are expensive since all operations require iterative scans over the child's at each node level. For example the configuration cho-

sen for the prototype has arity 100 and can generate a tree with tree depth 5 or 8, therefore descending the tree requires some processing. Furthermore several operations, such as enclose, insert or delete require a tree path to compute siblings and travel through the various tree levels. Therefore there is no reason not to store this tree path and in the advent the next operation occurs in the same leaf area, trying to adjust the tree path instead of recalculating it is less expensive.

The buffer stores a pointer to the consulted bitmap and the operation is performed over that position. The tree path is stored since a buffer is stored for each tree level of the b+ data structure. Therefore if a new desired path is dissimilar to the buffered path, the algorithm attempts to adjust to the new operation path rather than recalculating the whole path. The buffered tree path is a array and either the buffer is successful or the buffer is too dissimilar and is discarded in constant time. Since this is not essential to the thesis this buffer feature was not completely extended to the internal nodes.

For example, inserting a bit at a position involves computing a tree path and finding the right bitmap in the leaf. The tree path is calculated recursively computing three integers each time it descends a level, a bit position relative to that level, a rank relative to that position, and the child where to descend to the next level. The tree path stores the current node and the index to descend to the next tree depth. In case the current node is a leaf it additionally stores the relative position of the start of the leaf; the relative rank to that position; the relative position to the bitmap where the bit is; the relative rank to that bitmap; the position where the bit is and the relative rank to that position.

The buffering stores information on some ranges of positions. It is possible to calculate if the result of the operation is within the leaf range with some data about the range start and end. The leaf buffering stores one such range, and the leaf index buffering uses the index information to split the leaf in two parts. If the result is within the leaf the scan is processed before or after

the buffered index depending on the result position.

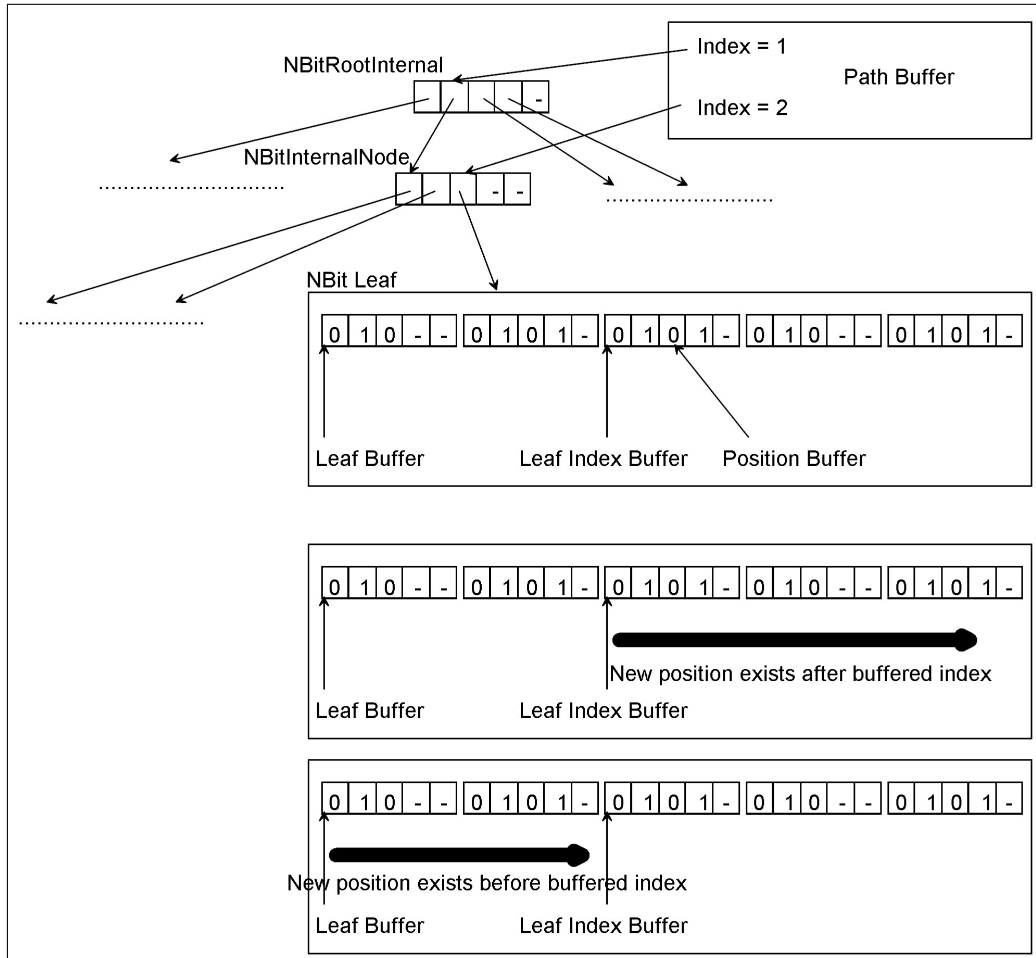


Figure 3.13: Above is the the bit tree internal nodes and one leaf with the index buffered. Bellow are two possible cases of the bitmap level buffering, before or after the buffered index and in the leaf.

In Figure 3.13 shows the buffered positions that are the result of any operation over the bit tree. A new operation will use the buffer following three steps:

1. If the result coincides with the "Position Buffer" go to 5

2. If the result is out of the limits of the buffered leaf go to 6

3. If the result is after the "Leaf Index Buffer" scan after the index to compute the result, go to 5

4. Do a scan starting at the beginning of the leaf to compute the result

5. Store the new buffered index, position and data needed. Terminate operation

6. Failed to use buffer, do a tree traversal to compute the result. Terminate operation

This allows us to answer the next operation over that leaf without computing all the previous steps to descend through the tree. The tree arity can be very large and the buffer will have a large impact on operations speed as it will prevent some internal nodes scanning. The size of the leaf has impact on the number of times the buffer is used, therefore the larger the bitmap blocks and the arity, the larger the leaf will be and the number of random times the buffer is used will increase.

This buffer only stores one possible path therefore the operations will take advantage of this buffer if they are within that path range. Every new operation overwrites the buffer because I consider that operations are more likely to use the buffer of the previous operation than the buffer of some other operation before that.

The path buffer is used to accelerate the insert and remove operation. When the critical bitmap occurs the bitmaps brothers are accessed in constant time and without any additional scans. This property extends to the nodes (internal or leafs) who become critical and need the brother nodes to rebalance.

The path buffer does not subsist if the operation that creates the buffer changes the structure of the tree. This is the case when inserts and deletes create a critical bitmap that requires a redistribution of the bitmaps and

possibly reshuffle the nodes balance. All other situations generate a valid buffer.

All operations are equivalent to the Rank1 operation. The insert; delete and getPosition count the number of bits down the tree path to reach a certain position. To buffer the Rank1 of that position the operation stores the number of "1"s the position is computed. The position as well as Rank1 is buffered, therefore the buffer is used for these operations.

Rank0(n) is the position n minus Rank1 (n), therefore Rank0 creates a Rank1 buffer and uses the buffered Rank1. Select is different to compute. However it is the reverse of Rank and therefore it is possible with some additional programming to transform Select in Rank. This is done in the bit tree. Select0 and Select1 both can use the buffer Rank1 for their operations and also create a buffer Rank1 without any loss of computations.

Other important point in time reduction is that the tree avoids memory allocations. Therefore prevents large memory copies to fill those new allocations. The tree also avoids small bitmaps redistribution's, if it needs to re-balance two bitmaps it will make the maximum re-balance in a single copy. For ex-

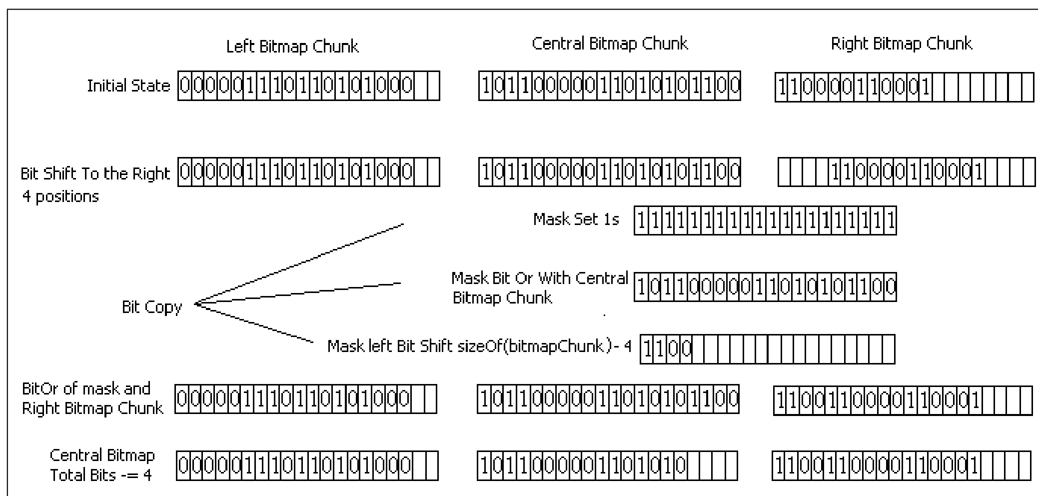


Figure 3.14: The redistribution operation of a critical bitmap block.

ample, in Figure 3.14 the central bitmap has 20 bits and is critical because it

has reached the maximum capacity. Therefore it will try to re-balance with one of its brother bitmaps. For chance one of its brothers (the one to his right) has plenty of room, more exactly 8 empty spaces. Therefore the left bitmap inserts 4 bitmaps at the start of the right bitmap.

3.5 Parentheses Bit Tree

The parentheses bit tree is based on the proposal by Chan et al [3] for parentheses maintenance and on the proposal by Veli et al [15] for the bitmap maintenance. The parentheses bit tree represents a tree with variable number of child's and operations getNode, parent, child, lca, insertNode and remove node. All nodes have a integer identifier. In the bit tree this identifier is the position of the node within the bit tree.

The parentheses bit tree is built on a B+ data structure, the structure for the bit tree is created with bitmaps in the leafs and functionality added in the form of Rank. Therefore this structure can provide insert, delete, rank and getBit. This is explained in the bit tree chapter, however it is implemented in a fundamentally different way in the Parentheses Bit Tree because the space bit ratio is not as demanding. Therefore some techniques to improve the bit ratio in the Bit Tree are not implemented in this data structure.

The buffer technique used in this implementation is different of the one described in 3.4.2. This is because transforming one operation in another equivalent is easier with Position, Rank and Select than it is with LCA or open enclose. However adding extra data and extra features this implementation is to fulfill the objective. All operations are converted into a buffered operation, this operation is both a Rank1 operation and a Position operation, the data stored includes the normal tree path and a extra path with parentheses information to allow the Parentheses operations that travel the tree path. These more complex operations such as enclosed parentheses, Parent, LCA,

open enclose, also generate a buffer unless they compromise the path buffer when travelling the tree.

This dissertation needs a tree with LCA, Parent, Child, and requires a variable number, n, of child's. This needs a call to new and a pointer to a array like structure of pointers to various child's. Therefore space is at least (3 + n) * 4 bytes. However this is the theoretical value, in fact the space is larger, even for a small n. This structure requires two "new" operations for each node, one to allocate the node, and other to allocate the array of child's. The operating system memory manager has to store data for each "new" call. A "new" call allocates multiples of a processor word, therefore for example if this word is 32 bytes and the operation requests 8 bytes, the new call allocates 32 bytes. If the operation requests 34 bytes, it allocates 64 bytes. It also uses a pointer to the reserved memory, a byte counter, a owner ID, a pointer to the class where "new" is called, and other data witch adds to the requested memory.

The DFCST requires two integers per node to store string depth and a counter for the dependent nodes operation. It also requires Rank and Select over the suffix tree nodes. The parentheses bit tree has Rank and Select of nodes, however it does not store two integers per node. Therefore the modified parentheses bit tree is a DFCST parentheses bit tree with a overhead of two integers per node. This overhead is in fact larger than two integers because the bit tree structure requires a array for every new integer, for every bit in the bit tree there are two integers, therefore because a node has two bits it uses 4 integers. The wasted space is presented in Figure 3.15 as 0's in the rows for string depth and for dependent links.

The space overhead is large, and since the first array is a bitmap, and both the String Depth and Dependent Links are 4 bytes arrays, nearly all space is in the 4 bytes arrays. Therefore it is very advantageous to remove space overhead in these integers arrays. For every opening bit "1" exists a corresponding "0" bit. The bit tree is able to compute this bit with a matching parentheses

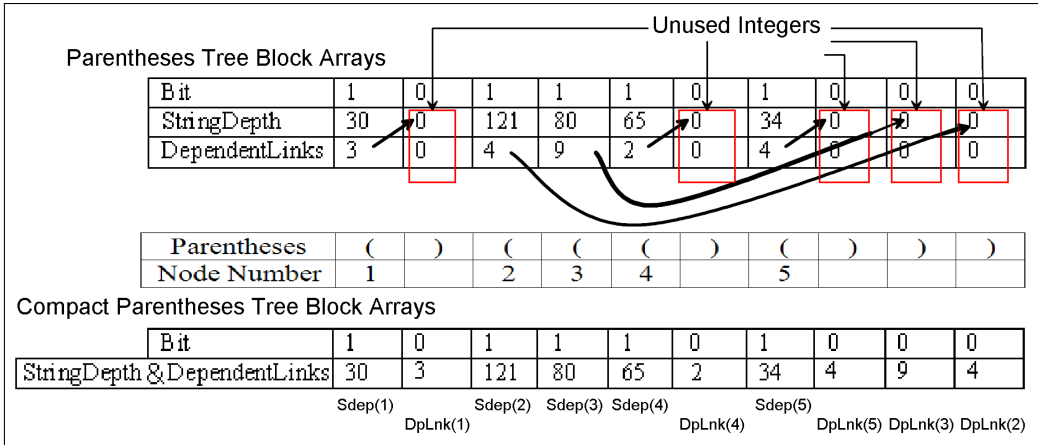


Figure 3.15: On top is the representation of the parentheses tree leaf data needed by the DFCST. The arrows represent direction where a node info is stored, it is the technique used to compact the leafs. Below is the corresponding leaf information as is implemented.

operation. Therefore the data structure stores data in the matching closed parentheses of a node. There is a overhead of computing the matching closed parentheses, however the space gains are large.

3.5.1 The Siblings Problem

The operations Parent, Child and Sibling are solved in the parentheses bit tree in two levels of abstraction. To solve the problem of Parent, Child and Sibling within the bitmap uses the matching and enclosing parentheses as proposed by Chan et al [3]. However the B+ tree data structure that stores these bitmaps is also required to solve this problem, to do this within the B+ four options were considered:

1. Store a pointer to the parent in each node.

    - The father is calculated in constant time.
    - To calculate a child the operation scans the N children and select

the node required.

- To calculate a sibling calculate the father and then do a scan to find the sibling.
- This option requires a parent pointer in each node.

2. Store a pointer to the parent and respective child number in each node.

    - The father is calculated in constant time.
    - To calculate a child do a scan of the N children and select the node required.
    - To calculate a sibling calculate the father and then increment or decrement the child number of the node.
    - This option requires a parent pointer in each node and a integer that stores the node child number in the parent node.
    - The dynamic structure requires nodes rotations, the rotation of the first child of a node will need to decrement or increase all the children position information. This problem extends to other situations such as node splitting and merging.

3. No additional data is stored in the tree structure. If a node is reached it is because the node parent and child number is already computed, the same is recursively true for the parent's parent, therefore this data can be used.

    - The father is calculated in constant time.
    - To calculate a child do a scan of the N children and select the node required.
    - To calculate a sibling first calculate the father and then increment or decrement the child number of the node.
    - Use a auxiliary array that is updated every time a node is visited.

4. Using solution 3 and creating a support for a binary scan to determine a child node.

- The father is calculated in constant time.
- To calculate the child use the binary search.
- To calculate a sibling first calculate the father and then increment or decrement the child number of the node.
- Use a auxiliary array that is updated every time the operations visit a node.
- Each bit insertion or deletion requires a scan of the Parent nodes, their auxiliary arrays that perform the binary search, and the arrays of Children nodes.
- A binary search for a Child node is unique to the type of operation being computed, for example *Rank* and *Position* are calculated differently and each operation needs its own array of binary search within each node.

The Sibling computation of this B+ tree data structure is important because the operations supported in the parentheses tree such as the matching parentheses travel sideways in the bitmap, whenever the end of a bitmap block is reached it is necessary to compute the blocks Sibling in the tree data structure. It is also important to avoid storing additional data for each bitmap. Therefore the development choice is number 3, this option lead to the development of the buffering path in the tree because the buffer uses this tree path for its operations.

Chapter 4

Experimental results

This chapter presents the performance results of the developed prototype and their comparison with similar prototypes. Section 4.1 presents the results of the prototype for dynamic *Rank* and *Select*, that uses a B+ tree, and compares with the implementation by Gerlach et al [25], that uses a red black, RB, tree. Section 4.2 shows the performance of the parentheses tree. Section 4.3 discusses the results obtained before and after compacting the wavelet tree. The chapter finishes with section 4.4 were we present a comparison between the DFCST, the FCST and the CST.

Table 4.1: Test machine descriptions.

	Machine Windows	Machine Linux
Operating system	Microsoft Windows	Ubuntu
Operating system version	Media Center Edition SP3	9.10
Ram	1 gigabyte at 200 Mhz	2 gigabytes at 666 Mhz
Processor	AMD Turion	Q6600 Quad Core
Processor Speed	2.00 Ghz	2.40 Ghz

The test results presented in sections 4.1 and 4.2 were performed in the

windows machine described in table 4.1. The results presented in section 4.4 used the linux machine, with the exception of the results presented in table 4.3. The tests in table 4.3 could not be all computed in the same machine, since the FCST was not developed for windows and the CST was not developed for linux. However since it is a space test and does not require time performance the CST used the windows machine, the DFCST and the FCST used the linux machine.

4.1 Bit Tree

The bit-tree is the data structure that supports a dynamic bit array with *Rank* and *Select* operations, therefore it is an essential component of the wavelet tree and of the B bitmap in the DFCST. Furthermore the parentheses tree is a similar data structure and the bit tree performance is an important indicator of the parentheses performance. The operations over the bit-tree consume a relevant portion of the overall time and therefore speed improvements over this data structure have a direct impact in the overall time of the DFCST implementation.

The tests for this data structure were performed over an 100 megabit random bitmap. The space ratio represents the number of bits necessary to represent 1 bit in the bitmap. For example, a space ratio of 2 means that the data structure consumes 2 bits for every bit inserted, therefore if a 100 megabits bitmap is inserted, the data structure uses 200 megabits, using 100 megabits in the data management structure of the tree and 100 megabits in the bitmaps.

We tested two different bitmap creation methods, the sequential bitmap insertion and the random bitmap insertion. The sequential creation simulates the bitmap where the insertions are very close, for example while reading from a file all insertions are in the same position. The buffer stores the last

insertion tree path and since the next insertion is within a close range of the previous, the tree path is not recalculated, see section 3.4.2. The random creation uses random insert positions. This means that very few operations are buffered and the tree is sparser than the one generated by the sequential creation.

4.1.1 Determining the arity of the B+ tree

The tests in this section were designed to determine the most efficient arity for the B+ trees. To determine the best arity are considered the space used, the time spent building the tree and the time necessary for a batch of operations. The space ratio in this section is the number of bits used by the data structure for every bit inserted. The tests in this section were performed over a bitmap with 100 megabits. A batch of operations is 2000 000 Select1, Select0, Rank0 and Rank1 operations, in a total of 10 000 000 operations. The operations are alternated among each other during the batch and the arguments for each operation are random.

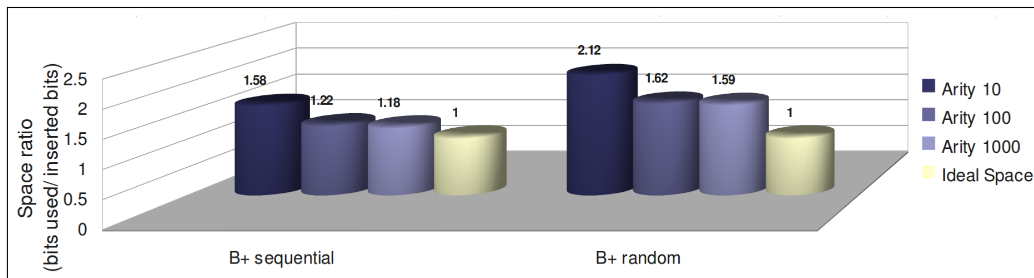


Figure 4.1: The space ratio for bit trees with arity 10, 100 and 1000. The ideal space represents the ratio of 1 bit spent for every 1 bit inserted.

In the first test, see Figure 4.1, the space ratio shows a considerable difference between arity 10 and arity 100, however this difference is less significant between arity 100 and arity 1000. Space is a very important measure of performance of dynamic FCST, hence an arity of 100 or 1000 is preferable.

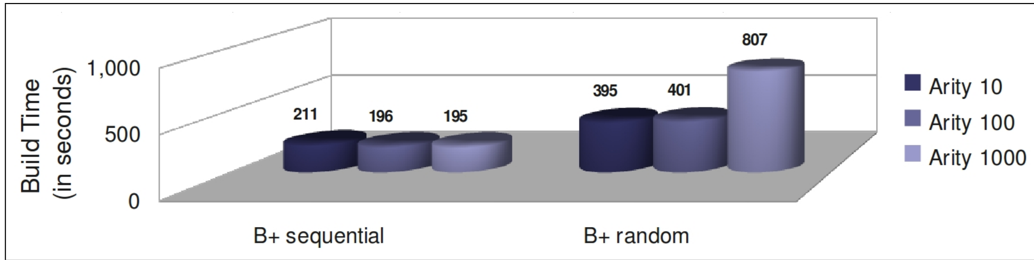


Figure 4.2: Time spent building a 100 megabit bit tree.

The second test, see Figure 4.2, shows the build time of the bit-tree. This Figure 4.2 shows a large difference between arity 100 and arity 1000 when building the B+ tree with random insertions, arity 100 needs less space during construction when compared to arity 1000 and has a lower space ratio than arity 10.

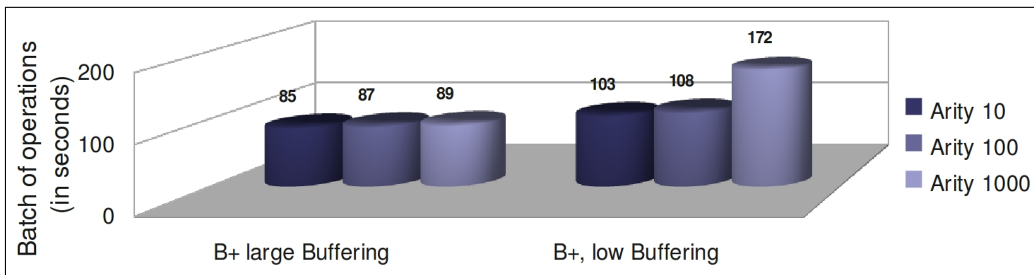


Figure 4.3: Time spent computing a batch of operations on a 100 megabit bit tree. The x axis represents the buffer intensive operations versus the low buffering operations.

In the last test, see Figure 4.3, the difference between arity 100 and arity 1000 is seen when there are few successfully buffered operations. Arity 100 is faster at building a random bitmap than arity 1000, which has a much lower space ratio than arity 10 and is faster for operations than arity 1000. Therefore in this dissertation the B+ trees use arity 100 for the remainder of the tests.

4.1.2 Performance

The performance of the B+ bit tree implemented during this dissertation is compared to the Red Black bit tree developed by Gerlarch. The implementation by Gerlach is based on Mäkinen and Navarro [15], and Sadakane et.al [11], however the bit tree implemented in this dissertation is based Chan et. al [3] and Mäkinen and Navarro [15].

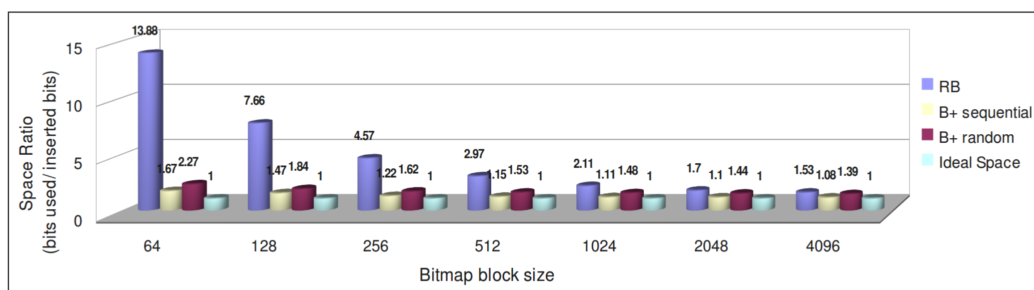


Figure 4.4: The space ratio for the RB bit tree and the B+ bit tree. The RB is the red black bit tree implemented by Gerlach. The ideal space represents the ratio of 1 bit spent for every 1 bit inserted.

The space ratio test, figure 4.4, shows a logarithmic curve relating the space used by the bit trees and the bitmap block size, however the logarithmic curvature of the red RB tree is larger than the curvature of the B+ bit tree. Therefore the B+ is always in advantage although both bit trees have similar space ratios as the block size increases. Block sizes larger than 4096 are not tested because they are very slow and therefore not interesting for the implementation of the dynamic FCST.

The construction time degrades with a logarithmic curve as the block size increases. The sequential and random B+ builds are very different when the block size is smaller than 512, as sequential constructions use less than half the time of a random construction. The experiments show, consistently, that the B+ can be built in less time than the red black.

The dynamic suffix tree has many situations where the insertions are sequential. For example, more than half the insertions in Bitmap B are leaf

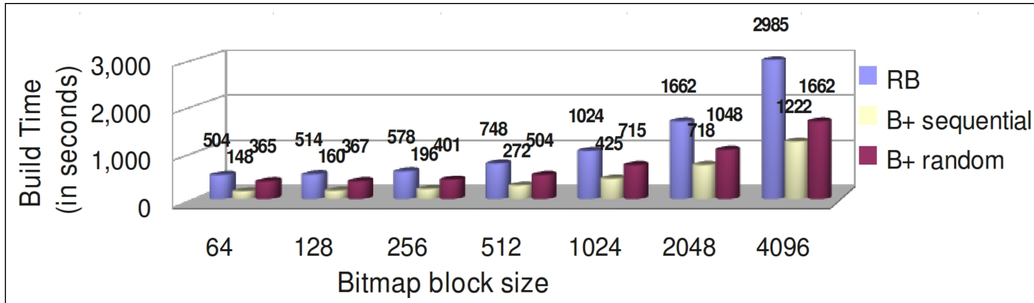


Figure 4.5: The time in seconds, y axis, is the time necessary to build a 100 megabit bit tree. On the right is the B+ random and sequential. The RB is the red black bit tree implemented by Gerlach.

samplings, notice that inserting a leaf in Bitmap B involves inserting two bits in sequential positions. Therefore the time for constructing a dynamic FCST will be somewhere between the worst case possible that is a random B+ build and the best case of a sequential B+ build.

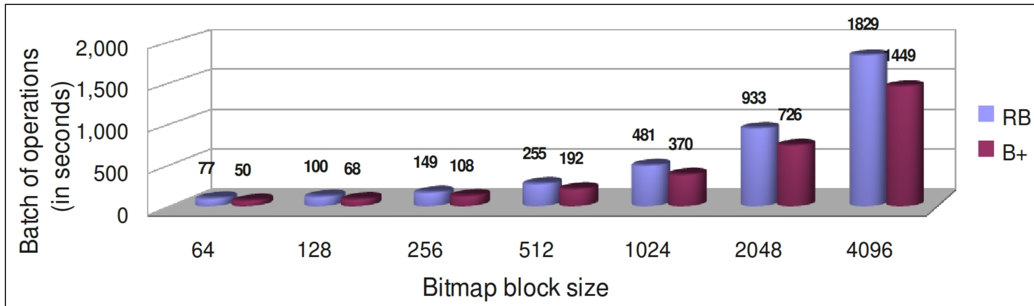


Figure 4.6: In the y axis is the time in seconds spent to complete a batch of operations. The x axis represents size of the bitmap blocks, from 64 bits to 4096 bits. RB is the red black bit tree implemented by Gerlach.

The operations time shows that it is much faster to use smaller bitmap block size than larger blocks, it also shows that the B+ is always faster than the RB for each block size. In every test, both in speed, build time or space, and for every block size, the B+ bit tree is more efficient, therefore it is the selected structure to use in the dynamic FCST.

4.1.3 Comparing Results

This section compares three possible configurations of the bit trees presented in figure 4.7, for each configuration there is a criteria: smallest space ratio, under the title "Lowest Space Ratio Test"; similar ratio, under the title "Balanced Space Ratio Test"; fastest operations, under the title "Speed Test".

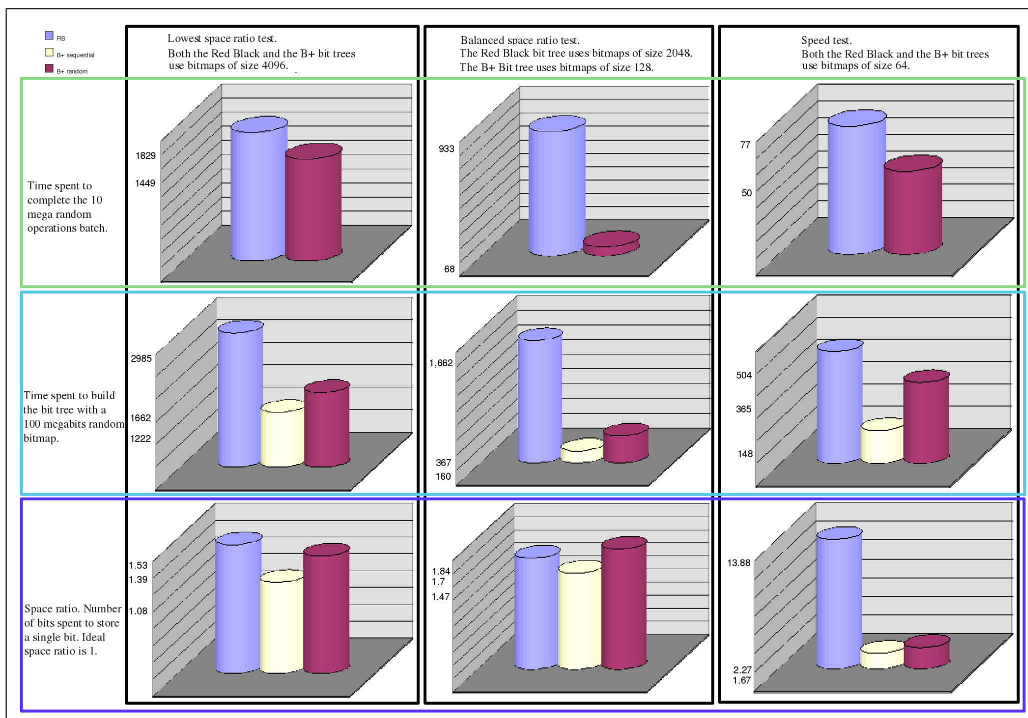


Figure 4.7: From left to right, lowest space ratio configuration compares the bit trees at their best space ratio, balanced ratio test compares the bit trees at a usable space ratio, the speed test compares both bit trees for their fastest configuration. Time values are in seconds.

The smallest ratio the bit trees achieve, in the tests, occurs when the block size is 4096. The bottom of the left column in Figure 4.7 shows that the B+ has a lower space ratio and therefore should be selected under this criteria. It also is faster to build and faster for operations. However this large block size has slow operations, therefore in the dynamic suffix tree smaller bitmap

blocks are used.

The best operation times occur when the block size is 64. The right column in figure 4.7 has the results of this test, notice that the B+ is faster for operations and build time. In the bottom of figure 4.7 notice the very large difference between the space ratios, it shows that there is a large space cost when the RB speed increases, however it exists a smaller cost for the B+. The balanced criteria demands block sizes where the space ratio value is smaller than 2. Space is a decisive factor for the dynamic FCST therefore a bit tree that occupies more than twice the size of the inserted bit array is not acceptable. The block size with a ratio under 2 for the RB that achieves the fastest operation time is the 2048 block size. The B+ with the closest ratio inferior to the RB, with a 2048 block size, is the 128 bits block. Therefore these two configurations are compared in the middle column of figure 4.7.

The operations time is very different, the RB used 933 seconds to complete the operations batch while the B+ needed 68 seconds, therefore the B+ uses 7% of the time used by the RB. The build time of the RB is 1662 and the build time for the B+ is in the worst case 367 and in the best case 160, therefore it uses between 22% and 9% of the RB build time.

The space ratio of the largest bitmap block size in the B+ uses 91% of the equivalent RB bit tree, the smallest bitmap block size of the B+ uses 16% of the equivalent RB bit tree. The speed for consult operations in the B+ is 64% of the RB in the smallest bitmap block size and 79% for the largest bitmap bock size. The build time for the B+ smallest bitmap is 72% of the equivalent RB, and the largest bitmap block size build time is 55% of the equivalent RB.

4.1.4 Dynamic Environment Results

The implemented B+ bit tree is the chosen structure for the dynamic bit arrays in this dissertation, however a test of the performance under stress conditions using a mix of insertions and deletions is required determine if the tree loses performance. Figure 4.8 presents the results of evaluating the random environment where for every two random insertions one random deletion occurs.

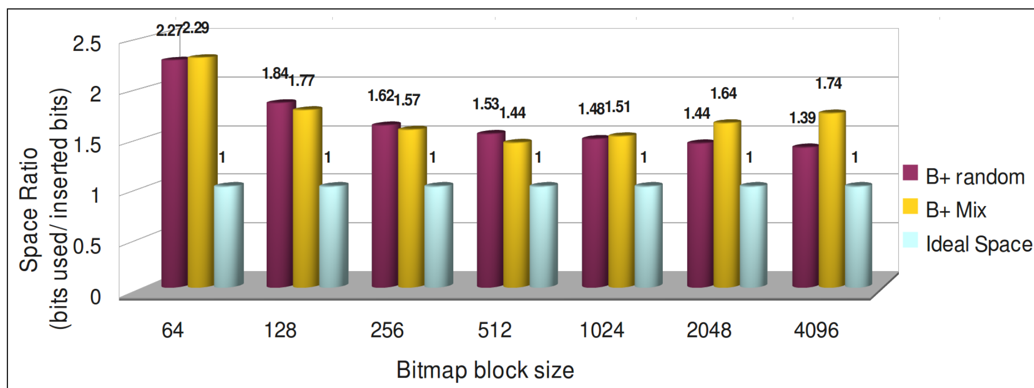


Figure 4.8: The figure shows the graphic comparing the construction without deletions and the construction with alternated deletions.

The performance of the bit tree maintains the same level for the mixed test and the random test for bitmaps up to 2048 block size. The bit trees with blocks 2048 and 4096 have a lower occupation rate because they have very large leafs allocated, the large leafs do not often contract when deletions occur.

4.2 Parentheses Bit Tree

This section compares the parentheses tree to a simulated tree that performs the same operations. A tree was not developed for the purpose of this test however the space use is simulated as realistically as possible. For example,

the test involves 1000 000 nodes in each tree, therefore the test measures the space used by the allocation of 1000 000 nodes with the characteristics of the nodes used for a tree with the required DFCST operations. The simulated tree structure is not designed to support the Rank and Select operations that the parentheses bit tree supports. The node structure has a array of child pointers, each suffix tree has at least two children. Therefore the child counter is initialized at two:

- unsigned long: Child Counter = 2
- Node**: Child Nodes = new Node[2]
- Node*: father
- unsigned long: node identifier

These tests do not have a comparison with another dynamic parentheses bit tree structure because there is no other know implementation, therefore the tests are directed at space performance of two optimizations implemented.

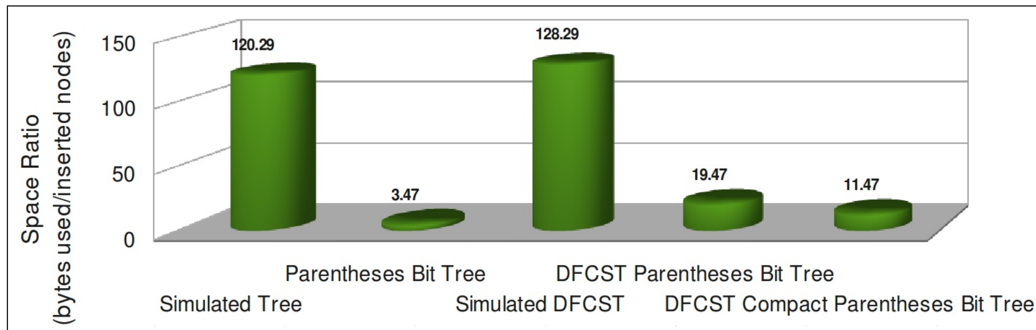


Figure 4.9: The figure shows a graphic with the number of bytes per node ratio for a simulated tree, the same tree stored in a parentheses tree data structure, a simulated tree for the DFCST, the DFCST stored in a parentheses tree data structure, the compact version of the DFCST parentheses tree.

The results show that a parentheses tree uses 2.9% of the space used by a simulated tree with the essential functions. The DFCST requires 2 integers

to store data on the string depth and a link counter per node. Therefore a tree for the DFCST would require extra 8 bytes, 128.3 bytes, the normal parentheses tree uses 15.2% of this space and the compact parentheses bit tree uses 8.9%.

4.3 Wavelet Tree

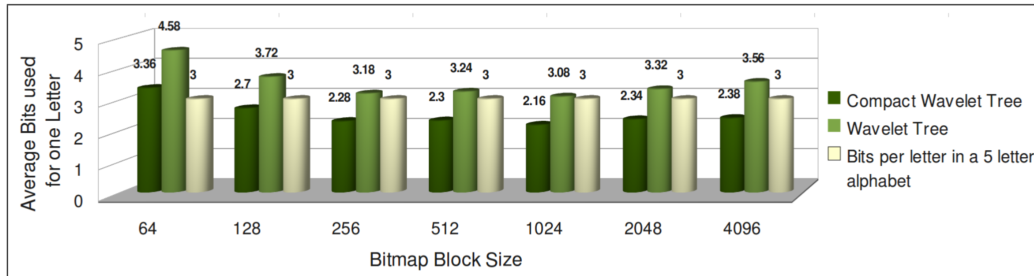


Figure 4.10: The figure shows the graphic comparing the space used for each inserted letter, in a test with 100 megas of DNA, using the normal and the compact wavelet trees.

To compare the wavelet trees performance the test uses a batch of operations, each operation has random parameters. A total of 1000000 iterations over each possible operation was executed and measured. The space test used 100 megas of DNA and was measured using the Windows Management Console.

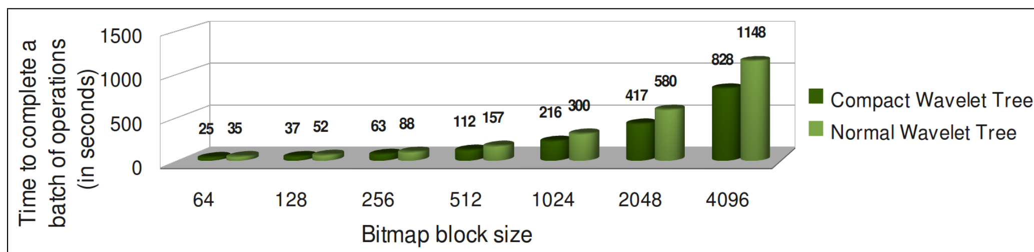


Figure 4.11: The figure shows the graphic comparing the speed needed to complete a batch of operations with the normal and the compact wavelet trees.

The results of the compact wavelet tree show that it uses on average 29% less space and 28% less time than the normal wavelet tree. The compact wavelet tree is smaller and faster than the normal wavelet tree for every bitmap block size, therefore the compacted wavelet tree is used in the DFCST prototype.

4.4 DFCST

This section presents the data obtained in performance tests of the DFCST. First in section 4.4.1 the tests results demonstrate that the DFCST achieves very different performances according to the specified sampling or the bit trees specifications. Afterwards the space usage and operations time of the DFCST is showed and compared with the FCST. The space ratio in this section is the number of bytes used by the compressed suffix tree divided by the number of inserted letters.

4.4.1 Space used by the DFCST

The space use is fundamental for the DFCST, and the advantage of understanding and determining the performance of the DFCST lies with the sampling distance and the wavelet tree bitmap size configuration. Figure 4.12 shows the relation between the size of the sampled tree and the wavelet tree as the sampling distance increases. The space used decreases as the sampling increases, since fewer nodes are stored in the sampled tree the parentheses tree uses less space. The figure 4.12 also shows that as the bitmap sizes in the wavelet tree increase, the space used for the wavelet tree decreases. This is in accordance to previous tests results that indicate that bit trees with larger bitmaps use less space, see figure 4.8.

4.4.2 The DFCST operations time

The time for each operation depends mainly on the sampling distance δ and the wavelet tree configuration. Therefore it is necessary to study the variation caused by the different possible choices. The results show that the increase of the sampling distance causes a deterioration of the operations speed.

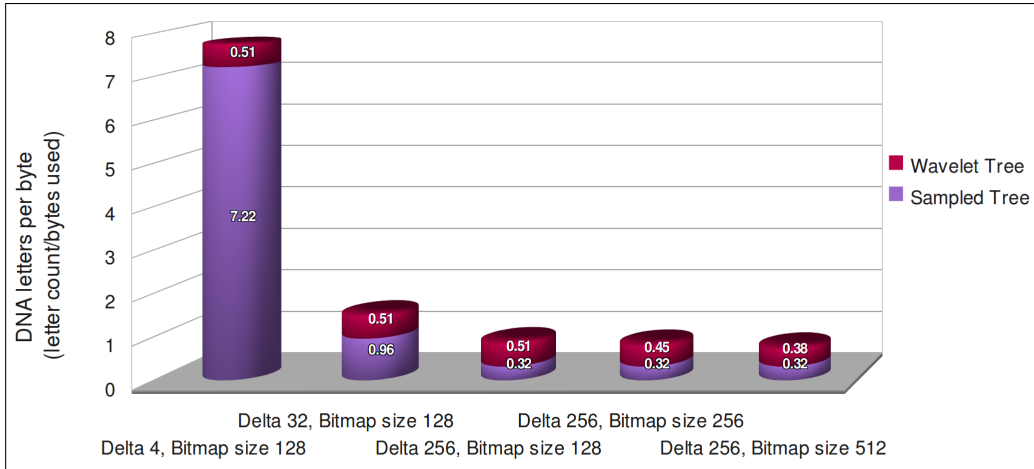


Figure 4.12: DFCST space in bytes for each DNA letter. Test uses 3 megas of DNA.

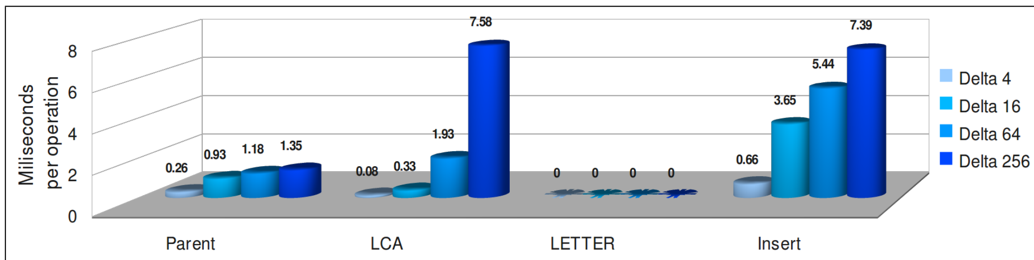


Figure 4.13: DFCST speed results over 3 megas of DNA.

4.4.3 Comparing the DFCST and the FCST

A important objective in this dissertation is the comparison of the DFCST and the FCST. The comparison is presented in this section and has three important aspects, these are the space required to build the data structures, the space used after the data structures are complete and the operations time.

The table 4.2 shows that the operations time are similar in both implementations. The DFCST is expected to be slower than the FCST since all operations are affected by a $O(\log n)$ factor necessary to maintain the structure dynamic. However the results show that this factor has little practical im-

Table 4.2: The table shows DFCST and FCST time to complete a operation in milliseconds. (*) Insertion and removal are not possible in the static FCST.

	DFCST	FCST
Parent	2.195	1.76
SLink	0.441	1.107
LCA	1.454	1.156
LETTER	0.00007	0.0058
SDEP	1.362	0.085
Insert	5.67	*
Remove	6.86	*

pact. The speed similarities occurs because the configuration of the DFCST is using a smaller sampling distance than the FCST. If a larger sampling distance is used in the DFCST the speed will deteriorate, however the space use will improve substantially, see 4.4.1 and 4.4.2.

Table 4.3: The table shows DFCST, FCST and CST space use in megabytes for 50 mega dna letters.

	DFCST	FCST	CST
Final Space	73.2	29.3	161.9
Build Space	73.2	619.2	562.9

The table 4.3 shows that the space used in the construction of the FCST is very large compared to the final space, while the DFCST is very stable and does not exceed the final space. The final space of the DFCST is more than twice the space used by the final FCST, however if a different configuration is used the DFCST can ocupy 35 megabytes. The configuration for 35 megabytes DFCST with 50 mega of dna has a sampling of 256 and a bit tree bitmap block size of 512, see 4.12.

Chapter 5

Conclusions and Future work

5.1 Conclusions

Suffix trees are a key element in the future of large data indexing. Thus, it is important for such large volumes of data to apply compression techniques and the use of dynamic structures. This dissertation implemented a software library of a dynamic compressed suffix trees, following a iterative software development approach. This iterative approach allowed improvements that achieve the desired space performance. From the user point of view, the system creates a suffix tree, with its basic operations plus the operations of inserting and removing letters from a text file. From the programmer view the system software library is modular and there are four data structures that can be integrated in other projects: the DFCST, the B+ Bit Tree, the Compact Wavelet Tree and the B+ Parentheses Tree.

The main contribution and goal of this work is an implementation of the DFCST. Experimental results show that our prototype can obtain the smallest dynamic suffix trees. Compared with static suffix trees it is the second

smallest (after the static FCST). The DFCST does not use prohibitive space during construction like other compressed suffix trees implementations and the final space is considerably low even compared to the smallest compressed suffix trees. Although it is a dynamic data structure, the performance of the operations is very competitive when compared to static compressed suffix trees.

The results show that the space used in the management of the dynamic data structures has a considerable weight compared to the space used by the indexed data itself. Therefore efforts to improve the dynamic data structures can result in large space gains, even larger gains than the effect of compressing the indexed data.

The compact wavelet tree is a data structure that improves the normal wavelet tree. The results show that it performs considerably faster and using less space. This is due to the tree trimming which avoids unnecessary computations and data storage. The results prove that the technique used to compact the wavelet tree is realistic and viable for future implementations.

The bit tree results show that this implementation is a good alternative to known implementations of a dynamic bit array for rank and select. Therefore future projects that require this type of data structure can choose to use this implementation including the NBitTree files in their projects.

The implemented dynamic parentheses tree is the only dynamic data structure implementation of this kind known to us. The results show that it can replace a normal tree structure by a much smaller representation and shows that additional fields can be added with the same space loss as a normal tree data structure has. Therefore it is a possible choice for future projects that require succinct space and dynamic tree capabilities.

The results show that the dynamic fully compressed suffix tree is a good step to achieve a small and dynamic suffix tree. With future improvements it can become a useful tool for further research in bio-informatics.

5.2 Future work

The implemented DFCST has some limitations and can still be improved to obtain better speed and lower space requirements. Faster data structures allow larger bitmaps and therefore lower space use. Smaller space use in data structures, such as the sampled tree, allow a smaller sampling distance resulting in faster computations.

The reverse buffered path lacks the functionality to regress through the internal nodes. To achieve this the method that is being used at the level of bitmap blocks and leaf nodes can used at the internal nodes level and therefore take advantage of this feature.

The use of threads can save time in several operations. In the wavelet tree the use of a CPU with several cores renders higher gains. For example if the alphabet has 5 letters, as the case of the DNA plus the dollar letter, the wavelet tree has three levels. The insert and remove letter operations compute the bitmaps where the bits are modified, which is determined with the letter encoding, furthermore the bit position within the bitmaps is determined iteratively for each level with Rank. However the time dominant operation is the bit shift, therefore since we know the bitmaps and respective bits positions we can delegate the bits insertion to other threads and proceed with the computations. Using one processor for each level will thus reduce speed, in this alphabet case, to one third. Other possible improvement can be used in the insert operation using the weiner algorithm which has operations which do not need to be sequential. The operation can insert simultaneously in the B bitmap and in the wavelet tree, hence saving insertion time.

The use of several processors and the buffered tree path will increase the buffer probability of success. The bit tree can be prepared to store as many buffered paths as requested, for every operation each processor verifies a buffered paths to see it is the result of the current operation. This increases significantly the probability of one buffered path being correct or close to

the result and generate the result in constant time. This also allows a race among threads starting in different positions of the tree and converging to the operation result, the thread that returns first stops the operation.

The parentheses bit tree lacks one optimization which is implemented in the bit tree. This optimization decreases space use significantly and involves allocating all leaf bitmaps within the leaf as a superblock and not as individually allocated blocks. This will save a call to the "new" for each block decrease the space used by each leaf node. The parentheses bit tree does not need to use unsigned short for the management of the blocks, it should use unsigned char and thus reduce the space used with data management structures at the leaf level by nearly half.

To reduce the space used by the DFCST, the bitmaps within the bit and parentheses trees can be compressed. The data structure used to represent the bitmap array are the type bitset, therefore if a interface replaces the bitset with compacted operations the overall project will use the compacted operations. The data structure for the parentheses tree is a array of integers which regularly stores integers whose values are close to 1, therefore it is highly compressible.

If stricter space requirements are needed the bit tree should try to re-compact sparse bitmaps more often. This can be achieved by a algorithm which for every insertion attempts a local merge of bitmaps.

Bibliography

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *WABI '02*, pages 449–463, London, UK, 2002. Springer-Verlag.

[2] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86, 2004.

[3] Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. *ACM Trans. Algorithms*, 3(2):21, 2007.

[4] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398, 2000.

[5] Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. An(other) entropy-bounded compressed suffix tree. In *CPM '08*, pages 152–165, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[7] R. González and G. Navarro. Improved dynamic rank-select entropy-bound structures. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 4957, pages 374–386, 2008.

[8] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *SODA '03*, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[9] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *STOC '00*, pages 397–406, New York, NY, USA, 2000. ACM.

[10] Dan Gusfield. *Algorithms on strings, trees, and sequences : computer science and computational biology.* Cambridge Univ. Press, January 2007.

[11] Wing-Kai Hon, Tak-Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. Compressed index for dynamic text. In *DCC '04: Proceedings of the Conference on Data Compression*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[12] G. Navarro L. Russo and A. Oliveira. Fully-Compressed Suffix Trees. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 4957, pages 362–373, 2008.

[13] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3109, pages 420–433, 2004.

[14] Veli Mäkinen. Compact suffix array. In *COM '00*, pages 305–319, London, UK, 2000. Springer-Verlag.

[15] Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms*, 4(3):1–38, 2008.

[16] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *SODA*, pages 319–327, 1990.

[17] Giovanni Manzini. An analysis of the burrows—wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[18] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *FOCS '97*, page 118, Washington, DC, USA, 1997. IEEE Computer Society.

[19] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[20] Kashyap Dixit Niko Välimäki, Wolfgang Gerlach and Veli Mäkinen. Experimental algorithms, 6th international workshop, wea 2007, rome, italy, june 6-8, 2007, proceedings. In *WEA*, pages 217–228, Washington, DC, USA, 2007. Springer.

[21] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007.

[22] Luís M. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Dynamic fully-compressed suffix trees. In *CPM '08*, pages 191–203, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] Luís M. Russo and Arlindo L. Oliveira. A compressed self-index using a Ziv—Lempel dictionary. *Inf. Retr.*, 11(4):359–388, 2008.

[24] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theor. Comp. Sys.*, 41(4):589–607, 2007.

[25] Peter Weiner. Linear pattern matching algorithms. In *SWAT '73*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.