



UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ESTRUCTURAS COMPRIMIDAS PARA ÁRBOLES DE SUFIJOS

**TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS, MENCIÓN
COMPUTACIÓN**

RODRIGO ANTONIO CÁNOVAS BARROSO

PROFESOR GUÍA:

GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:

JÉRÉMY BARBAY

NIEVES RODRÍGUEZ BRISABOA

RODRIGO PAREDES MORALEDA

Este trabajo fue financiado por el Instituto de Dinámica Celular y Biotecnología (ICDB), Iniciativa Científica Milenio, Proyecto ICM P05-001-F, Mideplan, Chile, y el Proyecto Fondecyt 1-080019, Conicyt, Chile.

SANTIAGO DE CHILE
SEPTIEMBRE 2010

Resumen

El árbol de sufijos es una de las estructuras más importantes que se han creado para el manejo de cadenas de caracteres. Esta estructura tiene muchas aplicaciones en variadas áreas de la investigación, destacándose en la bioinformática. Una implementación clásica de esta estructura ocupa un espacio de memoria muy grande, lo cual hace difícil su uso para problemas de gran tamaño. En este trabajo se presentará una nueva representación comprimida para los árboles de sufijos, la cual ofrece muchas variantes, haciéndola una interesante alternativa con respecto a las implementaciones ya existentes.

En la práctica, además de la presentada en este trabajo, existen dos implementaciones disponibles para un árbol de sufijos comprimido. Ambas implementaciones ofrecen espacio y tiempos completamente diferentes. La primera fue propuesta por Sadakane e implementada por Välimäki et al. Esta estructura soporta la mayoría de las operaciones de navegación en tiempo constante, tomando unas pocas decenas de microsegundos por operación, pero en la práctica requiere entre 25 a 35 bits por símbolo. La segunda implementación fue propuesta e implementada por Russo et al. Las operaciones de navegación se resuelven en el orden de los milisegundos por operación, ocupando muy poco espacio en memoria, alrededor de 4 a 6 bits por símbolo. Esto hace que sea atractiva cuando la secuencia es grande en comparación con la memoria principal disponible, pero la implementación es mucho más lenta que la de Välimäki et al. De hecho los tiempos de navegación de Russo et al. son cercanos al tiempo de acceso que toma almacenar datos en disco.

En este trabajo inicialmente se implementa una propuesta teórica reciente de Fischer et al., demostrando que de su trabajo se obtienen muchas estructuras interesantes que ofrecen tiempos competitivos y requerimientos de memoria accesibles en la práctica. Luego se exploran diferentes variantes y nuevas ideas para todas las estructuras que componen al árbol de sufijos comprimido propuesto por ellos. Una de estas variantes resultó ser superior a la implementación de Sadakane tanto en espacio como en tiempo, utilizando alrededor de 13 a 16 bits por símbolo y tomando unos pocos microsegundos en resolver cada operación. Otra de estas variantes ocupa entre 8 y 12 bits por símbolo y toma unos pocos cientos de microsegundos por operación, siendo todavía varias veces más rápida que la implementación de Russo.

En esta tesis también se implementan y comparan las técnicas actuales más importantes, que usan $2n + o(n)$ bits y toman tiempo constante por operación, para representar árboles generales en forma sucinta. Cabe notar que en la literatura no existe un estudio exhaustivo que compare la eficiencia práctica de estas técnicas. Estas se pueden clasificar en tres grandes tendencias: las basadas en BP (balanced parentheses), las basadas en DFUDS (depth-first unary degree sequence) y las basadas en LOUDS (level-ordered unary degree sequence).

Los experimentos realizados permiten concluir que una técnica reciente para representar árboles generales, basada en los llamados range min-max trees, se destaca como una excelente combinación de espacio, tiempo y funcionalidad, mientras que las otras (en particular LOUDS) siguen siendo interesantes para algunos casos especiales. Este estudio aporta nuevas ideas que fueron aplicadas en la implementación del árbol de sufijos comprimido presentado en esta tesis, con lo cual se obtuvieron implementaciones competitivas e incluso mejores que las ya existentes.



Agradecimientos

Aunque en la portada sólo aparezco yo como autor, en realidad hay muchos cómplices de este crimen. Quisiera agradecer a mis padres, Rodrigo y Silviana, por apoyarme y por ser tan fanáticos y buenos en sus trabajos que me convencieron para que estudiara algo diferente a ellos. A mi hermana Camila por haber nacido y no dejarme ser hijo único. A mis abuelos Silvia, Maruja, René y Cheché quienes siempre mostraron una confianza infinita en mis conocimientos, aunque realmente no sepan mucho sobre lo que estudié.

También quisiera agradecer a esos amigos que me han aguantado casi toda mi vida, Bernardita, María de los Angeles, Pablo Fassi, Nicolás Grisanti e Ignacio Fantini, quienes hasta hoy de repente se toman el tiempo para escucharme alegar. A esos amigos que conocí mientras estaba en la Universidad, Pancho Molina, Rafa Rodríguez, Gonzalo Dávila, Julio Villane, Gaston L'Huilier y Alan Shomaly, quienes hicieron que mi paso por la Universidad fuera mucho más grato. A la gente de BT y Química con quienes más de una vez pude desestresarme saliendo a tomar, comer o simplemente juntándose para pasar el rato.

Agradezco a todos aquellos que me ayudaron en el desarrollo de esta tesis, ya sea en su desarrollo como en la redacción de este documento: Johannes Fischer, Diego Arroyuelo, Simon Puglisi, Luis Russo, Kunihiko Sadakane, Susana Ladra, Rodrigo González, Juha Kärkäinen, Veli Mäkinen, Francisco Claude, Sebastian Kreft, Iván López y Adriana López. A los miembros de mi comisión, Nieves Brisaboa, Jérémy Barbay y Rodrigo Paredes quienes realmente se dieron el tiempo para leer completamente esta tesis, entregándome comentarios de gran utilidad para la realización del documento final.

Hay varios que ya he mencionado pero los mencionaré nuevamente por su importancia en mi vida: Pablo, con quien he podido siempre contar y a quien siempre consideraré como mi hermano; Panky, quien hace que despertarse cada día valga la pena, quien me hace feliz; Claude, quien ha sido un gran amigo y fue uno de los que me convenció que me dedicara al área de Algoritmos; Pancho, quien siempre está dispuesto a hablar conmigo.

Especialmente quiero agradecer a Gonzalo Navarro, quien más allá de ser mi profesor guía, fue un gran amigo y apoyo, gracias a él he encontrado que la investigación puede ser divertida. Además debo agradecerle por la paciencia infinita (aunque esto se lo agradezco a Martina, Aylén y Facundo), por cada asado al cual fui invitado y por cada consejo que me haya dado tanto para mi trabajo como investigador como para mi futuro.

Hay millones de personas a quienes me gustaría agradecer, pero lamentablemente nunca he sido muy bueno para escribir lo que siento, por lo tanto espero no haber dejado a nadie afuera y si alguien encuentra que lo omití o que debería haber dicho algo más... para la próxima tesis será =).

*Rodrigo Cánovas
Yigo*



UNIVERSITY OF CHILE
FACULTY OF PHYSICS AND MATHEMATICS
DEPARTMENT OF COMPUTER SCIENCE

COMPRESSED DATA STRUCTURES FOR SUFFIX TREES

SUBMITTED TO THE UNIVERSITY OF CHILE IN FULFILLMENT OF THE
THESIS REQUIREMENT TO OBTAIN THE DEGREE OF MSC. IN
COMPUTER SCIENCE

RODRIGO CÁNOVAS

ADVISOR:

GONZALO NAVARRO

COMMITTEE:

JÉRÉMY BARBAY
RODRIGO PAREDES

NIEVES BRISABOA
(University of A Coruña, Spain)

This work has been supported by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile, and by Fondecyt Project 1-080019, Conicyt, Chile.

SANTIAGO - CHILE
SEPTEMBER 2010

Abstract

The suffix tree is a powerful data structure capable of solving many problems in stringology, with many applications in computational biology. Classical implementations require much space, which renders them useless for large problems. In this work, we explore practical implementations of compressed suffix trees, introducing several variants that offer an interesting alternative to the existing proposals.

In practice, besides the ones presented in this work, there are two other compressed suffix tree implementations available, both offering a completely different space-time trade-off. The first one was proposed by Sadakane and implemented by Välimäki et al. This structure supports most of the tree navigation operations in constant time. It achieves a few tens of microseconds per operation, but in practice it requires about 25–35 bits per symbol. The second was proposed and implemented by Russo et al. The navigation operations are supported in order of milliseconds per operation and achieve very little space, around 4–6 bits per symbol. This makes it attractive when the sequence is large compared to the amount of main memory available, but it is much slower than Välimäki et al.’s implementation of Sadakane’s structure. Indeed the navigational times in Russo et al.’s structure are close to the access time when storing the data on disk.

We present practical implementations of a recent theoretical proposal by Fischer et al., showing that they yield many interesting structures that offer both practical times and affordable spaces. We first implement verbatim Fischer et al.’s proposal, then we try different variants and new ideas for all the structures that compose the compressed suffix tree presented by them. One of these variants turn out to be superior to Sadakane’s compressed suffix tree implementation in both space and time, using around 13 to 16 bits per symbol and requiring a few microseconds per operation. Another alternative works within 8–12 bits per symbol and requires a few hundreds of microseconds per operation, being still several times faster than Russo’s implementation.

In this thesis we also implement and compare the major current techniques for representing general trees in succinct form using $2n + o(n)$ bits and implementing the navigation operations in constant time. There was no exhaustive study in the literature comparing the practical performance of such techniques. These can be classified into three broad trends: those based on BP (balanced parentheses in preorder), those based on DFUDS (depth-first unary degree sequence), and those based on LOUDS (level-ordered unary degree sequence).

We experiment over various types of real-life trees and of traversals, and conclude that a recent technique for representing general trees based on so-called range min-max trees stands out as an excellent practical combination of space occupancy, time performance, and functionality; whereas others (particularly LOUDS) are still interesting in some limited-functionality niches. This study yields new ideas that were applied in other areas of our compressed suffix tree implementation, and that finally made it competitive or even better than the existing ones.

Contents

1	Introduction	1
1.1	Outline and Contributions of the Thesis	3
2	Basic Concepts	4
2.1	Empirical Entropy	4
2.2	Succinct Data Structures	5
2.3	Rank and Select on Sequences	5
2.3.1	Binary Sequences	5
2.3.2	Arbitrary Sequences	7
2.4	Tries or Digital Trees	8
2.5	Suffix Trees	9
2.6	Suffix Arrays (SA)	11
2.6.1	Sadakane’s Compressed Suffix Array (CSA)	12
2.6.2	The FM-Index	14
2.7	Longest Common Prefix (LCP)	16
3	Succinct Tree Representations	17
3.1	Trees and their Representation	18
3.2	Balanced Parentheses	21
3.2.1	Hash-Based Heuristic (HB)	21
3.2.2	Recursive Pioneer-Based Representation (RP)	22
3.2.3	Range Min-Max-Tree Based Representation (RMM)	23
3.3	Succinct Tree Representations	24
3.3.1	Preorder Balanced Parentheses (BP)	24
3.3.2	Depth-First Unary Degree Sequence (DFUDS)	25
3.3.3	Level-Ordered Unary Degree Sequence (LOUDS)	26
3.3.4	A Fully-Functional Representation (FF)	27
3.4	Experimental Comparison	27
3.4.1	Core Operations	28
3.4.2	Tree Operations	31
3.4.3	Asymptotics	35
3.4.4	Non-Succinct Tree Representations	36

4	Compressed Suffix Trees	39
4.1	Sadakane's Compressed Suffix Tree	39
4.2	Russo et al.'s Compressed Suffix Tree	42
4.3	Fischer et al.'s Compressed Suffix Tree	45
5	Representing the Array LCP	47
5.1	Sadakane's LCP (Sad)	47
5.2	Fischer et al.'s LCP (FMN)	49
5.3	Puglisi and Turpin's LCP (PT)	50
5.4	Kärkkäinen et al.'s LCP (PhiSpare)	52
5.5	Directly Addressable Codes (DAC)	52
5.6	Re-Pair LCP (RP)	54
5.7	Experimental Comparison	54
6	Computing NSV/PSV/RMQ (NPR)	58
6.1	Fischer et al.'s NPR	58
6.2	A Novel Practical Solution for NPR	59
6.3	Experimental Comparison	61
7	Our Compressed Suffix Tree	66
7.1	Comparing the CST Implementations	67
7.2	Experimental Comparison	68
8	Epilogue	78
8.1	Breaking News	78
8.1.1	Wee LCP	78
8.1.2	Range Minimum Queries without LCP Accesses	79
8.1.3	CST++	80
8.2	Conclusion and Future Work	80
	Bibliography	83

List of Figures

2.1	Example of rank, select, and access	6
2.2	Example Wavelet Tree	7
2.3	Example of a Trie	8
2.4	Example of a Suffix Tree	10
2.5	Example of a Suffix Array	11
2.6	Comparing P against $T_{A[i],n}$	13
2.7	Example of the BWT	15
2.8	Counting on FM-Indexes	15
3.1	Space/time for the core operations	29
3.2	Space/time for the core operations on parenthesis structures, for $p=1.0$	31
3.3	Performance for operation $\text{parent}(x)$	32
3.4	Performance for operation $\text{child}(x,i)$	33
3.5	Performance for operation $\text{labeled_child}(x,s)$	34
3.6	Performance for operation $\text{level_ancestor}(x,d)$	34
3.7	Performance for operation $\text{lca}(x,y)$	35
3.8	Performance as a function of the tree size, for random binary trees	36
4.1	Sadakane's CST	40
5.1	Example of how H is computed and used	48
5.2	Computing arrays S and S'	51
5.3	Computing L , using S , S' , and D'	51
5.4	Scheme PhiSpare LCP	52
5.5	Example of LCP-DAC.	53
5.6	Space/time for randomly accessing the LCP array	55
5.7	Space/time for random sequential accesses to the LCP array	57
6.1	Space/time for the operation NSV using $Sad-Gon$ and $FMN-RRR$	62
6.2	Space/time for the operation RMQ using $Sad-Gon$ and $FMN-RRR$	63
6.3	Space/time for the operation NSV using DAC and $DAC-Var$	64
6.4	Space/time for the operation and RMQ using DAC and $DAC-Var$	65

7.1	Space/time trade-off performance for the operation <i>Parent</i>	68
7.2	Space/time trade-off performance for the operation <i>TDepth</i>	69
7.3	Space/time trade-off performance for the operation <i>SLink</i>	70
7.4	Space/time trade-off performance for the operation <i>LCA</i>	71
7.5	Space/time trade off performance for the operation <i>SDepth</i>	72
7.6	Space/time trade-off performance for the operation <i>Child</i>	73
7.7	Space/time trade-off performance for the operation <i>Letter</i>	74
7.8	Space/time trade-off performance for a full traversal of the tree	75
7.9	Space/time trade-off performance for the operation $LAQ_S(d)$	76
7.10	Space/time trade-off performance for the operation $LAQ_T(d)$	77

List of Tables

2.1	Complexity for binary rank and select	6
2.2	Table of Suffix Tree operations	11
3.1	Operations on parentheses and trees	19
3.2	Constant-time reduction of tree operations	24
3.3	Trees tested	28
3.4	Default space usage for the representations	32
3.5	Space usage for the larger representations with minimum functionality	37
5.1	Table of texts	54

Table of Global Symbols

Symbol	Details	Page
Σ	The alphabet.	4
σ	The alphabet size, $\sigma = \Sigma $.	4
H_0	The zero-order empirical entropy.	4
H_k	The k-context empirical entropy.	4
$\$$	End marker, smaller than other symbols, $\$ \in \Sigma$.	8
A	A suffix array.	11
A^{-1}	The inverse function of the suffix array A .	12
$\Psi(i)$	Permutation built on A , $A[\Psi(i)] = A[i] + 1$.	12
$LF(i)$	The inverse function of Ψ .	12
CSA	(Sadakane's) Compressed suffix array	12
LCP	The longest common prefix array.	16
rmq	The range minimum data structure	24
CST	The compressed suffix tree	39
NSV	The next small value function over the LCP array	45
PSV	The preview small value function over the LCP array	45
RMQ	The range minimum querie function over the LCP array	45
NPR	The data structure that include NSV , PSV , and RMQ operations	58

Chapter 1

Introduction

The *suffix tree* [Wei73, McC76] is arguably the most important data structure for string analysis. It has been said to have a myriad of virtues [Apo85] and there are even books dedicated to its applications in areas like bioinformatics [Gus97]. A *suffix tree* is a data structure that represents all the suffixes of a given string in a way that allows answering many string problems.

The classic application for suffix trees is *exact string matching*. Given a text T of length n and a string P of length $m \leq n$, a suffix tree structure can find an occurrence of P in T (or determine that P is not contained in T) in time $O(m)$. In the same time it can give the number of occurrences of the pattern P in the text, and it takes $O(m + k)$ time to give all the occurrence positions in the text, where k is the number of occurrences of the pattern.

Many other complex sequence analysis problems are solved through sophisticated traversals over the suffix tree, and thus a fully-functional suffix tree implementation supports a variety of *navigation operations*. These involve not only the classical tree navigation operations (parent, child) but also specific and sophisticated ones such as suffix links and lowest common ancestors.

One serious problem of suffix trees is that they take much space. A naive implementation can easily require 20 bytes per character of T , and a very optimized one reaches 10 bytes [Kur99]. A way to reduce this space to about 4 bytes per character is to use a simplified structure called a *suffix array* [MM93], but this does not contain sufficient information to carry out all the complex tasks suffix trees are used for. Enhanced suffix arrays [AKO04] extend suffix arrays so as to recover the full suffix tree functionality, raising the space to about 6 bytes per character in practice. Some other heuristic space-saving methods [MRR01] achieve about the same space.

To have an idea of how bad is this space, consider that, on DNA, each character could be encoded with 2 bits, whereas the alternatives we have considered require 32 to 160 bits per character. Using suffix trees on secondary memory makes them orders of magnitude slower as most traversals are non-local. This situation is also a heresy in terms of Information Theory: whereas the information contained in a sequence of n symbols over an alphabet of size σ is

$n \log_2 \sigma$ bits in the worst case, all the alternatives above require $\Theta(n \log n)$ bits.

Recent research on *compressed suffix trees (CSTs)* has made much progress in terms of reaching space requirements that approach not only the worst-case space of the sequence, but even its information content. All these can be thought of as a *compressed suffix array (CSA)* plus some extra information that encodes the tree topology and *longest common prefix (LCP)* information.

One of the first such proposals was by Sadakane [Sad07a]. It requires $6n$ bits on top of his *CSA* [Sad03], which in turn requires $nH_0 + O(n \log \log \sigma)$ bits, where H_0 is the zero-order entropy of the sequence. This structure supports most of the tree navigation operations in constant time (except, notably, going down to a child, which is an important operation). It has recently been implemented by Välimäki et al. [VGDM07]. They achieve a few tens of microseconds per operation, but in practice the structure requires about 25–35 bits per symbol (close to a suffix array), and thus its applicability is limited.

The second proposal was by Russo et al. [RNO08b]. It requires only $o(n)$ bits on top of a *CSA*. By using an *FM-index* [FMMN07] as the *CSA*, one achieves $nH_k + o(n \log \sigma)$ bits of space, where H_k is the k -th order empirical entropy of the sequence, for sufficiently low $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$. The navigation operations are supported in polylogarithmic time (at best $\Theta(\log n \log \log n)$ in their paper). This structure was implemented by Russo and shown to achieve very little space, around 4–6 bits per symbol, which makes it attractive when the sequence is large compared to the available main memory. On the other hand, the structure is much slower than Sadakane’s. Some navigation operations take the order of milliseconds, which starts to approach disk operation times.

Both existing implementations are unsatisfactory in either time or space (though certainly excel on the other aspect), and become very far extremes of a trade-off: either one has sufficient main memory to spend 30 bits per character, or one has to spend milliseconds per navigation operation.

In this thesis we present a third implementation, which offers a relevant space/time trade off between these two extremes. One variant shows to be superior to the implementation of Sadakane’s *CST* in *both* space and time: it uses 13–16 bits per symbol (i.e., half the space) and requires a few microseconds (i.e., several times faster) per operation. A second alternative works within 8–12 bits per symbol and requires a few hundreds of microseconds per operation, that is, smaller than our first variant and still several times faster than Russo’s implementation.

Our implementation is based on a third theoretical proposal, by Fischer et al. [FMN09], which achieves $nH_k(2 \max(1, \log_2(1/H_k)) + 1/\epsilon + O(1)) + o(n \log \sigma)$ bits (for the same k as above and any constant $\epsilon > 0$) and navigation times of the form $O(\log^\epsilon n)$. Fischer et al.’s compressed suffix tree is built over a representation of the *LCP* array and expresses all its operations using *range minimum queries (RMQ)* and a novel primitive called *next/previous smaller value (NSV/PSV)* on the *LCP* array. This work [FMN09] contains several theoretical structures and solutions, whose efficient implementation was far from trivial, and required significant algorithm engineering that completely changed the original proposal in some cases. We studied both the original and our

proposed variants, and came up with the two trade-offs mentioned above.

1.1 Outline and Contributions of the Thesis

Chapter 2 : We describe all basic concepts relevant to the practical study of the compressed suffix tree representations presented in this work.

Chapter 3 : We present an exhaustive comparison of the best techniques, which use $2n + o(n)$ bits, to represent general trees of n nodes. This chapter was published in the *Workshop on Algorithm Engineering and Experiments (ALENEX)* [ACNS10].

Chapter 4 : We explain the practical related work on compressed suffix tree representations.

Chapter 5 : We explore different alternatives to represent the *LCP* array, and compare them experimentally in terms of space and time.

Chapter 6 : We introduce a new idea, inspired in Sadakane and Navarro's succinct tree representation [SN10], to compute operations *NSV/PSV* and *RMQ* over the *LCP* array. This idea differs markedly from the original theoretical proposal presented by Fisher et al., and proved to be far superior in practice.

Chapter 7 : We present a new practical *compressed suffix tree* representation which offers very relevant space/time trade-offs. This new structure is the result of combining the best structures of Chapters 5 and 6 plus a compressed representation for the suffix array. This work was published in the *9th International Symposium on Experimental Algorithms (SEA)* [CN10].

Chapter 8 : We give our conclusions and further lines of research that could be explored based on the results obtained in this work.

Chapter 2

Basic Concepts

In this section we introduce some basic concepts that will be useful for the good understanding of the next chapters. We will call the string S a sequence of characters. The alphabet is usually called Σ and its size $|\Sigma| = \sigma$, and it is assumed to be totally ordered. The length (number of characters) of S is denoted $|S|$. Let $n = |S|$, then the characters of S are indexed from 1 to n , so that the i -th character of S is S_i or $S[i]$. A substring of S is written $S_{i,j} = S_i S_{i+1} \dots S_j$. A prefix of S is a substring of the form $S_{1,j}$, and a suffix is a substring of the form $S_{i,n}$. For what remains of this work all logarithms are in base 2 unless stated otherwise, and we assume $0 \log 0 = 0$.

2.1 Empirical Entropy

A common measure of the compressibility of a sequence is the empirical entropy. For a text T of length n with an alphabet Σ of size σ , the zero-order empirical entropy, H_0 , is the average number of bits needed to represent a symbol of T if each symbol receives always the same code. Mathematically this is defined as follows:

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

where n_c is the number of occurrences of the symbol c in T .

If we can encode each symbol depending on the context in which it appears, we can achieve better compression ratios. For example, if we consider the context th in English, it is likely to find an a, e, i, o or u following th , but it is very unlikely to find a k . The definition of empirical entropy can be extended to consider contexts as follows [Man01]:

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

where T^s is the sequence of symbols preceded by the context s in T . This is called the “ k -Context Empirical Entropy” . It can be proved that $0 \leq H_k(T) \leq H_{k-1}(T) \leq \dots \leq H_0(T) \leq \log \sigma$.

2.2 Succinct Data Structures

A *succinct data structure* is a compressed representation of a structure that uses an amount of space close to the information theoretic lower bound, together with efficient algorithms for simulating its operations.

As an example, consider a binary tree of n nodes. It is possible to represent the tree succinctly using about $2n$ bits and support operations over any node like finding its parent or its left or right child in constant time, without the need to decompress the structure. This is succinct because the number of different binary trees with n nodes is $\binom{2^n}{n}/(n+1)$. For large n , this is about 4^n ; thus we need at least about $\log(4^n) = 2n$ bits to encode it.

An important example of succinct data structures are the compressed indexes, which take advantage of the regularities of the text to operate in space proportional to that of the compressed text. An even more powerful concept is that of a *self-index*, which is a compressed index that, in addition to providing search functionality, contains enough information to efficiently reproduce any text substring. A self-index can therefore replace the text.

In this thesis, as in most work on succinct data structures, we will assume that the computation model is the *word RAM* with word length $\Theta(\log n)$ in which arithmetic and logical operations on $\Theta(\log n)$ -bit integers and $\Theta(\log n)$ -bit memory accesses can be done in constant time.

2.3 Rank and Select on Sequences

One of the most basic structures used in the compressed data structures presented in this thesis is the sequence of symbols supporting *rank* and *select*. Given a sequence S of symbols $rank_a(S, i)$ counts the number of occurrences of character a until position i and $select_a(S, i)$ finds the position of the i -th occurrence of a in the sequence. These structures also support the operation $access(S, i)$ that returns the symbol at position i in the sequence. The most basic case is when the sequence is binary.

2.3.1 Binary Sequences

Let $B[1, n]$ be a 0, 1 string (bitmap) of length n and assume there are m ones in the sequence. Note that $rank_b(B, select_b(B, i)) = i$ and $select_b(B, rank_b(B, i)) \leq i$, where $b \in \{0, 1\}$. Figure 2.1 shows an example of rank and select queries.

There exist many succinct data structures that answer *rank*, *select*, and *access* for the binary

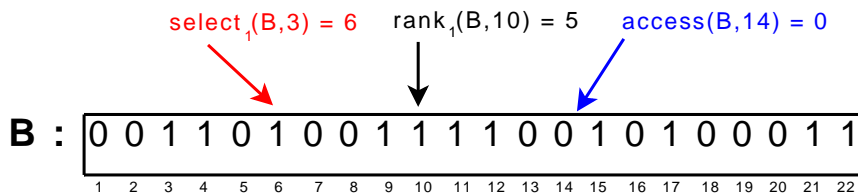


Figure 2.1: Example of rank, select, and access.

case. The first solution capable of answering these queries in constant time using $n + o(n)$ bits [Mun96, Cla98], used n bits for B itself and $o(n)$ additional bits for the data structures answering *rank* and *select* queries. This solution was implemented and studied in practice by González et al. [GGMN05]. Their data structure supports *rank* in $O(1)$ and *select* in $O(\log n)$ time, while requiring 5% extra space over the sequence. The idea is to store *rank* values, in 32 bits, every $s = 20 \cdot 32 = 640$ bits, and then scan byte-wise these 640 bits = 80 bytes using a precomputed table. For *select* they binary search the sampled values and finish with a sequential scan as well. We use this implementation under the name *ggmn*.

The space used was improved by Raman et al. [RRR02] to $nH_0(B) + o(n)$ bits, keeping the same times for answering all the queries. For this compressed bitmap we use a recent implementation [CN08], available at Google Code (`libcds`), which requires 27% extra space on top of the entropy of the bitmap, and implements *rank* in constant time and *select* in $O(\log n)$ time. We use this implementation under the name *rrr*.

Several other practical alternatives achieving good results have been proposed by Okanohara and Sadakane [OS07]. For this thesis we will use two of these alternatives: *sdarray*, that requires $m \log \frac{n}{m} + 2m + o(n)$ bits and solves *select* in time $O(\log^4 m / \log n)$ and *rank* in time $O(\log \frac{n}{m} + \frac{\log^4 m}{\log n})$, and *darray*, that requires about the same space as *ggmn* and answers *select* in $O(\log^4 m / \log n)$ and *rank* in constant time. Table 2.1 summarizes the time and space complexities achieved by these solutions.

Structure	Size	Rank	Select
<i>ggmn</i>	$n + o(n)$	$O(1)$	$O(\log n)$
<i>rrr</i>	$nH_0(B) + o(n)$	$O(1)$	$O(\log n)$
<i>sdarray</i>	$m \log \frac{n}{m} + 2m + o(n)$	$O(\log \frac{n}{m} + \frac{\log^4 m}{\log n})$	$O(\frac{\log^4 m}{\log n})$
<i>darray</i>	$n + o(n)$	$O(1)$	$O(\frac{\log^4 m}{\log n})$

Table 2.1: Space in bits and query time achieved by the data structures.

The reason to consider using Okanohara and Sadakane's version is that in practice, while the *rank* queries are slower, their implementations have a better performance for answering *select* queries than the other structures presented. As we will see in the next chapters, some of the studied structures make heavy use of the *select* query, where *darray* and *sdarray* become a better option.

2.3.2 Arbitrary Sequences

Rank, *select* and *access* operations can be extended to arbitrary sequences S drawn from an alphabet Σ of size σ . There exist several solutions for this case, but we will only explain one, the *wavelet tree*.

A *wavelet tree* [GGV03, NM07] is a perfectly balanced tree that stores a bitmap of length n in the root; every position in the bitmap is either 0 or 1 depending on the value of the most significant bit of the symbol in that position in the sequence. In general, wavelet trees are described as dividing alphabet segments into halves. A symbol with a 0 goes to the left subtree and a symbol with a 1 goes to the right subtree. This decomposition continues recursively with the next highest bit, and so on. The tree has σ leaves and requires $n \lceil \log \sigma \rceil$ bits, n bits per level. Every bitmap in the tree must be capable of answering *access*, *rank* and *select* queries.

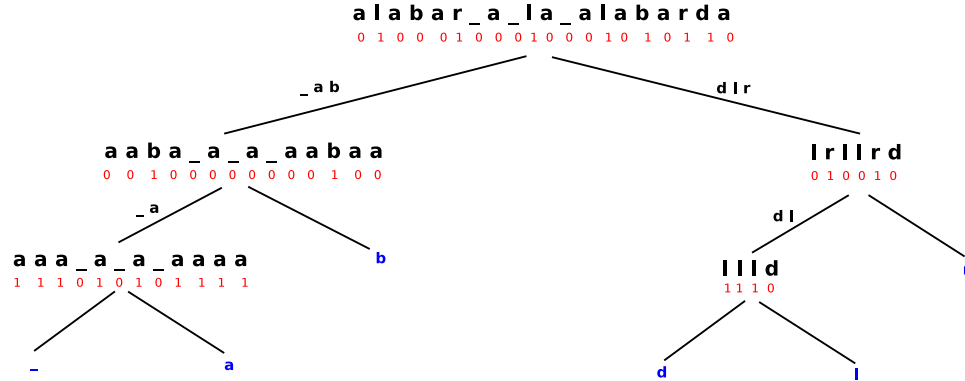


Figure 2.2: The wavelet tree of the sequence “alabar a la alabarda”. The white space is written as an underscore for clarity, and it is lexicographically smaller than the characters “a”-“z”.

The *access* query for position i can be solved by following the path described by position i . At the root, if the bitmap at position i has a 0/1, we descend to the left/right child, switching to the bitmap position $rank_{0/1}(i)$ in the left/right subtree. This continues recursively until reaching the last level.

The *rank* query for a symbol s up to a position i can be answered in a similar way as *access*, the difference being that instead of considering the bit at position i we consider the bit value of the symbol s . Therefore if the bit value of s is 0/1 we descend to the left/right child, switching to the bitmap position $rank_{0/1}(i)$ in the left/right subtree. This continues recursively until reaching the leaves and the bitmap position in the last level corresponds to the answer to *rank* of symbol s in the sequence.

The *select* query does a similar process as *rank*, but upwards. To select the i -th occurrence of character c , we start at the leaf where c is represented and do *select* of the i -th b , where b is the bit of c corresponding to this level. Using the position obtained by the binary *select* query we move to the parent, querying for this new position and the new bit value corresponding to this

level, and so on. At the root, the position is the final result.

The cost of the operations is $O(\log \sigma)$ assuming constant time for *rank*, *select*, and *access* over the bitmaps. With some care the *wavelet tree* can be represented in $n \log \sigma + o(n \log \sigma)$ bits [NM07]. A practical variant to achieve $n(H_0(S) + 1)(1 + o(1))$ bits of space is to give the *wavelet tree* the same shape than the Huffman tree of S [GGV03].

2.4 Tries or Digital Trees

A digital tree or *trie* [Fre60, Knu73] is a data structure that stores a set of strings. It can support the search for a string in the set in time proportional to the length of the string sought, independently of the set size.

Definition 2.1. A *labeled tree* is a tree in which each edge is associated with a label from a given alphabet Σ .

Definition 2.2. A *trie* for a set G of distinct strings S^1, S^2, \dots, S^N is a labeled tree where each node v represents a distinct prefix $\pi(v)$ in the set, called its *path-label*. The root node represents the empty prefix, $\pi(\text{root}) = \varepsilon$. If node v is a child of node u and character c labels the tree edge from u to v , then $\pi(v) = \pi(u).c$.

We assume that all strings are terminated by “\$”. Under this assumption, no string S^i is a prefix of another, and thus the trie has exactly N leaves, each corresponding to a distinct string.

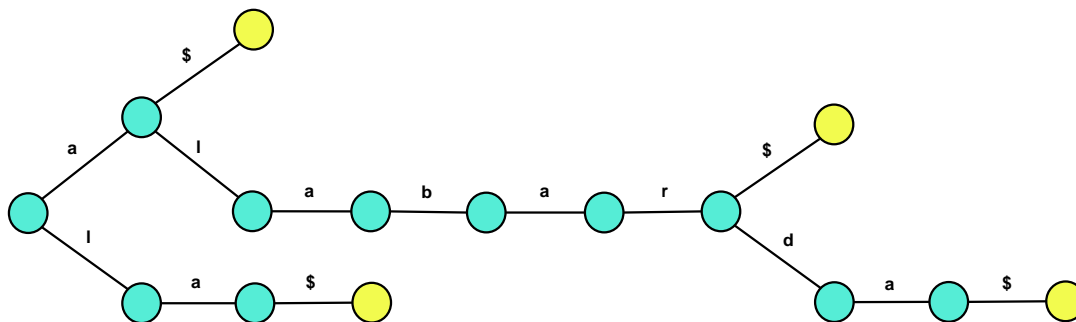


Figure 2.3: A trie for the set {“alabar”, “a”, “la”, “alabarda”}. In general the arity of trie nodes may be as large as the alphabet size.

A trie for $G = \{S^1, S^2, \dots, S^N\}$ is easily built in time $O(|S^1| + |S^2| + \dots + |S^N|)$ by successive insertions. Any string S' can be searched for in the trie in time $O(|S'|)$ by following from the trie root the path labeled with the characters of S' . Two outcomes are possible: (i) at some point i there is no edge labeled $S'[i]$ to follow, which means that S' is not in the set S' , (ii) we reach a leaf corresponding to S' (assume that S' is also terminated with character “\$”).

Actually, the above complexities assume that the alphabet size σ is a constant. For general σ , we must multiply the above complexities by $O(\log \sigma)$, which accounts for the overhead of searching the correct character to follow inside each node. This can be made $O(1)$ by using at each node a direct addressing table of size σ , but in this case the size must be multiplied by $O(\sigma)$ to allocate the tables at each node. Alternatively, perfect hashing of the children of each node permits $O(1)$ search time and $O(1)$ space factor, yet the construction cost is multiplied by $O(\sigma^2)$.

2.5 Suffix Trees

Let us now consider how tries can be used for text indexing. Given text $T_{1,n}$ (terminated with $T[n] = \$$), T defines n suffixes $T_{1,n}, T_{2,n}, \dots, T_{n,n}$.

Definition 2.3. The *suffix trie* of a text T is a trie data structure built over all the suffixes of T .

The suffix trie of T makes up an index for fast string matching. Given a pattern $P_{1,m}$ (not terminated with “\$”), every occurrence of P in T is a substring of T , that is, the prefix of a suffix of T . Entering the suffix trie with the characters of P leads us to a node that corresponds to all the suffixes of T prefixed by P (or, if we do not arrive at any trie node, then P does not occur in T). This permits counting the number of occurrences (*occ*) of P in T in $O(m)$ time, by simply recording the number of leaves that descend from each suffix tree node. It also permits finding all the *occ* occurrences of P in T in $O(m + \text{occ})$ time by traversing the whole subtree (some additional pointers threading the leaves and connecting each internal node to its first leaf are necessary to ensure this complexity).

As described, the suffix trie usually has $\Theta(n^2)$ nodes. In practice, the trie is pruned at a node as soon as there is only a unary path from the node to a leaf. Instead, a pointer to the text position where the corresponding suffix starts is stored. On average, the pruned suffix trie has only $O(n)$ nodes [SF96]. Yet it might have $\Theta(n^2)$ nodes in the worst case. Fortunately, there exist equally powerful structures that guarantee linear space and construction time in the worst case, called *suffix trees* [Wei73, McC76]. Figure 2.4 gives an example of a suffix tree.

Definition 2.4. The *suffix tree* of a text T is a suffix trie where each unary path is converted into a single edge. Those edges are labeled by strings obtained by concatenating the characters of the replaced path. The leaves of the suffix tree indicate the text position where the corresponding suffixes start.

Since there are n leaves and no unary nodes, it is easy to see that suffix trees require $O(n)$ words of space (the strings at the edges are represented with pointers to the text). Moreover, they can be built in $O(n)$ time [McC76, Ukk95].

The search for P in the suffix tree of T is similar to a trie search. Now we may use more than one character of P to traverse an edge, but all edges leaving from a node have different first characters. The search can finish in three possible ways: (i) at some point there is no edge leaving

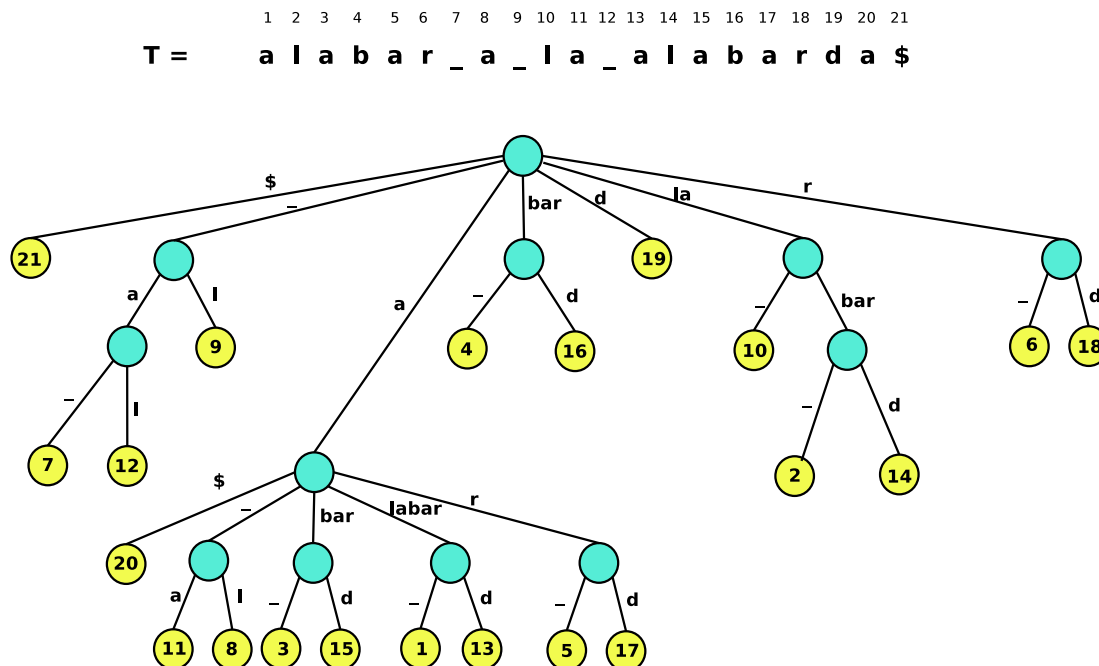


Figure 2.4: The suffix tree of the text “alabar a la alabarda\$”. The white space is written as an underscore for clarity, and it is lexicographically smaller than the characters “a”–“z”.

from the current node that matches the characters that follow in P , which means that P does not occur in T ; (ii) we read all the characters of P and end up at a tree node (or in the middle of an edge), in which case all the answers are in the subtree of the reached node (or edge); or (iii) we reach a leaf of the suffix tree without having read the whole P , in which case there is at most one occurrence of P in T , which must be checked by going to the suffix pointed to by the leaf and comparing the rest of P with the rest of the suffix. In any case the process takes $O(m)$ time (assuming the usage of perfect hashing to find the children in constant time) and suffices for counting queries.

Suffix trees permit $O(m + occ)$ locating time without the need of further pointers to thread the leaves, since the subtree with occ leaves has $O(occ)$ nodes. The real problem of suffix trees is their high space consumption, which is $\Theta(n \log n)$ bits and at the very least 10 times the text size in practice [Kur99].

Several navigation operations over the nodes and leaves of the suffix tree are of interest. Table 2.2 lists the most common ones. As an example of how useful can be these operations, we can mention that *SLink* is necessary to use the suffix tree as an automaton recognizing all the substrings of the text, which can be used to compute the longest common substring of two strings, matching statistics, etc. [Gus97]. Another example is the *lowest common ancestor (LCA)*

Operation	Description
Root()	the root of the suffix tree.
Locate(v)	the suffix position i if v is the leaf of suffix $T_{i,n}$, otherwise NULL.
Ancestor(v, w)	true if v is an ancestor of w .
SDepth(v)/TDepth(v)	the string-depth/tree-depth of v .
Count(v)	the number of leaves in the subtree rooted at v .
Parent(v)	the parent node of v .
FChild(v)	the alphabetically first child of v .
NSibling(v)	the alphabetically next sibling of v .
Letter(v, i)	the i -th letter of v 's path-label, $\pi(v)[i]$.
SLink(v)	the suffix-link of v ; i.e., the node w s.th. $\pi(w) = \beta$ if $\pi(v) = a\beta$ for $a \in \Sigma$.
SLink ^{i} (v)	the iterated suffix-link of v ; i.e., the node w s.th. $\pi(w) = \beta$ if $\pi(v) = a\beta$ for $a \in \Sigma^i$.
LCA(v, w)	the lowest common ancestor of v and w .
Child(v, a)	the node w s.th. the first letter on edge (v, w) is $a \in \Sigma$.
LAQ _S (v, d)/LAQ _T (v, d)	the highest ancestor of v with string-depth/tree-depth $\geq d$.

Table 2.2: Operations over the nodes and leaves of the suffix tree.

operation, which is necessary to compute the longest common extension of two suffixes in constant time, useful in approximate string matching problems [LV88].

2.6 Suffix Arrays (SA)

A suffix array [MM93] is a permutation of all the suffixes of T so that the suffixes are lexicographically sorted. Figure 2.5 gives an example of a suffix array (disregard for now LCP and Ψ arrays).

Definition 2.5. The *suffix array* of a text $T_{1,n}$ is an array $A[1, n]$ containing a permutation of the interval $[1, n]$, such that $T_{A[i],n} < T_{A[i+1],n}$ for all $1 \leq i < n$, where “ $<$ ” between strings is the lexicographical order.

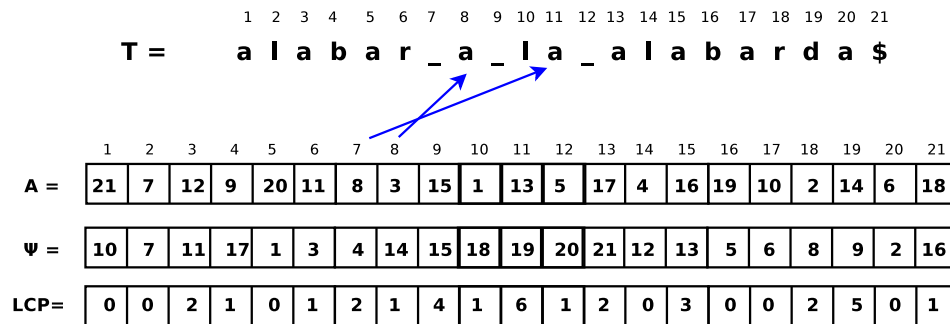


Figure 2.5: The suffix array (A), longest common prefix array (LCP), and Ψ array of the text “alabar a la alabarda\$”.

Note that if the children of each node of the suffix tree are ordered lexicographically by their string label, then the suffix array can be obtained by collecting the leaves of the suffix tree in

left-to-right order. However, it is much more practical to build it directly [KSB06, PST07]. In principle, any comparison-based sorting algorithm can be used, as it is a matter of sorting the n suffixes of the text, but this could be costly especially if there are long repeated substrings within the text. In 2003, it was shown that the suffix array can be constructed in $O(n)$ worst-case time [KA03, KS03, KSPP03].

To search for a pattern $P[1, m]$ in T one can search for the interval $A[sp, ep]$ of the suffixes starting with the pattern. This can be done via two binary searches on A . The first binary search determines the starting position sp for the suffixes lexicographically larger than or equal to P . The second binary search determines the ending position ep for suffixes that start with P . Then the answer is the interval $A[sp, ep]$, that is, each $A[i]$, $sp \leq i \leq ep$, is the starting position of an occurrence of P in T . The binary searches take time $O(m \log n)$ and the $occ = ep - sp + 1$ occurrences are reported in time $O(occ)$.

There are several *compressed suffix arrays* [NM07, FGNV09], which offer essentially the following functionality: (1) Given a pattern $P[1, m]$, find the interval $A[sp, ep]$ of the suffixes starting with P ; (2) obtain $A[i]$ given i ; (3) obtain $A^{-1}[j]$ given j . An important function that most of the compressed suffix arrays implement efficiently is $\Psi(i)$ and its inverse $LF(i)$. These functions, to be defined in the sequel, let us move virtually in the text, from the suffix i that points to text position $j = A[i]$, to the one pointing to $j + 1 = A[\Psi(i)]$ (see Figure 2.5) or $j - 1 = A[LF[i]]$.

In this thesis we consider Sadakane's Compressed Suffix Array (*CSA*) [Sad03] and the *FM-Index* [FM00]. It is important to mention that both compressed suffix arrays presented in this thesis are *self-indexes* (recall Section 2.2).

2.6.1 Sadakane's Compressed Suffix Array (CSA)

Sadakane's *CSA* [Sad03] represents the suffix array and the text using the function Ψ .

Definition 2.6. Let $A[1, n]$ be the suffix array of a text T . Then $\Psi(i)$ is defined as the position i' in the suffix array where $A[i'] = (A[i] \bmod n) + 1$, which is $A[i'] = A[i] + 1$ except for the case $A[i] = n$, where it holds $A[i'] = 1$.

It is not hard to see that Ψ is monotonically increasing in the areas where the suffixes start with the same symbol [GV05]. To see that $\Psi(i) < \Psi(i+1)$ whenever $T[A[i]] = T[A[i+1]]$, assume that $T_{A[i],n} = c.X$ and $T_{A[i+1],n} = c.Y$, so $c.X < c.Y$ and then $X < Y$. Thus $T_{A[i]+1,n} = T_{A[\Psi(i)],n} = X$ and $T_{A[i+1]+1,n} = T_{A[\Psi(i+1)],n} = Y$. So $T_{A[\Psi(i)],n} < T_{A[\Psi(i+1)],n}$, and thus $\Psi(i) < \Psi(i+1)$. In addition, there are long parts where $\Psi(i+1) = \Psi(i) + 1$ [MN05]. These properties permit a compact representation of Ψ and its fast access. Essentially, we differentially encode $\Psi(i) - \Psi(i-1)$, run-length encode the long runs of 1's occurring over those differences, and for the rest use an encoding favoring small numbers [Sad03]. Absolute samples are stored at regular intervals to permit the efficient decoding of any $\Psi(i)$. The sampling rate gives a space/time trade-off for accessing and storing Ψ .

To obtain $A[i]$ the basic idea is to store a sample of values of the suffix array. Sadakane samples A at regular intervals of length SR in the text, explicitly storing $A[k]$ iff $A[k] = i \cdot SR$ for $i \in \{1, \dots, n/SR\}$. These sampled positions k are marked in a bitmap. Thus, if after applying the Ψ function g times we get a sampled value $A[k]$, then the value of the original position is $A[k] - g$. The sampling rate of A is a crucial parameter that trades space, requiring $(n \log n)/SR$ bits for storing the samples, and taking time $O(SR)$ to compute $A[i]$.

To extract an arbitrary suffix $T_{A[i],n}$ Sadakane stores a bitmap $D_{1,n}$, so that $D[i] = 1$ iff $i = 1$ or $T[A[i]] \neq T[A[i-1]]$, and a string $charT$, of size at most σ , where the distinct characters of T are concatenated in alphabetical order. Then if we want the first character of some suffix $j = A[i]$, we just need to compute $c = T[j] = charT[rank_1(D, i)]$, which can be done in constant time using only $n + o(n)$ bits for D (Section 2.3) and $\sigma \log \sigma$ bits for $charT$. To extract the next letter of the suffix $T_{A[i],n}$, we use the identity $T[A[i] + 1] = T[A[\Psi(i)]]$, thus we simply have to move to $i' = \Psi(i)$ and carry out the same process again to obtain $T[A[i']]$, and so on. By using this we can find the interval $A[sp, ep]$ of the suffixes starting with a given pattern $P[1, m]$. Since each step of the binary searches requires a lexicographical comparison between P and some substring $T_{A[i], A[i+m-1]}$, it takes $O(m \log n)$ worst-case time. Figure 2.6 shows the pseudocode for the comparison.

Algorithm *CSA-compare*($P, m, i, \Psi, D, charT$)

1. $l \leftarrow 1$;
 2. **do**
 3. $c \leftarrow charT[rank_1(D, i)]$;
 4. **if** $P_l < c$ **then return** “<”;
 5. **if** $P_l > c$ **then return** “>”;
 6. $i \leftarrow \Psi(i)$;
 7. $l \leftarrow l + 1$;
 8. **while** $l \leq m$;
 9. **return** “=”;
-

Figure 2.6: Comparing P against $T_{A[i],n}$ using Ψ , D , and $charT$.

Finally, in order to discard the text, we need to be able to extract any substring $T_{a,b}$. For the same sampled text positions $i \cdot SR$ sampled above, we store $A^{-1}[i \cdot SR]$ in text position order. Thus, we find the latest sampled position $i \cdot SR$ preceding a , $i = \lfloor \frac{a}{SR} \rfloor$, and know that $i \cdot SR$ is pointed from $j = A^{-1}[i \cdot SR]$, which is sampled. From that j we use the mechanism we have described to extract a string using $charT$ and Ψ , to find out the substring $T_{i \cdot SR, b}$ which covers the one of interest to us. This is not the way Sadakane’s theoretical description handles this, but the way he implemented it in practice. Using this, a substring $T_{a,b}$ of length m can be extracted in $O(m + \log^{1+\epsilon} n)$ time, where $0 < \epsilon \leq 1$ is an arbitrary constant which depends on the sample size chosen when the index is built. Note that using the same sampling, we also can compute

$A^{-1}[i]$ in time $O(\log^{1+\epsilon} n)$.

This compressed suffix array [Sad00, Sad03] requires $nH_0(T) + O(n \log \log \sigma)$ bits of space in its practical implementation. Mäkinen and Navarro later improved the space analysis to $nH_k(T) + O(n \log \log \sigma)$, for $k \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$ [MN07a]. The time for counting the number of occurrences of some pattern P of size m is $O(m \log n)$, the time for locating is $O(\log^{1+\epsilon} n)$ per occurrence, and the time to display l symbols of the text is $O(l + \log^{1+\epsilon} n)$, where $0 < \epsilon \leq 1$ is an arbitrary constant which depends on the sampling step $SR = \log^\epsilon n$. Note that Ψ is computed in time $O(1)$ (actually this depends on the rate of absolute values sampled in Ψ). In theory they can achieve better complexities, but this has not been implemented as it requires much more space.

2.6.2 The FM-Index

The *FM-Index* is based on the Burrows-Wheeler transform (*BWT*) [BW94]. We first explain the transformation, how to recover the text from the transformation, and how to search for a pattern inside the transformed text. Then we describe FM-Index and how it represents the *BWT*.

The Burrows-Wheeler Transform (BWT)

For a text T of length n , imagine a matrix M_T of dimensions $n \times n$ whose rows are the rotations (cyclic shifts) of T in incremental order. If we sort in lexicographic order the rows of this matrix, then the *BWT* (T^{bwt}) is the last column of that matrix. This is equivalent to collecting the character preceding every suffix of the suffix array A of T , hence the *BWT* can be identified with A . Figure 2.7 shows an example of how the *BWT* is computed for the text “*alabar a la alabarda*”. Assuming that the last symbol of T is lexicographically smaller than all the others, and that it is unique, it is possible to reverse the transformation: $T[n-1]$ is located at $T^{bwt}[1]$, since the first element in the sorted matrix starts with $T[n]$. To continue we need a definition.

Definition 2.7. The *LF-mapping* is defined as $LF(i) = C[c] + rank_c(T^{bwt}, i)$, where $c = T^{bwt}[i]$ and $C[c]$ is the number of occurrences of symbols lexicographically smaller than c in T .

LF stands for *Last-to-First* column mapping since the character $T^{bwt}[i]$, in the last column of M_T , is located in the first column of M_T at position $LF(i)$. Thus, *LF* allows us to navigate T backwards, $T[n-2] = T^{bwt}[LF(1)]$, and for any position k , $T[n-k] = T^{bwt}[LF^{k-1}(1)]$. Note that *LF* is the inverse function of Ψ , because $LF(i)$ corresponds to the position in A of the suffix pointing to $T[A[i]-1]$.

Given the relation between T^{bwt} and A , it is natural to ask whether one can use T^{bwt} for searching. This is indeed the case, and the method is called *backward search*. Figure 2.8 shows the algorithm for counting the occurrences of a pattern P using the *BWT*.

applying the LF function k times we get a sampled value $A[j]$, then the value for the original position is $A[j] + k$. The sampling rate of A is again a crucial parameter that trades space for query time. A sampling similar to that in Section 2.6.1 allows one to compute A^{-1} and to extract any substring $T_{a,b}$.

There are several variants of the FM-Index [FMMN07, MN05, NM07, FGNV09]. For this thesis we use Ferragina et al.'s [FMMN07] version that uses $nH_k(T) + o(n \log \sigma)$ bits for $k \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$. In practice the time for counting the number of occurrences of a pattern P of size m is $O(m \log \sigma)$, locating takes $O(\log^{1+\epsilon} n)$ time per occurrence, and reporting a text substring of length l takes $O(l \log \sigma + \log^{1+\epsilon} n)$ time, where $\epsilon > 0$ is an arbitrary constant related to the sampling rate. Note that $A[i]$ and $A^{-1}[i]$ are computed in time $O(\log^{1+\epsilon} n)$, and LF in time $O(\log \sigma)$.

2.7 Longest Common Prefix (LCP)

Let $lcp(S^1, S^2)$ be the length of the longest common prefix between the sequences S^1 and S^2 . The longest common prefix (*LCP*) array of a text T is the array $LCP[1, n]$ such that $LCP[1] = 0$ and $LCP[i] = lcp(T_{SA[i-1], n}, T_{SA[i], n})$ for $i > 1$. Seen in another way, the $LCP[i]$ is the length of the string labeling the path from the root to the lowest common ancestor node of suffix tree leaves i and $i - 1$. The array requires $n \log n$ bits of space, and can be constructed in $O(n)$ time [KLA⁺01, KMP09]. In Chapter 5 we will discuss different ways to represent the *LCP* array in compact form. Figure 2.5 shows an example of the *LCP* array.

Chapter 3

Succinct Tree Representations

A general tree of n nodes can be represented in pointer form, requiring $O(n \log n)$ bits, whereas the succinct representations we studied require just $2n + o(n)$ bits and carry out many sophisticated operations in constant time. Yet, there is no exhaustive study in the literature comparing the practical magnitudes of the $o(n)$ -space and the $O(1)$ -time terms.

As part of our research we implemented and compared the major current techniques for representing general trees in succinct form. This study was done while we were searching for the best way to represent our compressed suffix tree. Although in the end we do not use any tree implementation, the study is of great interest and constitutes an added value to the thesis. Most important is the fact that this research lead us to find new ideas that were applied in other parts of this thesis (see Section 6.2). The work of this chapter was done in cooperation with Diego Arroyuelo and Kunihiko Sadakane, and was published in *Proc. ALENEX 2010* [ACNS10].

The techniques that we studied can be classified into three broad trends: those based on BP (balanced parentheses in preorder), those based on DFUDS (depth-first unary degree sequence), and those based on LOUDS (level-ordered unary degree sequence). BP and DFUDS require a balanced parentheses representation that supports the core operations *findopen*, *findclose*, and *enclose*, for which we implement and compare three major algorithmic proposals. All the tree representations also require core operations *rank* and *select* on bitmaps, which are already well studied in the literature. We showed how to predict the time and space performance of most variants via combining these core operations, and also studied some tree operations for which specialized implementations exist. This is especially relevant for a recent proposal [SN10] which, although belonging to the class BP, deviates from the main techniques in some cases in order to achieve constant time for the widest range of operations. We experimented over various types of real-life trees and traversals, concluding that the latter technique stands out as an excellent practical combination of space occupancy, time performance, and functionality, whereas others, particularly LOUDS, are still interesting in some limited-functionality niches.

3.1 Trees and their Representation

Trees are, on one hand, the paradigmatic data structure in Computer Science, probably rivalled in popularity only by arrays and linked lists; and on the other hand, one of the most striking examples of the success of succinct data structures. A classical representation of a general tree of n nodes requires $O(nw)$ bits of space, where $w \geq \log n$ is the bit length of a machine pointer. The associated constant is at least 2, and typically only operations such as moving to the first child and to the next sibling, or to the i -th child, are supported. By further increasing the constant, some other simple operations are easily supported, such as moving to the parent, knowing the subtree size, or the depth. Much research was needed (and further increase of the constant) to support sophisticated operations such as level-ancestor [BFC04] and lowest common ancestor [BFC00] queries. While the constant-time complexities achieved for the sophisticated queries are remarkable, the $\Omega(n \log n)$ -bit space complexity is not justified in terms of Information Theory: there are only $C_n \sim 4^n/n^{3/2}$ different general trees of n nodes, and thus $\log C_n = 2n - \Theta(\log n)$ bits are sufficient to distinguish any one of them. This huge space gap has motivated a large body of research [Jac89, MR01, MRR01, MR04, GRR04, BDM⁺05, GRRR06, DRR06, HMR07, GGG⁺07, Sad07a, JSS07, LY08, FM08, SN10] achieving $2n + o(n)$ bits of space and constant time for an impressive set of operations. Table 3.1 lists those we consider in this work. There are several others, yet most are solved analogously to those in this set. Note that we consider *labeled* trees, in which case $n \log \sigma$ additional bits are needed to represent the labels. There has been some work over labeled trees [GRR04, BDM⁺05, FLMM05, BGMR07, BHMR07] where different approaches to represent labeled trees, supporting many operations over the labels, are explored. For this thesis we will only consider operations *labeled_child* and *child_label* which are those with clear applications to suffix trees.

The existing proposals differ in their functionality, ranging from those that support basically child/parent navigation [Jac89, DRR06] to those supporting a full range of operations [BDM⁺05, JSS07, FM08, SN10]; in the nature of the $o(n)$ space overhead, ranging from $O(n/(\log \log n)^2)$ [LY08] to $O(n/\text{polylog}(n))$ [SN10]; and in some rare cases on their capability to take less space when the tree is compressible in some sense [JSS07]. In theoretical terms, the problem can be considered basically solved, at least in the static scenario.

In practice, however, the situation is much less satisfactory. Implementations of the theoretical proposals are scarce and far from trivial, even for those striving for simplicity [GRRR06, DRR06, SN10]. Verbatim implementations of the theoretical proposals are usually disastrous, as already noted in previous work [Nav09], and much empirical experience must be combined with the theoretical ideas in order to obtain theoretically and practically sound solutions. There are few practical comparisons among techniques, none of which is sufficiently exhaustive to give a broad idea of their performance. In practice, the $O(1)$ -time and the $o(n)$ -space terms in the asymptotic analyses do not give much clues about how the structures actually perform.

In this study, we carefully implement the theoretical proposals we consider most promising in practice, and compare them for various real-life trees and typical tree traversals. Tree represen-

parentheses operation	description
$findclose(i)/findopen(i)$	position of parenthesis matching $P[i]$
$enclose(i)$	position of tightest opening parenthesis enclosing node i
$rank_{(}(i)/rank_{)}(i)$	number of opening/closing parentheses in $P[1, i]$
$select_{(}(i)/select_{)}(i)$	position of i -th open/close parenthesis
tree operation	description
$pre_rank(x)$	preorder rank of node x
$pre_select(x)$	the node with preorder x
$isleaf(x)$	whether node x is a leaf
$ancestor(x, y)$	whether x is an ancestor of y
$depth(x)$	depth of node x
$parent(x)$	parent of node x
$first_child(x)$	first child of node x
$next_sibling(x)$	next sibling of node x
$subtree_size(x)$	number of nodes in the subtree of node x
$degree(x)$	number of children of node x
$child(x, i)$	i -th child of node x
$child_rank(x)$	number of siblings to the left of node x
$level_ancestor(x, d)$	ancestor y of x such that $depth(y) = depth(x) - d$
$lca(x, y)$	the lowest common ancestor of two nodes x, y
labeled tree operation	description
$labeled_child(x, s)$	child of node x labeled by symbol s
$child_label(x)$	label of edge between node x and its parent

Table 3.1: Operations on parentheses and trees considered in this study.

tations can be broadly classified into the following tracks:

BP: The *balanced parentheses* representation of a tree, first advocated as a succinct data structure by Jacobson [Jac89], and later achieving constant times [MR01], is built from a depth-first preorder traversal of the tree, writing an *opening* parenthesis when arriving to a node for the first time, and a *closing* parenthesis when going up (after traversing the subtree of the node). In this way, we get a sequence of $2n$ balanced parentheses. Each node is represented by a matching pair of parentheses '(' and ')', and we identify the node with its opening parenthesis. The subtree of x contains those nodes (parentheses) enclosed between its representing opening and closing parentheses. The core operations (see Table 3.1) on this sequence are sufficient to implement most of the functionality.

DFUDS: *Depth-first unary degree sequence* [BDM⁺05]. It is built by the same traversal, except that, upon arriving at each node, we append i opening and one closing parentheses, being i the number of children of the node. The node is represented by the position where its i parentheses start. The resulting sequence turns out to be balanced (if one prepends an extra opening parenthesis at the beginning) and the same core operations on parentheses are used to support the same functionality of BP in a different way (except for *depth*, which requires extra structures [JSS07]) plus others, most notably *child*, in constant time.

LOUDS: *Level-ordered unary degree sequence* [Jac89, BDM⁺05]. Nodes are processed and represented as in DFUDS, but they are traversed level-wise, instead of in depth-first order. It turns out that just the operators *rank* and *select* on symbols '(' and ')' (noted respectively $rank_{(}$, $rank_{)}$, $select_{(}$, and $select_{)}$) are sufficient to support a few key operations such as

parent and *child* in constant time, yet most other operations are not supported as efficiently (they can still be supported in $O(n)$ time).

FF: The recent so-called *fully-functional succinct tree* representation [SN10] is based on a BP representation. It includes a novel data structure to handle the core operations, called a *range min-max tree*. With little extra data, this same structure is able to solve in constant time many other sophisticated operations that are not usually handled by other BP representations, such as *child*, *lca*, and *level_ancestor*.

Specifically the implementations were done by K. Sadakane (FF), D. Arroyuelo (DFUDS), G. Navarro (a hash-based representation for BP), and the author of this thesis (LOUDS and a recursive pioneer-based representation of BP). We omit an extra track based on tree covering [GRR04, HMR07, FM08], of which we do not know of any practical implementations (we thank Arash Farzan and Meng He for confirming this).

We first study different alternatives to represent a sequence of balanced parentheses while supporting the core operations. These include a hash-based heuristic implementation developed for a compressed text index [Nav09], Geary et al.’s recursive pioneer-based representation [GRRR06], and the range min-max trees [SN10]. We exhaustively compare these operations over real-life trees coming from (a) LZ78 parsings of text collections, (b) suffix trees of text collections, and (c) the tag structure of XML collections. We emulate typical root-to-leaf (and vice versa) traversals coming from exact search, LZ78 decompression, and other very common navigation operations, as well as traversals where in addition every other child of the current node is visited with a given probability. These are meant to emulate traversals typical of backtracking, range or proximity searches, and XPath queries.

We briefly survey the way BP, DFUDS, and LOUDS build on these core operations, and show how to predict the performance of most operations in terms of the core ones. We also study some tree operations individually, when the prediction from the performance of core operations is less direct. Overall, our work provides implementations of the most promising techniques to represent succinct trees, the first exhaustive empirical comparison among them, and recommendations on which structures to use depending on the case. It turns out that the recent FF proposal [SN10] stands out as an excellent combination of wide functionality, little space usage, and good time performance. Other approaches are still relevant for certain niches. In particular, we show that LOUDS, which had received comparatively little attention, is an extremely simple and efficient alternative when only the simpler operations are needed. LOUDS excels when descending to an arbitrary ordinal or labeled child. For this latter operation, the DFUDS variant also shows to be a reasonable compromise if a wide functionality must be retained.

3.2 Balanced Parentheses

Let $P[1, 2n]$ be a sequence of n pairs of balanced parentheses (represented with bits 0,1). We describe the three major representations we will consider, which support the core parenthesis operations described in Table 3.1. Let us define the key function $excess(i) = rank_{(i)} - rank_{)}(i)$. Then it holds that $findclose(i)$ is the smallest $j > i$ such that $excess(j) = excess(i - 1)$ ($findopen(i)$ is analogous). Also, $enclose(i)$ is the greatest $j < i$ such that $excess(j) = excess(i) - 1$.

3.2.1 Hash-Based Heuristic (HB)

Instead of storing all the possible answers to $findclose$ (which would require $O(n \log n)$ bits of space), this representation [Nav09] divides the parentheses into three groups. Let $b = \log n$ and $s = b \log n$. Let *close* parentheses be those whose matching parentheses are at distance at most b ; *near*, between $b + 1$ and s ; and *far*, farther than s positions away. Only the answers for near and far parentheses are explicitly stored, in hash tables. The table for far parentheses uses $O(\log n)$ bits per value stored, while that for near parentheses uses just $\log s$ bits (because the distance from the argument is stored, rather than the absolute answer).

To compute $findclose(i)$, we first scan the next b positions, looking for the first parenthesis with the same $excess(i - 1)$. If the answer is not found in this way, the hash table for near parentheses is queried. Since the search key is not stored in this table, we consider all colliding answers: if more than one candidate answer is found with the correct excess, the closest one is the correct answer (because the sequence is balanced [Nav09]). Finally, if the answer is still not found, the hash table for far parentheses is queried. A similar approach is used for operations $findopen$ and $enclose$, requiring their own hash tables.

The search for close parentheses is implemented as a scan by chunks of k bits. Hence, for every different k -bit stream, we precompute its excess and, for every $j = 1, \dots, k$, the first position of the k -bit stream where the excess becomes $-j$, if any. To solve $findclose(i)$, we check whether the excess -1 is reached in the first chunk after position i . If so, the position where this happens is $findclose(i)$. Otherwise, we consider the second chunk, looking for excess $-1 - e_1$, where e_1 is the total excess of the first chunk, and so on. If the b bits are exhausted without a solution, then the answer is not close. Scanning for $findopen(i)$ and $enclose(i)$ is analogous.

The parenthesis operations are thus supported in $O(b/k)$ time, plus two searches on hash tables, which are $O(1)$ on average. If we set $k = \frac{\log n}{c}$, for constant $c > 1$, the average time is $O(c)$. The precomputed table for close parentheses requires $2^k(2k + 1) \log k = O(n^{1/c} \log n \log \log n) = o(n)$ bits. Furthermore, we need $\log s = O(\log \log n)$ bits per near parenthesis and $O(\log n)$ per far parenthesis. However, there are no theoretical bounds on these types of parentheses (e.g., on a tree formed by n nodes with one child, almost all of them are far).

Implementation notes. We choose $k = 8$ so as to handle bytes, hence the table requires just a few kilobytes and admits good caching. The hash tables are implemented by a closed hashing scheme, with load factor 1.8 for *enclose* and 1.6 for *findclose* and *findopen*, which gave us the best space/time trade-off (recall that we do not store the keys in the hash table for near parentheses, so we pay a noticeable cost per collision, and the cost is proportional to that of an unsuccessful search). For operation *excess*, we use González et al.’s lightweight *rank/select* data structure [GGMN05] (*ggmn* of Section 2.3).

3.2.2 Recursive Pioneer-Based Representation (RP)

The representation by Geary et al. [GRRR06] divides P into blocks of size $b = \frac{\log n}{2}$. A parenthesis at position i is said to be *close* if its matching pair belongs to the same block, otherwise the pair is called *far*. Computing *findclose* for a close parenthesis is done by using precomputed tables, as in Section 3.2.1, requiring $o(n)$ bits of space. An opening far parenthesis at position i is said to be a *pioneer* if the previous opening far parenthesis has its matching closing parenthesis in a block different to that of the closing parenthesis for i . Both parentheses forming the matching pair are called pioneers. The total number of pioneers is $O(n/b)$ [Jac89, GRRR06]. Thanks to the pioneer properties, *findclose*(i) is reduced to (a) finding the previous pioneer $i' \leq i$ (which could be i itself), (b) find $j' = \text{findclose}(i')$, (c) scan the block of j' backwards until finding $j = \text{findclose}(i)$ using the *excess* property (again by means of precomputed tables).

To solve *findclose* for pioneers, we form a *reduced sequence* with the pioneers, and apply the solution recursively once more. The second time the reduced sequence is of length $O(n/b^2)$ and thus we can store all the answers in $O(n \log(n)/b) = o(n)$ bits.

The remaining piece is a bitmap $B[1, 2n]$ marking the pioneers. With *rank*₁ and *select*₁ on this sequence we find the previous pioneer, map it to the reduced sequence, and map the answer to *findclose* back to the original sequence. As B has $O(n/b)$ 1s, it can be represented using $O(n \log b/b) = o(n)$ bits of space [RRR02].

The solution to *enclose*(i) is similar. After checking within the same block of i , we look for the first pioneer c following i . If it is a closing parenthesis, then $p = \text{findopen}(c)$, otherwise p is the pioneer most tightly enclosing c (i.e., its *parent* in the sequence of pioneers). Now let q be the first pioneer following p . If q is in the same block as p , then our answer is the first far parenthesis to the left of q ; else the answer is the rightmost far parenthesis in the block of p . Both require just within-block scans. See the original article [GRRR06] for more details.

Implementation notes. For compressed bitmaps [RRR02] we use a recent implementation [CN08] (*rrr* structure in Section 2.3). Because the second-level bitmap B is significantly smaller and not too compressible, we represent it in uncompressed form. All uncompressed bitmaps are implemented as those in Section 3.2.1.

3.2.3 Range Min-Max-Tree Based Representation (RMM)

The basic tool of this technique [SN10] is the *range min-max tree*. This is built over the (virtual) array of $excess(i)$ values. The sequence P is cut into *blocks* of $b = \frac{w}{2}$ parentheses (recall that $w \geq \log n$ is the machine word length), and we store the minimum and maximum (local) excess within each block. We then group k blocks into a *superblock*, and store the minimum and maximum excess per superblock. We then group k superblocks, and so on until building up a complete k -ary hierarchy. The total space is $O(n \log(b)/b) = o(n)$ bits. The range min-max tree yields the support of the following kernel operations. For conciseness, let us overload the operation $excess(i)$ by $excess(i, j) = excess(j) - excess(i - 1)$.

- $fwd_search(i, d)$: gives the minimum $j > i$ such that $excess(i, j) = d$
- $bwd_search(i, d)$: gives the maximum $j < i$ such that $excess(j, i) = d$

With the kernel operations, we easily implement the basic parentheses operations, and also some sophisticated tree operations:

$$\begin{aligned} findclose(x) &\equiv fwd_search(x, 0) \\ findopen(x) &\equiv bwd_search(x, 0) \\ enclose(x) &\equiv bwd_search(x, 2) \\ level_ancestor(x, d) &\equiv bwd_search(x, d + 1) \end{aligned}$$

To solve $fwd_search(i, d)$ we first scan the block of i in constant time using precomputed tables as before. If unsuccessful, we climb up the range min-max tree until we find a min/max excess range that, translated into absolute, contains $excess(i - 1) + d$: At each node, we scan the values to the right of the ancestor of i . Once the proper range min-max tree node is found, we go down finding the first child that contains the desired excess, until reaching the leaf block, which is scanned to find the exact j value. The process is analogous for bwd_search .

If $n = \text{polylog}(w)$ and $k = \Theta(w/\log w)$, then there are $O(1)$ levels and the scanning per node can be done in constant time using universal tables of size $o(2^w)$. A constant-time solution for large trees using $2n + O(n/\text{polylog}(n))$ bits is significantly more complex [SN10], so we use range min-max trees for all cases. As a result, the complexity becomes $O(\log n)$ for all operations, yet the structure is extremely efficient anyway.

Implementation notes. For each block i , we store its local minimum $m[i]$ and maximum $M[i]$ excess values. Thus $-b \leq m[i] \leq 1$ and $-1 \leq M[i] \leq b$ holds and they can be stored in $\lceil \log(b + 2) \rceil$ bits. Therefore the maximum value of b is 254 or $2^{16} - 2$ if we use one byte or two bytes, respectively, to store each value. For example, with $b = 512$ the space overhead is $2 \times 16 \cdot (2n)/b = 6.25\%$.

Operation	BP	DFUDS	LOUDS
$pre_rank(x)$	$rank_{\lceil}(x)$	$rank_{\rceil}(x-1)+1$	
$pre_select(p)$	$select_{\lceil}(p)$	$select_{\rceil}(p-1)+1$	
$isleaf(x)$	$P[x+1]=\lceil'$	$P[x]=\rceil'$	$P[x]=\rceil'$
$ancestor(x,y)$	$x \leq y \leq findclose(x)$	$x \leq y \leq findclose(enclose(x))$	
$depth(x)$	$excess(x)$		
$parent(x)$	$enclose(x)$	$prev_{\rceil}(findopen(x-1))+1$	$prev_{\rceil}(select_{\lceil}(rank_{\rceil}(x-1)))+1$
$first_child(x)$	$x+1$	$child(x,1)$	$child(x,1)$
$next_sibling(x)$	$findclose(x)+1$	$findclose(findopen(x-1)-1)+1$	$select_{\rceil}(select_{\lceil}(rank_{\rceil}(x-1))+1)+1$
$subtree_size(x)$	$(findclose(x)-x+1)/2$	$(findclose(enclose(x))-x)/2+1$	
$degree(x)$		$next_{\rceil}(x)-x$	$next_{\rceil}(x)-x$
$child(x,i)$		$findclose(next_{\rceil}(x)-i)+1$	$select_{\lceil}(rank_{\lceil}(x)+i-1)+1$
$child_rank(x)$		$next_{\rceil}(y)-y; y=findopen(x-1)$	$y-prev_{\rceil}(y); y=select_{\lceil}(rank_{\rceil}(x-1))$

Table 3.2: Constant-time reduction of tree operations in BP and DFUDS to operations on balanced parentheses.

A superblock consists of k blocks, and we build a range min-max tree on superblocks, yet thereafter we use a branching factor 2, forming a complete binary tree. Storing 32-bit numbers and using, say, $k = 32$ and $b = 512$, the space is $2 \times 2 \cdot 32 \cdot (2n)/(bk) \approx 0.78\%$ overhead.

Just as for the other solutions, the sequence P is scanned by bytes when looking for some target $excess$ value. Local excesses are converted into global by adding $excess$ at the beginning of blocks/superblocks. Since $excess(i) = rank_{\lceil}(i) - rank_{\rceil}(i) = 2 \cdot rank_{\lceil}(i) - i$, all we need is a particularly efficient $rank_{\lceil}$ at the beginning of superblocks. Thus we adapt a $rank/select$ scheme similar to the one used for the other representations [OS07], so that the $rank$ structures use blocks aligned with b and the scheme requires little space. We store 4-byte global sums of big blocks of size 2^{16} , 2-byte local sums of blocks of size b , and 1-byte sums of blocks of 255 bits. For $b = 512$ this is $32 \cdot (2n)/2^{16} + 16 \cdot (2n)/b + 8 \cdot (2n)/255 \approx 6.31\%$ overhead.

3.3 Succinct Tree Representations

3.3.1 Preorder Balanced Parentheses (BP)

Table 3.2 shows the way many operations in BP representation are expressed in terms of basic parenthesis operations (we assume that the operations can be applied, otherwise an additional constant-time check is necessary). Operation $child(x,i)$ can be computed in $O(i)$ time by applying $first_child$ and then, repeatedly, $next_sibling$. Operations $degree(x)$ and $child_rank(x)$ are solved similarly. Similarly, we implement $level_ancestor(x,d)$ in $O(d)$ time by successive $parent$ operations. There are constant-time theoretical solutions for these operations [MR04, LY08], but they have not been implemented.

A complex operation that has a practical solution is $lca(x,y) = enclose(rmq(x,y)+1)$ [Sad07a]. Operation $rmq(x,y)$ gives the minimum excess in $P[x,y]$, and is implemented in constant time using $O(n(\log \log n)^2/\log n) = o(n)$ bits of space on top of the excess array [BFC00, Sad07a] (we simulate this excess array with operation $excess$).

Implementation notes. The most practical implementation we know of for *rmq* [FH07] requires 62.5% extra space on top of $P[1, 2n]$, and this would put the representation out of competition in terms of space requirement. A theoretical, not implemented, alternative solution to $lca(x, y)$ is given by operation *double-enclose* [MR01]. Therefore, we chose to take *parent* on the less deep node until it becomes an *ancestor* of the deeper one, then this is the *lca*.

Labeled trees. The labels are stored in preorder in a sequence $L[1, n]$, so that $child_label(x) = L[pre_rank(x)]$ and $labeled_child(x, s)$ is solved by a linear scan, just like $child(x, i)$.

3.3.2 Depth-First Unary Degree Sequence (DFUDS)

Table 3.2 also shows how to translate many operations in DFUDS representation to those on balanced parentheses. For conciseness we note $prev_{\lrcorner}(x) \equiv select_{\lrcorner}(rank_{\lrcorner}(x))$ and $next_{\lrcorner}(x) \equiv select_{\lrcorner}(rank_{\lrcorner}(x) + 1)$. If x is within a sequence of opening parentheses, these operations find the preceding and following closing parenthesis, respectively.

There is a theoretical proposal for implementing *depth* and *level_ancestor* in constant time [JSS07], but this has not been implemented as far as we know. Again, $lca(x, y) = parent(rmq(x, y - 1) + 1)$ can be efficiently implemented in DFUDS [JSS07], where *rmq* works over the excesses of the DFUDS parentheses sequence. (The formula needs an easy fix if $ancestor(x, y)$.)

Implementation notes. For the same practical reasons related to *rmq* and the non-implemented theoretical proposals, we implement *lca*, *level_ancestor*, and *depth* by brute force. For *lca* this means that we have to trace the full path from both nodes towards the root using *parent*, and then find the lowest common node.

We solve $prev_{\lrcorner}$ and $next_{\lrcorner}$ by directly scanning P computer-word-wise, which is preferable to the verbatim solution unless the tree has extremely large arities.

Labeled trees. The labels are written in a separate sequence $L[1, n]$ by traversing the tree in preorder and writing, at each node, the symbols by which its children descend. In a labeled tree one can assume the children are ordered by their symbol. Thus to carry out $labeled_child(x, s)$ we binary search $L[rank_{\lrcorner}(x), rank_{\lrcorner}(x) + degree(x) - 1]$ and finish with a $child(x, i)$ operation. Also $child_label(x) = L[rank_{\lrcorner}(parent(x)) + child_rank(x) - 1]$. By sorting the children labels of each parent in reverse order we save some $prev_{\lrcorner}/next_{\lrcorner}$ operations. There is a constant-time theoretical proposal for *labeled_child* [BDM⁺05], not implemented as far as we know.

3.3.3 Level-Ordered Unary Degree Sequence (LOUDS)

The final column of Table 3.2 shows how some operations can be solved under the LOUDS representation. We assume that, just as for DFUDS, a tree node corresponds to the first position of P where we write its arity in unary. The formulas differ slightly from the original version [Jac89], but not significantly in terms of performance. Again, *first_child* and *next_sibling* can be expressed in terms of the others.

Other operations are not immediately supported. We discuss, however, some practical alternatives. To begin, *rank_l/select_l* provide a usually sufficient alternative to *pre_rank/pre_select*, as they map the tree nodes to consecutive numbers in $[1, n]$. In most cases the preorder primitives are used not because preorder is particularly relevant, but just for the sake of storing satellite information associated to the node identifiers in $[1, n]$. For this purpose, level-wise numbering is as good as preorder.

Although *ancestor(x, y)* is not supported, there is a property of LOUDS that simplifies a brute-force implementation: *if y is deeper than x, then y > x*. This property is easy to prove. LOUDS represents the tree in level order, so if $y > x$, then y is in a deeper level than, or in the same level of, x . If y is in the same level than x , then $y' = \text{parent}(y) < x$, so x cannot be an ancestor of y . Otherwise after computing $y' = \text{parent}^k(y)$, for $k > 1$, y' will be equal to x , in which case x is an ancestor of y , or $y' < x$, which would mean that y' is in the same level than x or it is in a higher level, so x cannot be an ancestor of y . Thus to compute *ancestor(x, y)*, while $y > x$, we take $y \leftarrow \text{parent}(y)$. At the end, either $x = y$ (in which case the answer is yes) or $y < x$ (in which case the answer is no).

This same property permits a lightweight brute-force implementation of *lca(x, y)*: If $y > x$ do $y \leftarrow \text{parent}(y)$; else if $x > y$ do $x \leftarrow \text{parent}(x)$. Continue until $x = y$, which is the answer. Note that in the worst case we will continue until $x = y = \text{root}$. This works because if $y > x$ then y is not an ancestor of x , hence $\text{lca}(x, y) = \text{lca}(x, \text{parent}(y))$, and symmetrically for x .

Another operation that LOUDS lacks is *subtree_size(x)*. Since the nodes belonging to the subtree of x are contiguous at each level, this can be implemented by carrying an interval $[y, z]$ along the levels and counting the number of nodes at each level (with $\text{rank}_l(z) - \text{rank}_l(y - 1) + 1$). Initially $[y, z] \leftarrow [x, x]$. For each next level we update $y \leftarrow \text{child}(y, 1)$ and $z \leftarrow \text{child}(z, \text{degree}(z))$, until $z < y$.

The other operations, *level_ancestor* and *depth*, are implemented by brute force using *parent* without any special improvements.

Implementation notes. We use González et al.'s *rank/select* solution [GGMN05] (*ggmn* of Section 2.3). Several special cases are avoided by prepending $'()$ ' at the beginning of P . In a previous practical study on LOUDS [DRR06] they choose differently the bit that represents a tree node. We found that they require more operations than us and than Jacobson [Jac89] (except that they get *next_sibling* almost for free) and their space overhead is too high (50%-100%).

Labeled trees. These are solved just as for DFUDS (reversing the order of children is not necessary).

3.3.4 A Fully-Functional Representation (FF)

The operations in Table 3.2 are solved as for BP, since the parentheses are set in BP order. Furthermore, we have shown how to implement operation *level_ancestor* using the same primitives *fwd_search* and *bwd_search*, as well as several others not considered here [SN10].

The range min-max tree also solves easily operation *rmq(x, y)*, and thus *lca(x, y)*: The area $P[x, y]$ is covered by $O(k \log_k(n/b))$ (super)blocks, which can be scanned in constant time for the minimum excess value. By storing also the number of minima, we can in a similar way compute *degree(x)* (which is the number of excess minima in $P[x + 1, findclose(x) - 1]$), *child(x, i)* (which is the i -th excess minimum in $P[x + 1, findclose(x) - 1]$), and *child_rank(x)* (which is the number of excess minima in $P[parent(x), x - 1]$).

Implementation notes. The number of minima in each (super)block is stored as described for $m[i]$ and $M[i]$, thus using other $\approx 3.52\%$ overhead. Other small-space structures can be added to support further operations [SN10].

Labeled trees. We use $L[1, n]$ as for BP. Yet, both binary (using *child(x, i)*) and sequential search are possible with the RMM. We test both.

3.4 Experimental Comparison

We performed our experiments on trees coming from (a) LZ78 parsings of text collections, (b) suffix trees of text collections, and (c) the tag structure of XML collections. For cases (a) and (b) we used 200 MB and 50 MB (respectively) prefixes of the the DNA, protein and XML texts from the Pizza&Chili corpus (<http://pizzachili.dcc.uchile.cl>). The resulting trees have 16,373,737, 30,080,186, and 11,775,736 nodes respectively for case (a), whereas for (b) they have 75,095,011, 72,396,959 and 70,723,755 nodes. For case (c), we used an XMark [SWK⁺02] document of about 558 MB, a Medline document (<ftp://ftp.cogsi.ed.ac.uk/pub/disp/OHSXML.tar.gz>) of about 380 MB, and the XML text form Pizza&Chili, of 200 MB. The resulting tree structures have 14,238,042, 3,966,917, and 9,560,024 nodes, respectively.

We tested the performance of the tree operations by emulating random root-to-leaf traversals of trees (root itself excluded), where in addition every other child of the current node in the path is visited with a given probability p . We retain the order in which such nodes are visited, so $p = 0.0$ corresponds to a pure random root-to-leaf traversal, $0 < p < 1$ simulates range- or approximate-search traversals, and $p = 1$ simulates a full traversal of the tree. The general behavior is quite

similar within each class of trees (a), (b) and (c), so for LZ78 we only show the case of DNA text (and later, once, proteins), while for suffix trees we only show the case of proteins, and for XML the XMark document. Table 3.3 gives some of their characteristics.

	LZ78 DNA	LZ78 Protein	S. tree Protein	XML XMark
Size	16,373,737	30,080,186	72,396,959	14,238,042
Height	64	126	173	14
Avg. leaf depth	14.21	7.79	11.15	7.76
Max.arity	16	25	26	127,500
Avg.arity	2.01	2.63	2.90	2.11

Table 3.3: Basic characteristics of the trees tested.

In general, as p grows, a higher proportion of small and nearby subtrees are visited. This translates into more close matching parentheses and more cache hits in BP and DFUDS. As closer parentheses are found faster in both schemes, such structures become faster as p grows. LOUDS, instead, is generally insensitive to the subtree sizes, as it proceeds levelwise (indeed, it is slightly slower on deeper levels, as the parent/children are farther away).

The computer used features an Intel(R) Core(TM)2 Duo processor at 3.16 GHz, with 8 GB of main memory and 6 MB of cache, running version 2.6.24-24 of Linux kernel.

We first measure the times of the core operations, which allow predicting the performance of many operations in Table 3.2. Some specific tree operations are studied later.

3.4.1 Core Operations

Figure 3.1 shows the space/time trade-offs achieved for the core parenthesis and bitmap operations, for the cases $p = 0.0$ and $p = 0.2$. The times are averaged over at least 20,000 nodes (for $p = 0.0$), and up to 20 million for larger p values. We show the times for *findclose* and *enclose*. For the RP representation we use block sizes $b = 32, 64, 128, 256, 512$. For HB we use block sizes $b = 128, 256, 512, 1024, 2048$. For the RMM representation we set $b = 512$ and $k = 8, 16, 32, 64, 128$ (other values of b did not yield better results). These combinations yield the observed space/time trade-offs.

Modifier “DFUDS” means that the parentheses sequence used is that of DFUDS rather than BP. RP-DFUDS was very close to RP, which is clearly the worst performer as we comment soon, so we omitted it to lighten the plots. On XMark, BP-DFUDS needs more than 9 bits per node and thus it does not appear in the XML plots. We also omitted RMM-DFUDS from all these plots as it was indistinguishable from RMM.

We also show the time for *rank* and *select*, which offers its own space/time trade-off (with the same space one solves both) and *prev_j/next_j*, which is drawn as a line because it uses brute force and hence does not need any extra space. Operations *prev_j/next_j* were tested over the nodes traversed in the DFUDS sequence (which is equivalent to having chosen LOUDS), as these are

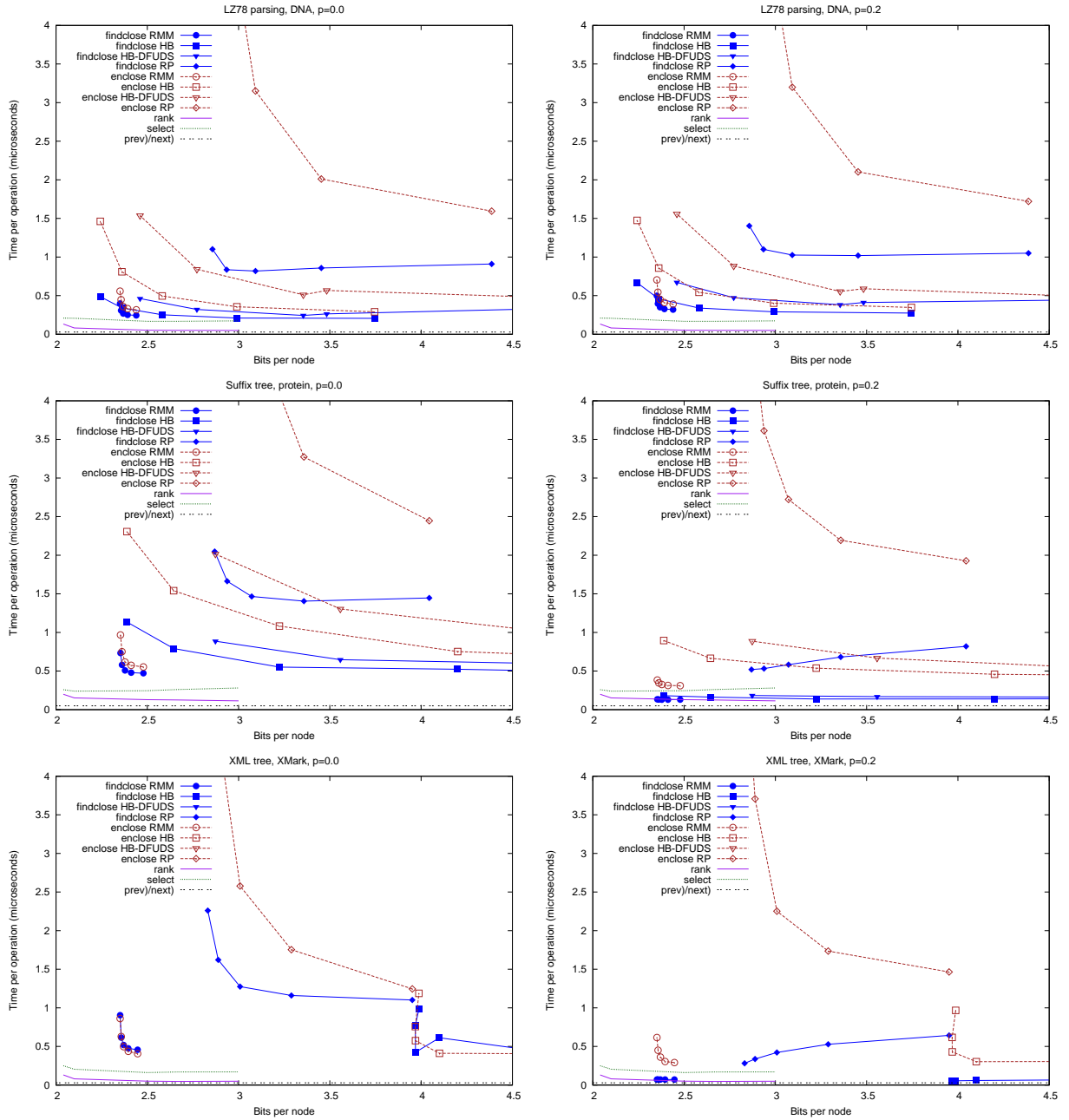


Figure 3.1: Space/time for the core operations on parenthesis structures, for $p = 0.0$ (left) and $p = 0.2$ (right).

the representations where such operations are applied.

The times for *findopen* are similar to those for *findclose* on BP. On DFUDS, instead, they are asymmetric: *findclose*(x) is usually faster as it returns a typically closer node. Yet, the operations *findopen*($x - 1$) of Table 3.2 have a cost very similar to *enclose*(x) (which usually returns *findopen*($x - 1$) - 1).

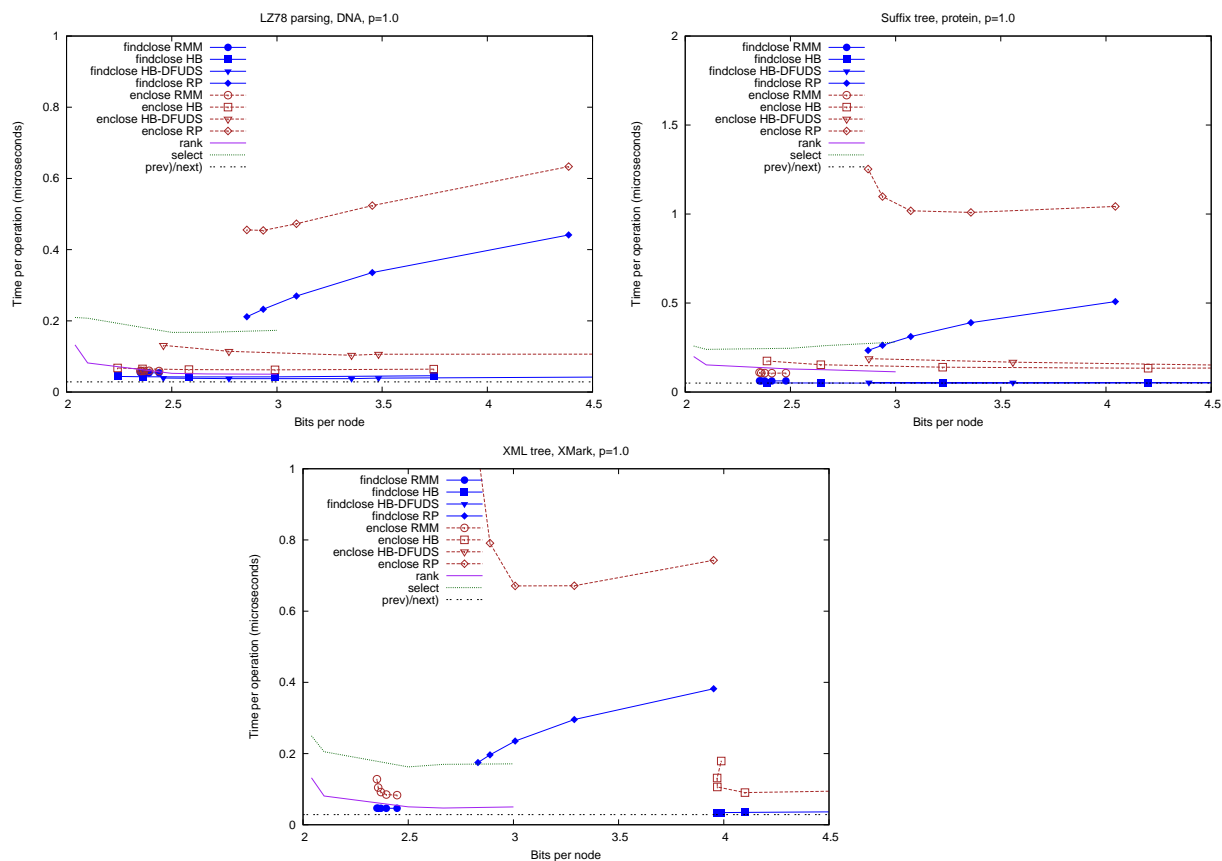
Clearly RMM (and RMM-DFUDS) is in general the best performer, both in space and time. HB is generally able of approaching its time, yet the space required by HB varies widely depending on the tree shape. On DNA, where the arity is low, HB can improve upon RMM in space, while being competitive in time (at least for *findclose*). In proteins, where the arity is larger, many parentheses become far (especially for *enclose*) and the space of HB cannot anymore reach that of RMM. The effect is dramatic on XMark, where the arity is much higher and HB is unable of operating with less than 100% space overhead over the $2n$ bits. In the DFUDS variants of HB the parentheses tend to be farther than on BP, thus HB-DFUDS requires more space than HB(-BP). In XMark, HB-DFUDS has a space overhead of more than 450%.

This shows that the lack of space guarantees of HB is indeed a problem in practice, and emphasizes the importance of offering such guarantees in fundamental data structures, where one can hardly predict the type of instances that will be faced.

The time performance of RP is clearly the worst. RP is affected by the operations on the compressed bitmap, which in practice pose a significant time overhead, as well as space redundancy over the entropy. This is illustrated, in particular, by the fact that, on *findclose* for $p > 0$, where many small subtrees are handled, RP in many cases worsens in time when using more space. The reason is that, when the answer is not too far away, it is better to find it sequentially through sparser blocks than to use the heavy machinery of the compressed bitmaps.

Finally, note that the extra space for *rank/select* is the only one required by LOUDS, and that it works well within just 2% of extra space. This means that, within the small set of operations it supports, LOUDS is competitive in time and better in space than RMM. Note in particular that *select* may worsen when using more space. This is because the binary search is done over a denser (and larger) set of samples, whose impact on the locality is not recovered by scanning a shorter segment of the bitmap (as this scan is cache-friendly).

Figure 3.2 shows the extreme case of $p = 1.0$, which simulates a full tree traversal. Here most of the extra structures are useless, as the majority of the operations are carried out on very small trees, where they are easily solved by brute force. In this case LOUDS representation (that is, *rank/select*) is not that fast compared to the others, as this kind of locality does not show up in its levelwise traversal.

Figure 3.2: Space/time for the core operations on parenthesis structures, for $p = 1.0$.

3.4.2 Tree Operations

We tested several operations, a few to show that their times can be predicted with those of the core operations, and most because they are harder to predict: $parent(x)$, $child(x, i)$, $labeled_child(x, s)$, $level_ancestor(x, d)$, and $lca(x, y)$. We choose one representative tree for each operation, as the conclusions are roughly the same for the others. Except for $lca(x, y)$, we chose one reasonable space/time point per structure. These choices are given in Table 3.4 (recall that FF-* corresponds to RMM).

Figure 3.3 gives the times for operation $parent(x)$ on the suffix tree for proteins. We use the same sampling method (i.e., via tree traversals) of the core operations, with varying p . As in these, we average the times over at least 20,000 nodes. FF-BP outperforms the other alternatives except for $p = 0.0$, where LOUDS is faster. For higher p (where more smaller subtrees are chosen) LOUDS does not benefit from locality, as explained. The difference between HB-DFUDS and HB-BP is small, and can be attributed to the extra $prev$ operation. The difference between FF-DFUDS and FF-BP is larger.

Repres.	Params.	LZ78 DNA	LZ78 Protein	S. tree Protein	XML XMark
FF-BP and FF-DFUDS	$b = 512,$ $k = 32$	2.37	2.37	2.38	2.37
HB-BP	$b = 512$	2.58	2.75	3.22	3.97
HB-DFUDS	$b = 512$	3.35	4.17	4.84	9.52
RP-BP	$b = 64$	3.45	3.33	3.36	3.29
LOUDS	$s = 640$	2.10	2.10	2.10	2.10

Table 3.4: Default space usage for the representations in Section 3.4.2.

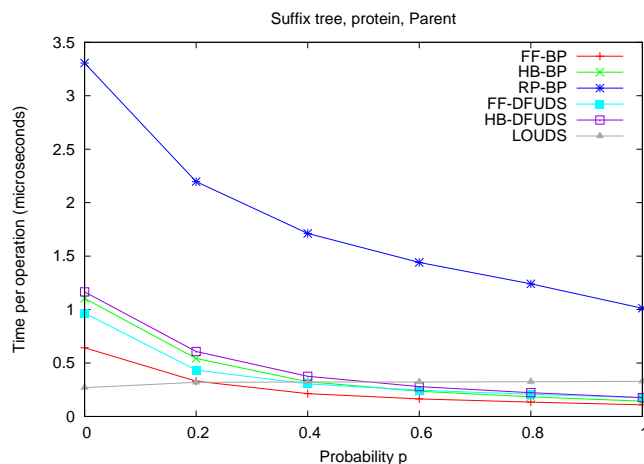
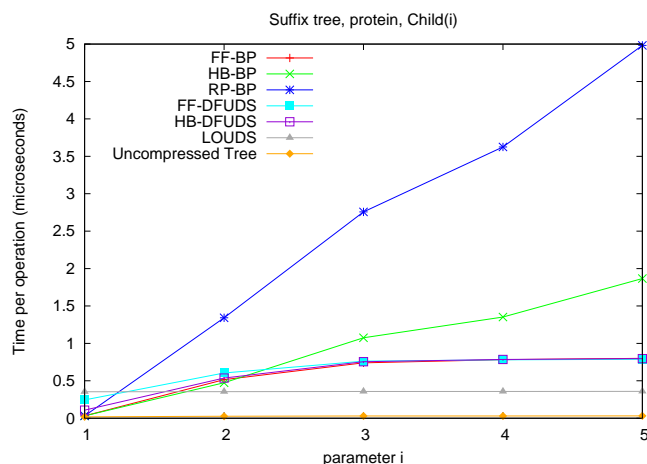
Figure 3.3: Performance for operation $parent(x)$ as a function of p .

Figure 3.4 shows the times for operation $child(x, i)$ on the suffix tree of proteins, for $1 \leq i \leq 5$. We choose nodes from the same traversal done for $p = 0.0$ on the core operations, keeping all nodes with degree at least 5. This gives us a sampling of 14,000 nodes to average over. This time we have also included the times over a pointer-based uncompressed tree, which will be discussed in Section 3.4.4. That time is essentially that of a single memory access, and is close to the time of $first_child(x) = child(x, 1)$ on BP representations. For larger i , as HB-BP and RP-BP operate by brute force, they are essentially $O(i)$, though with different constants (related to their time to carry out $findclose$). LOUDS is clearly $O(1)$ for this operation, and faster than all the others except for the trivial case $i = 1$ (which, on the other hand, is the dominant case on various common traversals). HB-DFUDS is also $O(1)$ in theory (as it always needs one $findclose$ operation), yet it is faster for smaller i values because the answer is closer in those cases. The time of FF-DFUDS is in practice logarithmic on the distance of the answer, as it climbs the range-min-max tree. FF-BP uses its own algorithm for $child$, and its time is also logarithmic on the distance of the answer (and thus very similar to FF-DFUDS). This explains the slow growth of these alternatives. All the three also get closer to LOUDS for larger p (this plot is for $p = 0.0$, as explained), finally surpassing it around $p = 0.6$.

Figure 3.5 illustrates the performance for operation $labeled_child(x, s)$ on the LZ78 parsing

Figure 3.4: Performance for operation $child(x, i)$ as a function of i .

of proteins. We use exactly the same method of sampling nodes of $child(x, i)$, and also plot the times as a function of i , so this time the argument s is the one resulting in descending to the i -th child. This is an operation where LOUDS excels, DFUDS does well, and FF-BP falls far behind. The binary search of LOUDS and DFUDS is done directly over a range in L , which makes them very fast, much faster than the linear searches of BP alternatives (still DFUDS and LOUDS need to finish with one $child(x, i)$ operation after finding i). The time differences among LOUDS and DFUDS alternatives are lower than for $child(x, i)$ because all must carry out the same binary search. On the other hand, the linear times of the BP alternatives are slightly costlier than for $child(x, i)$ because, although they probe the same nodes, they must carry out one further *rank* operation per node to find the letter in L . They are only faster in the lucky case of s labeling the first child. Finally, FF-BP is surprisingly slow. Although it carries out a binary search, this search is not local and requires various $child(x, i)$ operations, which renders it slower than some linear searches in practice. Indeed, it is much better to linearly scan for the i -th child using *findclose*, as illustrated by FF-Seq-BP, which is the fastest among the $O(i)$ methods.

Figure 3.6 compares the performance for operation $level_ancestor(x, d)$ on the LZ78 tree of DNA, for $1 \leq d \leq 5$. We choose all the nodes from the sampling of core operations corresponding to $p = 0.0$, and keep the nodes with depth at least 5. This leaves us 2,000 nodes to average over. Here all the techniques except FF-BP use brute force (note FF-DFUDS cannot benefit from the FF-BP algorithm, as excesses do not correspond to depths in DFUDS). The times, consequently, are basically linear, except for FF-BP, whose algorithm based on operation *bwd_search* is logarithmic in the distance to the answer. This dominates all the other costs and shows its practicality. Note the times are about 50% faster than the corresponding to $parent(x)$ on the suffix tree of proteins, as these nodes have fewer children and hence their parents are closer.

Finally, Figure 3.7 shows operation $lca(x, y)$ on XMark, for different memory usage of the structures. This time we simply chose 100,000 pairs of tree nodes at random. This operation is

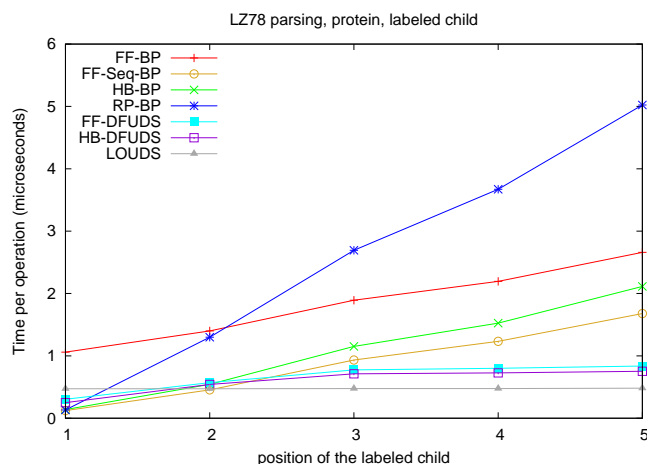


Figure 3.5: Performance for operation $labeled_child(x, s)$ as a function of i , the position of the resulting child.

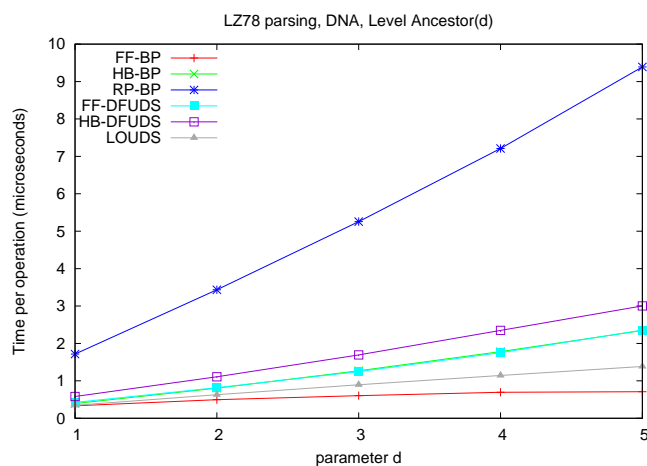


Figure 3.6: Performance for operation $level_ancestor(x, d)$ as a function of d .

implemented by brute force on all representations except FF-BP and FF-DFUDS, which provide a native solution to lca . It can be seen that FF and LOUDS (with its lightweight brute-force algorithm) sharply dominate the space/time trade-off. FF-DFUDS is slightly slower than FF-BP because it needs an extra check that invokes an $ancestor$ query. Using the rmq -based solution for the other alternatives would achieve competitive times (0.9 microseconds according to a simple experiment we carried out), yet it would require including the extra constant-time rmq structure [FH07], which adds $6.12n$ bits to the already large HB and RP representations.

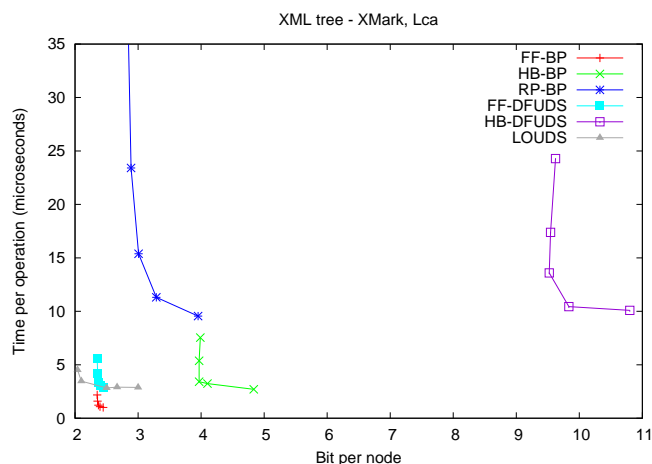


Figure 3.7: Performance for operation $lca(x, y)$ as a function of the space usage.

3.4.3 Asymptotics

Our experiments have considered fixed trees. While all the operations are in theory constant-time, in practice the implementation of *select* is $O(\log n)$, as well as our FF operations (while the theoretical article [SN10] achieves constant time, our FF implementation builds solely on the RMM, which seems much more practical but requires time logarithmic on the tree size). Furthermore, cache effects can significantly affect the performance. Thus, one may wonder how the tree size n affects the results given.

Figure 3.8 shows the times for operations *findclose* and *enclose* as a function of the tree size n . For this sake, we created random binary trees of increasing size, from 50 to 350 million nodes, and tried the BP alternatives on them. We maintained the space usage parameters of Table 3.4, obtaining 2.37–2.38 bits per node for RMM, 2.62–2.99 for HB, and 3.68–3.74 for RP. The nodes to operate are collected over 100,000 random root-to-leaf traversals (akin to $p = 0.0$ above).

As it can be seen, apart from some oscillations, there is no clear increase in the time as we handle larger trees. This is due in part to the fact that many operations (most clearly those of RMM) depend not on the total tree size, but on the distance between the query and the answer node, and this average distance varies very mildly on n even for the traversal with $p = 0.0$.

This result confirms the scalability of the solutions and the stability of our results on larger trees. Note in passing that *findclose*(x) is slower than *enclose*(x) on binary trees, as half the times the answer of the latter is simply $x - 1$.

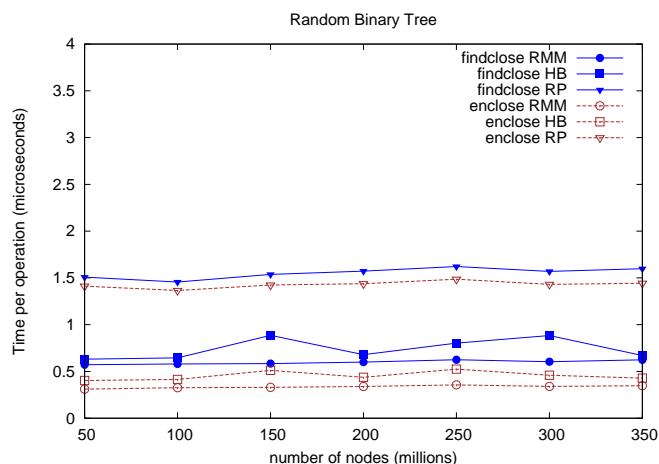


Figure 3.8: Performance as a function of the tree size, for random binary trees.

3.4.4 Non-Succinct Tree Representations

A natural question is how the times we have shown (usually below the microsecond per operation, for the best performing structures) compare to a plain pointer-based tree representation. For this sake, we set up such a tree, where each node has a pointer to an array of its children. This allows one to implement $child(x, i)$ using just one memory access. This takes around 18–31 nanoseconds in our machine and is illustrated in Figure 3.4.

This means that an uncompressed tree, in the cases where it can actually fit in main memory, is about 12–20 times faster than LOUDS, our best compressed representation for this operation. If we consider all the operations, and take FF-BP as the alternative (as it gives the widest support), we have that in most cases the time of FF-BP is within half a microsecond (30 times slower than the single memory access needed by the uncompressed representation) and for some complex operations, up to 2 microseconds (120 times slower).

Still, it is important to illustrate in numeric terms what it means to be succinct or plain, especially when we need more functionality than moving to a child. A plain representation on a 32-bit machine spends 32 bits per node (that is, 13 times more than FF-BP and 15 more than LOUDS), if we arrange the tree in levelwise order on an array, and the cell of each node points to the cell of its first child. This arrangement supports efficiently the most basic navigation: $child$, $degree$, $isleaf$, $first_child$, and $next_sibling$. However, *each* other operation we wish to support efficiently requires another integer: $parent$, $depth$, $subtree_size$, pre_rank , etc. Thus, just supporting these four operations we would require 160 bits per node. The more functionality we wish to support, the less likely is that the pointer-based tree will fit in main memory, in applications managing massive trees. In contrast, the powerful FF-BP representation supports all those operations within the same space (less than 2.4 bits per node).

The more sophisticated operations, like *lca* and *level_ancestor*, deserve a special mention. They can be solved within $O(n)$ -word space and $O(1)$ time, but they require several (not just 1) further integers per node. Worse than that, they require several memory accesses. For example, the classical *lca* algorithm [BFC00] requires 7 extra integers (224 bits) per node and 12 non-local memory accesses. Thus its expected time is around 0.2–0.4 microseconds, whereas FF-BP solves *lca* within the microsecond (just up to 5 times slower) and the same 2.4 bits per node.

On the other hand, there are intermediate tree representations [Jac89, BBK04, GHSV06] which, although do not guarantee $2n + o(n)$ bits of space usage (only $O(n)$ bits in general), offer a relevant trade-off in practice. Translated into a general tree representation, they usually boil down to a scheme as follows: Arrange the nodes of the tree on an array $T[1, n]$ by performing a preorder traversal. The cell of each node stores where is its next sibling. Unlike our previous uncompressed scheme, akin to a LOUDS layout, this resembles more BP: we immediately have $first_child(i) = i + 1$ and $next_sibling(i) = T[i]$ (yet, as before, we lack most of the other operations unless we spend the space of storing the answers explicitly). By storing the pointers in relative form and using a variable-length encoding, compression is possible since most of the subtrees are small. The value of the relative pointer from node x to its next sibling will be just $subtree_size(x)$.

To measure the practical performance of this idea, we computed $\sum_{x \in T} \lceil \log(subtree_size(x) + 1) \rceil$ on our trees, which gives a lower bound to the size that can be achieved by this technique. It is shown in the first line of Table 3.5. It is a lower bound because (i) storing the exact number of bits required to encode the number is impossible (a representation like, say, γ -encoding, is necessary); and (ii) since a variable-length code is used, the pointers must point to bit offsets, not to cell numbers of T .

Repres.	LZ78 DNA	LZ78 Protein	S. tree Protein	XML XMark
gaps	1.62	1.46	1.36	1.59
γ -code	2.24	1.92	1.72	2.18
real γ	3.43	2.72	2.44	2.90

Table 3.5: Space usage for the larger representations with minimum functionality. Only the last column is a real encoding scheme.

The second row of Table 3.5 addresses problem (i) using γ -codes, which gave us good results. It shows the value $\sum_{x \in T} |\gamma(subtree_size(x))| = \sum_{x \in T} (2 \lceil \log(subtree_size(x) + 1) \rceil - 1)$. The last line of the table addresses problem (ii), only now reaching a real solution to the problem. It γ -encodes not the subtree size of x , but the length of the representation of the subtree of x (which recursively uses γ -codes, a leaf using $|\gamma(1)| = 1$ bit). One adds this value to the pointer to x and reaches the bit-offset of its next sibling.

As it can be seen, these representations are space-efficient in practice, competing with our $2.4n$ -bit representations, and possibly being very fast for operations *first_child*, *next_sibling*, and *isleaf*, that is, for a full tree traversal. Yet, as we have seen in Figure 3.2, one can do pretty well with the plain $2n$ parentheses for such traversal. Still, assuming that *next_sibling* will be

significantly faster by decoding a γ -code, the representation offers a rather limited repertoire of operations. By somehow mixing fixed- and varying-length representations one could include *child* and *degree*, at some price in space. Yet, as before, each further operation to support requires storing extra data that is already included in our $2.4n$ -bit representations (and many are also included in our $2.1n$ -bit LOUDS format).

This shows that the key advantage in the modern succinct tree representation space is not their $2.x$ -bit space usage, as this is not that difficult to achieve, but rather the wide functionality that these succinct representations support within this space.

Chapter 4

Compressed Suffix Trees

A *compressed suffix tree (CST)* is obtained by enriching the compressed suffix array with some extra data. In order to get a suffix tree from a suffix array, one needs at least two extra pieces of information: (1) the tree topology; (2) the *longest common prefix (LCP)* information (see Section 2.7). Indeed, the suffix tree topology can be implicit if we identify each suffix tree node with the suffix array interval containing the leaves that descend from it. This range uniquely identifies the node because there are no unary nodes in a suffix tree.

4.1 Sadakane’s Compressed Suffix Tree

Sadakane’s compressed suffix tree [Sad07a] was the first representation that used linear size, that is $O(n \log \sigma)$ bits, and supported all the navigation operations over the tree efficiently. It builds on top of the compressed representation for the suffix array [Sad03], described in Section 2.6.1. In addition his structure needs a representation for the topology of the tree and the *LCP* data. For the topology he uses a balanced parentheses encoding of a tree, that we will call P (similar to that presented in Section 3.2.1). Since there are at most $2n - 1$ nodes in a suffix tree for a text of length n , i.e., exactly n leaves and at most $n - 1$ internal nodes, the suffix tree of the text can be encoded in at most $4n + o(n)$ bits, where $o(n)$ is the extra space used for supporting *rank*, *select*, *access* (see Section 2.3) and the tree operations over the sequence (see Chapter 3). The *LCP* was compressed into a bitmap H of $2n$ bits, by noticing that, if sorted by text order rather than suffix array order, the *LCP* numbers decrease by at most 1. For further details see Section 5.1.

Figure 4.1 shows an example of the basic components of Sadakane’s suffix tree (for simplicity we show A and LCP in plain form).

Another structure that Sadakane’s implementation needs for solving operations like *LCA* and *SLink*, is one for answering *range minimum queries* over the array P' , where $P'[i] = \text{rank}_\zeta(P, i) -$

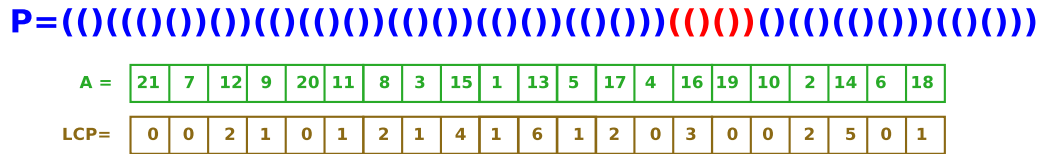
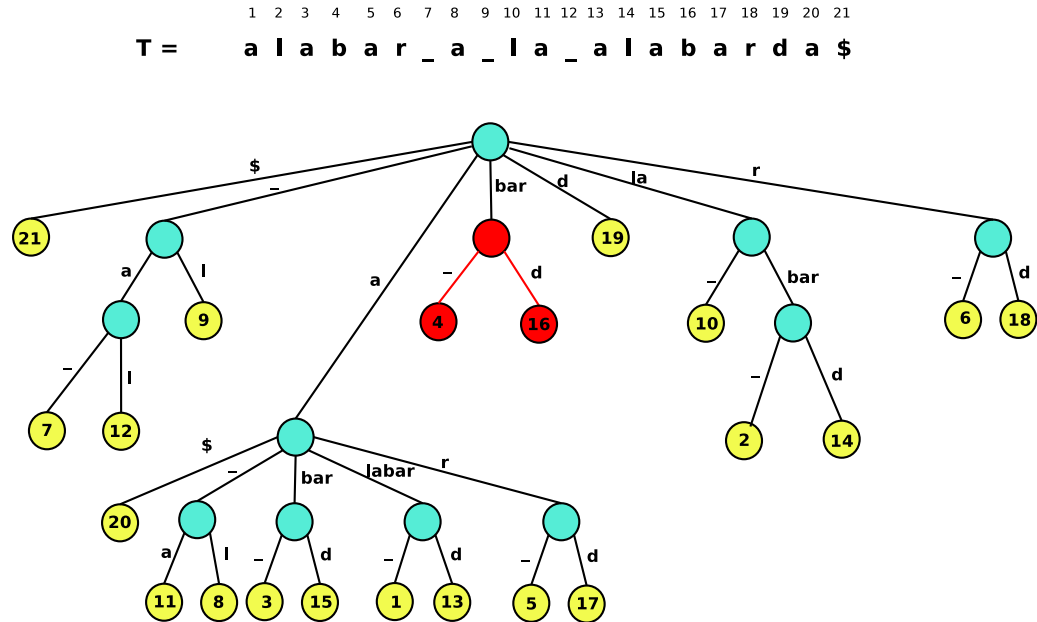


Figure 4.1: Sadakane's compressed suffix tree of the text "alabar a la alabarda\$".

$rank_y(P, i)$. Note that P' is not stored, as each element of it is computed in constant time using operation $rank$ over the balanced parentheses sequence P when necessary.

Definition 4.1. For indices l and r , between 1 and n , of an array G , the *range minimum query* $RMQ_G(l, r)$ returns the index of the smallest element in the subarray $G[l..r]$. If there is a tie we choose the leftmost one.

Sadakane divides the whole array P' into blocks of size $\log^3 n$, and defines an array $L'[0, n/\log^3 n]$ such that $L'[i]$ stores the minimum value in the i -th block. To compute $RMQ_{P'}(x, y)$ he defines the next algorithm:

- calculate the minimum of $P'[x..e]$ and its index, where e is the last element in the same block as x .
- calculate the minimum of $P'[s..y]$ and its index, where s is the first element in the same block as y .

- calculate the minimum of $L'[(x' + 1)..(y' - 1)]$ and its index, where $x' + 1$ is the index of the block that follows the block of x and $y' - 1$ is the index of the block that precedes the block of y .

For the third step Sadakane constructs a two-dimensional table M , where $M[i, k]$ ($i = 0, 1, \dots, n/\log^3 n$, $k = 0, 1, \dots, \lfloor \log n \rfloor$) stores the index of the minimum value in $L'[i, i + 2^k - 1]$. Then the index of the minimum value between $L'[x' + 1]$ and $L'[y' - 1]$ can be computed in constant time by $\min\{M[x' + 1, k], M[y' - 2^k, k]\}$ where $k = \lfloor \log(y' - x' - 2) \rfloor$. To compute the minimum in a block Sadakane divides the block into subblocks of size $\frac{\log n}{2}$. For each j -th block ($j = 0, 1, \dots, n/\log^3 n$) he creates a table $M_j[i, k]$, where $i = 0, 1, \dots, 2\log^2 n - 1$ and $k = 0, 1, \dots, \log(2\log^2 n)$. These tables occupy $o(n)$ bits in total. So the first and second steps can be computed in constant time using these tables. Therefore, using the precomputed tables and the two-dimensional table M Sadakane is able to compute RMQ_P in constant time using $o(n)$ extra space [Sad07a].

Following we explain how Sadakane implements the operations defined in Table 2.2 for navigating the suffix tree:

- **Root**: The root node is the first open parenthesis in the sequence, therefore we return the position 1.
- **Locate**(v): If $v + 1 =)'$ (i.e., v is a leaf) return the value of the position w of the suffix array, where $w = \text{rank}_\cup(P, v)$. Otherwise return -1 . To compute rank_\cup Välimäki et al. [VGD07] added a bitmap $brLeaf$ where the “ $()$ ”s of P are marked, so $\text{rank}_\cup(P, i) = \text{rank}_1(brLeaf, i)$.
- **Ancestor**(v, w): If $v \leq w \leq \text{findclose}(P, v)$ then return true. Otherwise return false.
- **SDepth**(v): If $v = 1$ return 0. If v is a leaf return $n - \text{Locate}(v)$. Otherwise the answer is $LCP[i + 1]$, where $i = \text{inorder}(v) = \text{rank}_\cup(P, \text{findclose}(P, v + 1))$ is the inorder position of the node v .
- **TDepth**(v): The number of opening minus closing parentheses in the sequence P until v , i.e., $\text{rank}_\cup(P, v) - \text{rank}_\cup(P, v) = 2\text{rank}_\cup(P, v) - v$.
- **Count**(v): Return $\text{rank}_\cup(P, \text{findclose}(P, v)) - \text{rank}_\cup(P, v)$.
- **Parent**(v): Return $w = \text{enclose}(v)$ unless v is the root. In that case return -1 .
- **FChild**(v): If $P(v + 1) =)'$ then node v is a leaf and has no children. Otherwise return $v + 1$.
- **NSibling**(v): Let $w = \text{findclose}(P, v) + 1$. If $P(w) =)'$ then node v has no next sibling. Otherwise return w .

- **SLink**(v): First we compute the leftmost and rightmost leaves of the subtree rooted at node v , $x = \text{rank}_\emptyset(P, v - 1) + 1$, and $y = \text{rank}_\emptyset(P, \text{findclose}(P, v))$. Then we move to the leaves that represent the positions next to x and y , i.e., x' and y' such that $T_{A[x']} = T_{A[x]+1}$ and $T_{A[y']} = T_{A[y]+1}$. This can be done using Ψ and select_\emptyset : return $\text{LCA}(\text{select}_\emptyset(P, \Psi(x)), \text{select}_\emptyset(P, \Psi(y)))$. Note that $\text{select}_\emptyset(P, i) = \text{select}_1(\text{brLeaf}, i)$, where brLeaf was defined in operation `Locate`.
- **SLink** ^{i} (v): Return the value resulting of applying i times *SLink* recursively same as above but instead of $\Psi(z)$ use $\Psi^i(z) = A^{-1}[(A[z] + i) \bmod n + 1]$. In practice he computes Ψ^i iteratively using Ψ , as this is faster up to relatively large values of i .
- **LCA**(v, w): If one of the nodes is ancestor of the other then the answer will be that node. This is easily verified using *Ancestor*. Otherwise return $\text{Parent}(\text{RMQ}_{P'}(v, w) + 1)$. The range minimum query returns the index m of the minimum element in $P'[v..w]$. Then $P[m] = '('$ and $P[m + 1] = ')'$ always hold. Therefore $P[m + 1]$ is the open parenthesis of a child of $\text{LCA}(v, w)$, and that is why we compute *Parent* at the end.
- **Child**(v, a): We iteratively get all the children of the node v . In each step we recover the first character of the edge-label between v and a child w and compare with a until we find the desired child. Note that the edge-label between w and v is represented by $T[A[i] + d_1, A[i] + d_2 - 1]$, where $i = \text{inorder}(w)$, $d_1 = \text{LCP}[\text{inorder}(v)]$, and $d_2 = \text{LCP}[i]$.

Sadakane's compressed suffix tree supports most of the tree navigation operations in constant time, except for *SDepth*, which takes $O(\log^{1+\epsilon} n)$ time in practice (i.e., computing $A[i]$), and *Child*, which takes $O(\sigma \log^{1+\epsilon} n)$ time.

4.2 Russo et al.'s Compressed Suffix Tree

Russo et al. [RNO08b] recently proposed a new suffix tree representation, called *FCST* (Fully-Compressed Suffix Tree), requiring $nH_k + o(n \log \sigma)$ bits of space for any $k \leq \alpha \log_\sigma n$, where $0 < \alpha < 1$ is any constant. This representation gets rid of the balanced parentheses that describe the topology, by instead identifying suffix tree nodes v with their corresponding suffix array intervals, $v = [v_l, v_r]$. The main idea is to sample some suffix tree nodes and use the compressed suffix array as a tool to find nearby sampled nodes. So this representation is built on top of a compressed suffix array, the most adequate one for this task being the *alphabet-friendly FM-index* (Section 2.6.2). Their structure is slower, in general, by an $O(\log n (\log \sigma + \log \log n))$ factor than Sadakane's compressed suffix tree. On the other hand the space required is very low and it was the first to break the $\Theta(n)$ -bits space barrier.

As we mentioned a few sampled nodes are stored. They use a sampling factor δ , so that the total sample is of size $O(\frac{n \log n}{\delta})$. The sampled tree can be represented using $o(n)$ bits, if $\delta = \omega(\log n)$. They assume that $\delta = \lceil \log n \log \log n \rceil$.

Definition 4.2. A δ -sampled tree S of a suffix tree τ with t nodes is formed by choosing $s = O(t/\delta)$ nodes of τ so that for each node v of τ there is an integer $i < \delta$ such that node $SLink^i(v)$ is sampled.

This means that if we start at a node v and apply $SLink$ recursively, we will find a sampled node in at most δ steps. Note that this means that the root of the suffix tree must be part of the sampled nodes.

In order to make an effective use of the sampled tree, Russo et al. use a way to map any node v to its *lowest sampled ancestor*, $LSA(v)$. Another important operation is the *lowest common sampled ancestor* $LCSA(v, w) = LSA(LCA(v, w))$. These operations are used to map from suffix tree nodes to sampled nodes as well as the reverse. They support both operations in constant time using $O((n/\delta) \log \delta)$ bits of space. Moreover for all sampled nodes v they store the values $SDepth(v)$ and $TDepth(v)$. They also support operation $Parents_S(v)$, that gives the node w that is the parent of v if we only consider the sampled nodes.

Summarizing, the *FCST* can support all the navigational operations and the extra space fits within $o(n)$ extra bits, which is negligible compared to the $o(n \log \sigma)$ of the *CSA*. The time complexities for most operations are logarithmic at best. The central tools for computing these operations are a particular sampling of suffix tree nodes, its connection with the suffix link and the *lowest common ancestor (LCA)* query, and the interplay with the compressed suffix array.

To explain how Russo et al. implement the navigation operations we will need first to extend the function LF given in Section 2.6.2.

Definition 4.3. $LF(X, v)$, where X is a string and v a position in the suffix array, gives the lexicographical rank of the suffix $X.T_{SA[v], n}$ among all the suffixes, whether it exists or not. This definition is easy to extend to suffix tree nodes v : $LF(X, v) = LF(X, [v_l, v_r]) = [LF(X, v_l), LF(X, v_r)]$.

Now we can explain how they implemented all the navigation operations. The order in which we present the operations is given by the dependencies between them.

- **Root/Count/Ancessor:** For $Root()$ return the interval $[1, n]$, $Count(v)$ is simply $v_r - v_l + 1$, $Ancessor(w, v)$ is true iff $w_l \leq v_l \leq v_r \leq w_r$.
- **Locate(v):** If v is a leaf ($v_r = v_l$), then return the suffix array value at position v_l . Otherwise return $NULL$.
- **SDepth(v):** If v is a sampled node the answer is already stored. Otherwise return $\max_{0 \leq i \leq \delta} \{i + SDepth(LCSA(\Psi^i(v_l), \Psi^i(v_r)))\}$. The *FM-Index* can compute Ψ as the inverse of LF .
- **LCA(v, w):** If one of v or w is ancestor of the other, return this ancestor node. Otherwise note that computing $LCA(v, w)$ is equivalent to computing $LCA(\min\{v_l, w_l\}, \max\{v_r, w_r\})$.

$\max\{v_r, w_r\}$). With this we move the operation to the leaves where if v' is a leaf then $SLink(v') = \Psi(v')$. Using this and LF we can compute $LCA(v, w)$ by returning the interval $LF(v[0, i - 1], LCSA(\Psi^i(\min\{v_l, w_l\}), \Psi^i(\max\{v_r, w_r\})))$, where i is the value that maximizes $SDepth(v)$.

- **Parent**(v): Returns either $LCA(v_l - 1, v_l)$ or $LCA(v_r, v_r + 1)$ whichever is lowest. Note that $Parent(v)$ must contain $[v_l - 1, v_r]$ or $[v_l, v_r + 1]$. If one of these nodes is undefined, either because $v_l = 1$ or $v_r = n$, then the parent is the other node. If both nodes are undefined then node v is the root, which has no parent.
- **SLink**(v): Return $LCA(\Psi(v_l), \Psi(v_r))$.
- **SLink** ^{i} (v): Return $LCA(\Psi^i(v_l), \Psi^i(v_r))$.
- **TDepth**(v): To compute $TDepth$ we need to add more nodes to the sampled tree so as to guarantee that, for any suffix tree node v , $Parent^j(v)$ is sampled for some $0 \leq j < \delta$. Given this $TDepth(v)$ returns $TDepth(LSA(v)) + j$, where $TDepth(LSA(v))$ is already stored and $j < \delta$ is obtained by iterating the operation $Parent$ from v until reaching $LSA(v)$. Note that for the next two operations we will also need the extended sampling.
- **LAQ_T**(v, d): If v is not a sampled node, we first go up in the tree using $Parent$, if we reach a node w where $TDepth(w) \geq d > TDepth(Parent(w))$ we found our answer. Otherwise we continue until reaching a sampled node, v' , then we binary search $Parent_S^i(v')$ to find the sampled node w' with $TDepth(w') \geq d > TDepth(Parent_S(w'))$. Finally from w' using $Parent$ by brute force we find our answer.
- **LAQ_S**(v, d): We start by binary searching for the value i such that $v' = Parent_S^i(SLink^{\delta-1}(v))$ satisfies $SDepth(v') \geq d - (\delta - 1) > SDepth(Parent_S(v'))$. Now we scan all the sampled nodes $v_{i,j} = Parent_S^j(LSA(LF(v[i..\delta.1], v')))$ with $SDepth(v_{i,j}) \geq d - i$ and $i, j < \delta$. This means that we start at node v' , follow LF , and reduce every node found to a sampled node using $Parent_S$ until the $SDepth$ of the node drops below $d - i$. Our aim is to find the $v_{i,j}$ that minimizes $SDepth(v_{i,j}) - (d - i) \geq 0$, and then apply the LF mapping to it.
- **FChild**(v): If v is a leaf, return $NULL$. Otherwise return $LAQ_S(v_l, SDepth(v) + 1)$.
- **NSibling**(v): If v is the root return $NULL$. Otherwise return $LAQ_S(v_r + 1, SDepth(Parent(v)) + 1)$.
- **Child**(v, a): The *generalized branching* for two nodes, v_1 and v_2 , consists in determining the node with path label $v_1.v_2$ if it exists. A simple solution is to binary search the interval of v_1 for the subinterval of w 's such that $\Psi^m(w) \in v_2$, where $m = SDepth(v_1)$. Using the same idea we are able to compute $Child(v, a)$, using v as v_1 and v_2 as the subinterval of the suffix array where the suffixes start with a . Note that this interval is given by $LF(a, Root())$.

- **Letter**(v, i): If $i = 1$ then return $charT[rank_1(D, v_l)]$ where D is a bitmap marking the first suffix in the suffix array starting with each different letter and $charT$ is an array of size σ where the letters that appear in $T_{1,n}$ are listed in alphabetical order. If $i > 1$, return $Letter(\Psi^{i-1}(v_l), 1)$. It is important to remark that structures $charT$ and D are already present, in one form or another, in the compressed suffix arrays (see, e.g., Section 2.6.1).

4.3 Fischer et al.'s Compressed Suffix Tree

Fischer et al. [FMN09] prove that Sadakane's bitmap H , which represents the LCP array (see Section 5.1), is compressible as it has at most $2R \leq 2(nH_k + \sigma^k)$ runs of 0s or 1s, for any k . For further explanations see Section 5.2. Another improvement over Sadakane's CST is to get rid of the tree topology and replace it with suffix array intervals. Fischer et al. show that all the navigation can be simulated by means of three operations: (1) $RMQ(i, j)$ gives the position of the minimum in $LCP[i, j]$; (2) $PSV(i)$ finds the last value smaller than $LCP[i]$ in $LCP[1, i - 1]$; and (3) $NSV(i)$ finds the first value smaller than $LCP[i]$ in $LCP[i + 1, n]$. All these could easily be solved in constant time using $O(n)$ extra bits of space on top of the LCP representation, but Fischer et al. give sublogarithmic-time algorithms to solve them with only $o(n)$ extra bits. Finally, the compressed suffix tree uses in total $nH_k(2\log(\frac{1}{H_k}) + \frac{1}{\epsilon} + O(1)) + o(n)$ bits, for any $k \leq \alpha \log_\sigma n$, where $0 < \alpha < 1$ is any constant.

In Section 6.1 we explain how they propose to solve RMQ , NSV and PSV over the LCP array and in Section 6.2 we give a novel idea to solve these operations.

As will be shown below all the operations listed in Table 2.2 for navigating the tree can be simulated using RMQ , NSV and PSV . Note that a node v is always associated with an interval in the suffix array, $[v_l, v_r]$. Given that Fischer et al.'s suffix tree nodes are based in suffix array intervals, many operations are implemented in the same way as in Russo et al.'s version. We will only describe the operations that are solved in a different way.

- **SDepth**(v): Return $LCP[k]$, where $k = RMQ(v_l + 1, v_r)$.
- **Parent**(v): If v is the root, return $NULL$. Otherwise, since the suffix tree is compact, we must have that the string-depth of $Parent(v)$ is either $LCP[v_l]$ or $LCP[v_r + 1]$, whichever is greater [RNO08b]. So, we set k to v_l if $LCP[v_l] > LCP[v_r + 1]$, else k is set to $v_r + 1$. The parent interval of v is then $[PSV(k), NSV(k) - 1]$.
- **FChild**(v): If v is a leaf, return $NULL$. Otherwise, the first child of v is given by $[v_l, RMQ(v_l + 1, v_r) - 1]$, assuming that RMQ s always returns the leftmost minimum in the case of ties.
- **NSibling**(v): First move to the parent of v by $w = Parent(v)$. If $v_r = w_r$, return $NULL$, since v does not have a next sibling. If $v_r + 1 = w_r$, v 's next sibling is a leaf, so return $[w_r, w_r]$. Otherwise, return $[v_r + 1, RMQ(v_r + 2, w_r) - 1]$.

- **SLink**(v): If v is the root, return *NULL*. Otherwise, first follow the suffix links of the leaves v_l and v_r , $x = \Psi(v_l)$ and $y = \Psi(v_r)$. Then locate $k = \text{RMQ}(x + 1, y)$. The final result is then given by $[\text{PSV}(k), \text{NSV}(k) - 1]$.
- **SLink** ^{i} (v): Same as above with $x = \Psi^i(v_l)$ and $y = \Psi^i(v_r)$. If the first *Letter* of x and y are different, then the answer is *Root*. Otherwise we go on with k as before.
- **LCA**(v, w): If one of v or w is ancestor of the other, return this ancestor node. Otherwise, without loss of generality, assume $v_r < w_l$. Compute $k = \text{RMQ}(v_r + 1, w_l)$, and the final answer is $[\text{PSV}(k), \text{NSV}(k) - 1]$.
- **Child**(v, a): If v is a leaf, return *NULL*. Otherwise, the minima in $\text{LCP}[v_l + 1, v_r]$ define v 's child intervals, so we need to find the position $p \in [v_l + 1, v_r]$ where $\text{LCP}[p] = \min_{i \in [v_l + 1, v_r]} \text{LCP}[i]$, and $T[A[p] + \text{LCP}[p]] = \text{Letter}([p, p], \text{LCP}[p] + 1) = a$. Then the final result is given by $[p, \text{RMQ}(p + 1, v_r) - 1]$, or *NULL* if there is no such position p . To find this p , they binary search the interval using *RMQ*.
- **TDepth**(v): They present a way to support $\text{TDepth}(v)$ using other $nH_k(2 \log \frac{1}{H_k} + O(1)) + o(n)$ bits of space. The idea is similar to Sadakane's representation of *LCP* (see Section 5.1): the key insight is that the tree depth can decrease by at most 1 if we move from suffix $T_{i,n}$ to $T_{i+1,n}$ (i.e., when following Ψ). Define $\text{TDE}[1, n]$ such that $\text{TDE}[i]$ holds the tree-depth of the *LCA* of leaves $A[i]$ and $A[i - 1]$ (similar to the definition of *LCP*). Then the sequence $k + \text{TDE}[\Psi^k(A^{-1}[i])]$, for $0 \leq k < n$, is nondecreasing and in the range $[1, n]$, and can hence be stored using $2n + o(n)$ bits. Further, the repetitions appear in the same way as in H (see Section 5.1), so the resulting sequence can be compressed to $nH_k(2 \log \frac{1}{H_k} + O(1)) + o(n)$ bits using the same mechanism as for *LCP* presented in Section 5.2. Then $\text{TDepth}(i) = \text{TDE}[k]$, where $k = \text{RMQ}_{\text{TDE}}(v_l + 1, v_r)$.
- **LAQ_S**(v, d): Let $u = [u_l, u_r] = \text{LAQ}_S(v, d)$ denote the (yet unknown) result. Because u is an ancestor of v , we must have $u_l \leq v_l$ and $v_r \leq u_r$. We further know that $\text{LCP}[i] \geq d$ for all $u_l < i \leq u_r$. Thus, u_l is the largest position in $[1, v_l]$ with $\text{LCP}[u_l] < d$. So the search for u_l can be conducted in a binary manner by means of *RMQ_S*: Letting $k = \text{RMQ}(\lfloor v_l/2 \rfloor, v_l)$, we check if $\text{LCP}[k] \geq d$. If so, u_l cannot be in $[\lfloor v_l/2 \rfloor, v_l]$, so we continue searching in $[1, \lfloor v_l/2 \rfloor - 1]$. If not, we know that u_l must be in $[\lfloor v_l/2 \rfloor, v_l]$, so we continue searching in there. The search for u_r is handled symmetrically.
- **LAQ_T**(v, d): The same idea as for *LAQ_S* can be applied here, using the array *TDE* instead of *LCP*, and *RMQ* on *TDE*.

The challenge faced in this thesis was to implement this CST. This can be divided into (1) how to represent *LCP* efficiently in practice, and (2) how to compute efficiently *RMQ*, *PSV*, and *NSV* over this *LCP* representation. In addition we made several improvements to the original algorithms. In the next chapters we focus on each subproblem separately and then compare the resulting CST with the existing ones.

Chapter 5

Representing the Array LCP

As explained the suffix array can be enhanced with the *LCP* array, which contains the lengths of the longest common prefixes of adjacent elements in the suffix array. The union of the *LCP* array with the suffix array is equivalent to the suffix tree. Conceptually the *LCP* array defines the “shape” of the suffix tree and thus allows any traversal to be simulated using the suffix array.

The study of the *LCP* array representations is very important for this thesis because it is one of the most important structures for our final compressed suffix tree. We carried out an exhaustive search of different alternatives, where we found all kinds of time/space trade-offs.

In the following sections we will explain the different alternatives that were considered to represent the *LCP* array. In the last section of the chapter we will compare the performance of the solutions. From this comparison we will be able to say which *LCP* representations are most suitable for our work.

5.1 Sadakane’s LCP (Sad)

Sadakane [Sad07a] describes a clever encoding of the *LCP* array that uses $2n + o(n)$ bits. The encoding is based on the following lemma, which states that *LCP* values, when listed in text position order, can decrease at most by 1.

Lemma 5.1. [Sad07a] *The values $j + LCP[A^{-1}[j]]$ are nondecreasing.*

Proof: By calling $i = A^{-1}[j]$, this is exactly the same as proving that $LCP[\Psi[i]] \geq LCP[i] - 1$. Let $p = A[i]$, $q = A[i + 1]$ and $l = LCP[i]$. If $LCP[i] = 0$ the inequality holds because $LCP[\Psi[i]] \geq 0$. If $T[p] = T[q]$, consider suffixes $T_{p+1,n}$ and $T_{q+1,n}$. From the definition of Ψ , $A[\Psi[i]] = p + 1$ and $A[\Psi[i + 1]] = q + 1$. The suffix $T_{q+1,n}$ is lexicographically larger than the suffix $T_{p+1,n}$ from the definition of lexicographic order. That is, $\Psi[i] < \Psi[i + 1]$. Therefore an integer i' such that $\Psi[i] + 1 = i' \leq \Psi[i + 1]$ exists. The suffix $T_{A[i'],n}$ has a prefix of length $l - 1$ that matches with

both prefixes of $T_{p+1,n}$ and $T_{q+1,n}$ because of the definition of lexicographic order. Therefore $LCP[\Psi[i]] \geq LCP[i] - 1$. \square

So sequence $S = s_1, \dots, s_{n-1} = 1 + LCP[A^{-1}[1]], 2 + LCP[A^{-1}[2]], \dots, n - 1 + LCP[A^{-1}[n - 1]]$ is nondecreasing.

To encode the nondecreasing list S , it is enough to encode each difference between consecutive values of the sequence in unary, i.e., $0^{diff(i)}1$, where $diff(i) = s_i - s_{i-1}$, 0^d denotes repetition of 0-bit d -times, and we assume that $s_0 = 0$. This encoding, call it H , takes at most $2n$ bits, because it has n 1's and at most n 0's, given that $\sum_{i=0}^{n-1} diff(i) \leq n$. We have the connection $diff(k) = select_1(H, k) - select_1(H, k - 1) - 1$. Bitvector H can be preprocessed to answer $select_1(H, k)$ queries in constant time using $o(|H|)$ bits of extra space [Mun96, Cla98] (see Section 2.3).

Computing $LCP[i]$ can now be done as follows: compute $k = A[i]$, and then $LCP[i] = select_1(H, k + 1) - 2k - 1$. Figure 5.1 shows an example of how H is computed and used, where LCP' is the LCP array in text order, that is, $LCP'[i] = LCP[A^{-1}[i]]$.

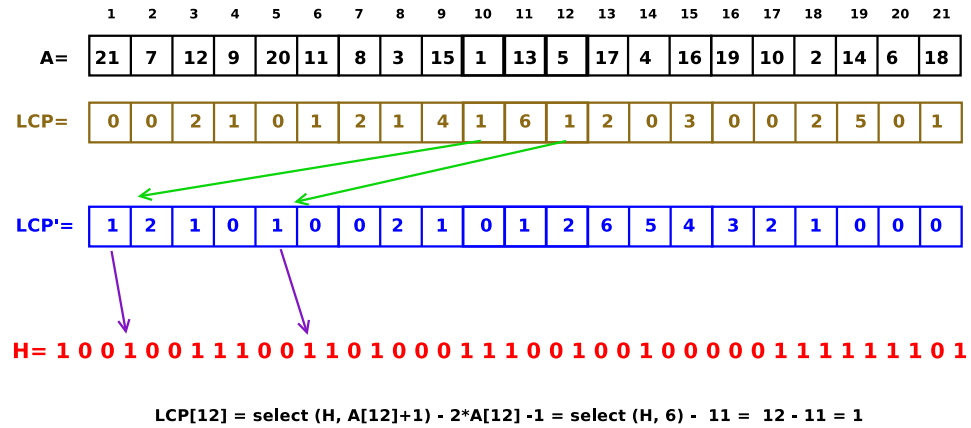


Figure 5.1: Example of how H is computed and used.

In this thesis we encode H in plain form, using the *rank/select* implementation of González [GGMN05], which takes $0.1n$ bits over the $2n$ used by H itself and answers $select$ in $O(\log n)$ time via binary search. This encoding will be called *Sad-Gon*. We also encode H using the *dense array* implementation of Okanohara and Sadakane [OS07]. This requires about the same space as the previous one and answers $select$ very fast in practice. This encoding will be called *Sad-OS*. Section 2.3 covers these implementations of $select$.

5.2 Fischer et al.'s LCP (FMN)

Fischer et al. [FMN09] suggested a more space-efficient encoding of H , obtaining $nH_k(2 \log \frac{1}{H_k} + O(1)) + o(n)$ bits, for small enough k . The result follows from the observation (Lemma 5.3) that the number of 1-bit runs in H is bounded by the number of runs in Ψ .

Definition 5.2. [MN05] A *run* in Ψ is a maximal sequence of consecutive i values where $\Psi(i) - \Psi(i-1) = 1$ and $T[A[i-1]] = T[A[i]]$, including one preceding i where this does not hold.

Note that an area in Ψ where the differences are not 1 corresponds to several length-1 runs. Let us define $R \leq n$ as the overall number of runs in Ψ .

Lemma 5.3. [FMN09] *The bitmap H has at most R runs of 1's (where even isolated 1's count as a run).*

Proof: Let us call position i a *stopper* if $i = 1$ or $\Psi(i) - \Psi(i-1) \neq 1$ or $T[A[i-1]] \neq T[A[i]]$. Hence Ψ has exactly R stoppers by the definition of runs in Ψ . Now say that a *chain* in Ψ is a maximal sequence $i, \Psi(i), \Psi(\Psi(i)), \dots$ such that each $\Psi^j(i)$ is not a stopper except the last one. As Ψ is a permutation with just one cycle, it follows that in the path of $\Psi^j(A^{-1}[1])$, $0 \leq j < n$, we will find the R stoppers, and hence there are also R chains in Ψ .

We now show that each chain in Ψ induces a run of 1's of the same length in H . Let $i, \Psi(i), \dots, \Psi^l(i)$ be a chain. Hence $\Psi^j(i) - \Psi^j(i-1) = 1$ for $0 \leq j < l$. Let $x = A[i-1]$ and $y = A[i]$. Then $A[\Psi^j(i-1)] = x+j$ and $A[\Psi^j(i)] = y+j$. Also, $LCP[i] = |lcp(T_{A[i-1],n}, T_{A[i],n})| = |lcp(T_{x,n}, T_{y,n})|$. Note that $T[x+LCP[i]] \neq T[y+LCP[i]]$, and hence $A^{-1}[y+LCP[i]] = \Psi^{LCP[i]}(i)$ is a stopper, thus $l \leq LCP[i]$. Moreover, $LCP[\Psi^j(i)] = |lcp(T_{x+j,n}, T_{y+j,n})| = LCP[i] - j \geq 0$ for $0 \leq j < l$. Now consider $s_{y+j} = y+j+LCP[A^{-1}[y+j]] = y+j+LCP[\Psi^j(i)] = y+j+LCP[i] - j = y+LCP[i]$, all equal for $0 \leq j < l$. This produces $l-1$ *diff* values equal to 0, that is, a run of l 1-bits in H . By traversing all the chains in the cycle of Ψ we sweep S left to right, producing at most R runs of 1's. \square

Lemma 5.4. [MN05] $R \leq nH_k + \sigma^k$, for any k .

Fisher et al. represent H in run-length encoded form, coding each maximal run of both 0 and 1 bits. Based on Lemma 5.4 they prove that the array H is compressible as it has at most R runs of 0's and 1's. The idea is to encode the 1-run lengths o_1, o_2, \dots and the 0-run lengths z_1, z_2, \dots separately. Given this encoding it is easy to compute $select_1(H, j)$ by finding the largest r such that $\sum_{i=1}^r o_i < j$ and then answering $select_1(H, j) = j + \sum_{i=1}^r z_i$. This so-called *searchable partial sum problem* is easy to solve. Store a bitmap $O[1, n]$ setting the bits at positions $\sum_{i=1}^r o_i$, so that $max\{r, \sum_{i=1}^r o_i < j\} = rank_1(O, j-1)$. Likewise, bitmap $Z[1, n]$ representing the z_i 's solves $\sum_{i=1}^r z_i = select_1(Z, r)$. Since both O and Z have at most R 1's, O plus Z can be represented using $2R \log \frac{n}{R} + O(R + \frac{n \log \log n}{\log n}) = nH_k(2 \log \frac{1}{H_k} + O(1)) + O(\frac{n \log \log n}{\log n})$ bits, for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$ [RRR02].

In this thesis this encoding for *LCP* will be called *FMN-RRR* and use Claude and Navarro's implementation of Ramman et al.'s *rank/select* representation [CN08] (recall *rrr* from Section 2.3). We also encode *O* and *Z* using the *sparse array* implementation of Okanohara and Sadakane for *rank/select* [OS07] (recall *sarray* from Section 2.3). This encoding will be called *FMN-OS*.

5.3 Puglisi and Turpin's LCP (PT)

This encoding is inspired in an *LCP* construction algorithm by Puglisi and Turpin [PT08]. Their strategy for saving space is to compute the *lcp* values for a sample of suffixes and then use those values to generate the rest. The particular sample that they use is defined by a *difference cover* with density selected such that the overall space requirements fit into the available memory.

Definition 5.5. [KSB06] A *difference cover* D modulo v is a set of integers in the range $0 \dots v - 1$ with the property that for all $i \in 0 \dots v - 1$ there exist $j, k \in D$ such that $i \equiv k - j \pmod{v}$.

A difference cover must have size $\geq \sqrt{v}$, and small ones (with size $\leq \sqrt{1.5v} + 6$) can be generated relatively easily with a method described by Colburn and Ling [CL00].

For any integers i, j , let the function $\delta(i, j)$ over a difference cover D modulo v be an integer $k \in 0..v - 1$ such that $(i + k) \pmod{v}$ and $(j + k) \pmod{v}$ are both in D [BK03]. To compute $\delta(i, j)$ they build a table d such that, for all $h \in [0, v)$, both $d[h]$ and $(d[h] + h) \pmod{v}$ are in D , and implement δ as $\delta(i, j) = (d[(j - i) \pmod{v}] - i) \pmod{v}$. Then $i + \delta(i, j) = d[(j - i) \pmod{v}] \in D$ and $j + \delta(i, j) = d[(j - i) \pmod{v}] + (j - i) \in D \pmod{v}$.

Their algorithm to create the *LCP* array comprises the following steps:

- Choose $v \geq 2$, a difference cover D modulo v with $|D| = \Theta(\sqrt{v})$, and compute the function δ .
- Fill an array $S[0, n|D|/v - 1]$ with the suffixes whose starting position modulo v is in D . This can be easily done if we first preprocess D and build a small table $D'[0..v - 1]$, such that $D'[i]$, if $i \in D$, is the number of elements smaller than i in D , otherwise $D'[i] = -1$. After filling the array S we build a data structure over S that can compute the *lcp* of an arbitrary pair of sample suffixes in constant time. The first part of this structure is a mapping S' that gives us the position in S of a given sampled suffix in constant time. Figure 5.2 shows how to fill arrays S and S' using the *suffix array*, D and D' . Finally using D' , S' and S we compute an array $L[i] = \text{lcp}(S[i - 1], S'[i])$, where $i \in [0, n|D|/v - 1]$. Figure 5.3 shows the pseudocode to compute the values of L .
- Scan the suffix array A left-to-right and compute all the *LCP* values. To compute $LCP[i]$ compare the first v symbols of $A[i]$ and $A[i - 1]$. If we find a mismatch at $A[i] + j$ then we know that the *lcp* is $j - 1$ and we are done. If they are equal after v symbols then compute the *lcp* using the fact that suffixes $A[i] + \delta(A[i - 1], A[i])$ and $A[i - 1] + \delta(A[i - 1], A[i])$

Algorithm *Fill-Arrays*(A, D, D')

1. $j \leftarrow 0$;
 2. **for** $i \leftarrow 0$ **to** n **do**
 3. **if** $D'[A[i] \bmod v] \neq -1$ **then**
 4. $S[j] \leftarrow A[i]$;
 5. $S'[\lfloor D \lfloor S[j]/v \rfloor + D'[S[j] \bmod v]] \leftarrow j$;
 6. $j \leftarrow j + 1$;
-

Figure 5.2: Computing arrays S and S' .

Algorithm *Compute-L*(S, S', D')

1. **for** $s \leftarrow 0$ **to** $|D| - 1$ **do**
 2. $k \leftarrow s$;
 3. $l \leftarrow 0$;
 4. **while** $k < m$ **do**
 5. $s_0 \leftarrow S[S'[k] - 1]$;
 6. $s_1 \leftarrow S[S'[k]]$;
 7. **while** $T[s_0 + l] = T[s_1 + l]$ **do**
 8. $l \leftarrow l + 1$;
 9. $L[S'[k]] \leftarrow l$;
 10. $l \leftarrow \max(0, l - v)$;
 11. $k \leftarrow k + |D|$;
-

Figure 5.3: Computing L , using S , S' , and D' .

are in the sample and their lcp can be computed in constant time using the data structure alluded to in the previous step.

Note that the original implementation is for creating the LCP array keeping the text T available. In the encoding presented here we store the particular sampling of LCP values, and compute the other values of the array using the sampled ones. So given a parameter v , the sampling requires $O(n/\sqrt{v} + v)$ integers plus the text, and computes any $LCP[i]$ by comparing at most the text symbols $T[j, j+v]$ and $T[j', j'+v]$. As we do not have the text, we must obtain these symbols using Ψ and the CSA up to $2v$ times, so the time to access a value will depend on the time of operation Ψ and the time used by the CSA to obtain a symbol of the text.

5.4 Kärkkäinen et al.'s LCP (PhiSpare)

In the same spirit of the previous technique, this is inspired by a construction from Kärkkäinen et al. [KMP09]. For a given parameter q , they store in text order an array LCP'_q with the LCP values for all text positions $q \cdot k$. Now assume $A[i] = q \cdot k + b$, with $0 \leq b < q$. If $b = 0$, then $LCP[i] = LCP'_q[k]$. Otherwise, we first calculate $h = q \cdot k + LCP'_q[k] - A[i]$. If $h > 0$, then the substring $T_{A[i], A[i]+h}$ is repeated at least once in the text, so it holds that $LCP[i] \geq h$ and we only need to compare $T_{A[i-1]+h, n}$ and $T_{A[i]+h, n}$. Note that this comparison is done by brute force, but cannot cost more than $(q-h) + LCP'_q[k+1]$, because otherwise the substring $T_{q(k+1)+LCP'_q[k+1]+z}$, being $z > 0$, would be repeated at least once in the text, and therefore $LCP'_q[k+1]$ should be greater, and that is not possible. If $h \leq 0$ we just calculate $LCP[i]$ by brute force. Figure 5.4 shows the different cases.

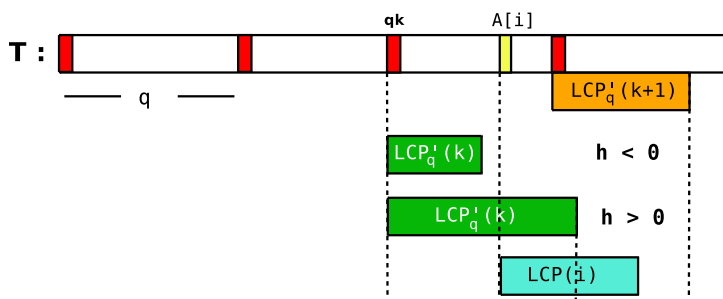


Figure 5.4: Scheme of PhiSpare-LCP.

So $LCP[i]$ is computed by comparing at most $q + LCP'_q[k+1] - LCP'_q[k]$ symbols of the suffixes $T_{A[i-1], n}$ and $T_{A[i], n}$, and the space used is n/q integers. If $A[i] = q \cdot k$ then computing it takes $O(1)$ time, else it takes at most $q + LCP'_q[k+1] - LCP'_q[k]$. Let $c(i)$ denote the number of comparisons needed for computing $LCP[i]$. Kärkkäinen et al. prove that $\frac{1}{n} \sum_{i=1}^n n \cdot c(i) \leq (q-1) + \frac{q^2}{n}$ [KMP09], then the computation of $LCP[i]$ requires $O(q)$ time on average. Just as in Section 5.3, when we use LCP the text will not be available, so we must obtain the symbols of each position of the text using Ψ and the CSA .

Note that in the extreme case $q = n$, the LCP'_q array is not used, so all the LCP positions are calculated using only the CSA and Ψ , and the encoding presented here does not use any space.

5.5 Directly Addressable Codes (DAC)

Brisaboa et al. [BLN09] introduce a symbol reordering technique called directly addressable variable-length codes (DAC), that implicitly synchronizes variable-length codes, such that it is possible to directly access the i -th codeword without need of any sampling method. The idea

builds on *Vbyte coding*.

Definition 5.6. *Vbyte coding* [WZ99] splits the $\lfloor \log(p_i + 1) \rfloor$ bits needed to represent a value p_i into blocks of b bits and stores each block in a chunk of $b + 1$ bits. The highest bit is 0 in the chunk holding the most significant bits of p_i , and 1 in the rest of the chunks. For clarity we write the chunks from most to least significant, just like the binary representation of p_i . For example, if $p_i = 25 = 11001_2$ and $b = 3$, then we need two chunks and the representation is 0011 1001.

Brisaboa et al. first encode the p_i s as a sequence of $(b + 1)$ -bit chunks. Next they separate the different chunks of each codeword. Assume p_i is assigned a codeword C_i that needs r chunks $C_{i,r}, \dots, C_{i,2}, C_{i,1}$. A first stream, V_1 , will contain the $n_1 = n$ least significant chunks (i.e., right-most) of every codeword. A second one, V_2 , will contain the $n_2 \leq n_1$ second chunks of every codeword (so that there are only n_2 codewords using more than one chunk). We proceed similarly with V_3 , and so on. Each stream C_k will be separated into two parts. The lowest b bits of the chunks will be stored contiguously in an array A_k (of $b \cdot n_k$ bits), whereas the highest bits will be concatenated into a bitmap B_k of n_k bits. The bits in each B_k identify whether there is a chunk of that codeword in V_{k+1} . They set up *rank* data structures on the B_k bitmaps (Section 2.3).

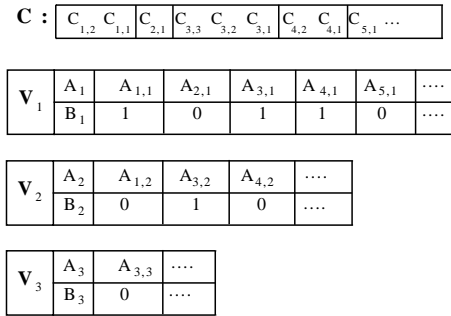


Figure 5.5: Example of reorganization of the chunks of each codeword.

Extraction of the i -th value of the sequence is carried out as follows. It starts with $i_1 = i$ and gets its first chunk $b_1 = B_1[i_1] : A_1[i_1]$. If $B_1[i_1] = 0$ we are done with $p_i = A_1[i_1]$. Otherwise it sets $i_2 = \text{rank}_1(B_1, i_1)$, which sends us to the correct position of the second chunk of p_i in B_2 , and gets $b_2 = B_2[i_2] : A_2[i_2]$. If $B_2[i_2] = 0$, we are done with $p_i = A_1[i_1] + A_2[i_2] \cdot 2^b$. Otherwise it sets $i_3 = \text{rank}_1(B_2, i_2)$ and so on.

Most *LCP* values are small ($O(\log_\sigma n)$ on average [Szp96]), and thus one could use few bits to represent them. Yet, some can be much longer. Then it seems promising to apply the *DAC* representation to the *LCP* array. There are two variants of this structure, both implemented by Ladra: one with fixed b (*DAC*), and another using different b values for the first, second, etc. blocks, and finding the values of b that minimize the total space (*DAC-Var*). Note that we represent *LCP* and not *LCP'*, thus we do not need to compute $A[i]$.

It is important to mention that these implementations have been improved in this thesis for supporting sequential accesses to the *LCP* array. The improvement is that for all the sequential

accesses we will only need to do one *rank* operation for each level visited. After this operation we set a pointer per level that avoids further *rank* computations.

5.6 Re-Pair LCP (RP)

Re-Pair [LM00] is a grammar-based compression method that factors out repetitions in a sequence. This method is based on the following heuristic: (1) Find the most repeated pair ab in the sequence; (2) Replace all its occurrences by a new symbol s ; (3) Add a rule $s \rightarrow ab$ to a dictionary; (4) Iterate until every pair is unique.

In recent work [GN07] *Re-Pair* is used to compress the differentially encoded suffix array, $A'[i] = A[i] - A[i - 1]$. It is shown that *Re-Pair* achieves $O(R \log \frac{n}{R} \log n)$ bits of space, R being the number of runs in Ψ . This differentially encoded suffix array contains repetitions because the *CSA* can be partitioned into R areas that appear elsewhere in A with the values shifted by 1 [MN05] (recall Section 5.2).

The *LCP* must contain the same repetitions shifted by 1, and therefore the *Re-Pair* compression of the differential *LCP* should perform similarly, as advocated in the theoretical proposal [FMN09]. To obtain $LCP[i]$ we must store some sampled absolute *LCP* values and decompress the nonterminals since the last sample.

5.7 Experimental Comparison

All the experimental results that will be shown here and in the next chapters were performed on 100 MB of the protein, sources, XML and DNA texts from the Pizza&Chili corpus (<http://pizzachili.dcc.uchile.cl>). Table 5.1 gives some information about the texts used for the tests. The computer used features an Intel(R) Core(TM)2 Duo processor at 3.16 GHz, with 8 GB of main memory and 6 MB of cache, running version 2.6.24-24 Linux kernel.

Name	Mean LCP	Max LCP	Size (bytes)	Description
dna.100MB	28.0958	17,772	104,857,600	DNA sequences obtained from Gutenberg Project
xml.100MB	44.4497	1,084	104,857,600	XML obtained from dblp.uni-trier.de
proteins.100MB	220.649	35,246	104,857,600	Protein sequences obtained from the Swissprot database
sources.100MB	625.065	307,871	104,857,600	Source code obtained by concatenating files of the linux-2.6.11.6 and gcc-4.0.0 distributions

Table 5.1: Data files used for tests, sorted in ascending order of average LCP.

It is important to mention that the *PT* and *PhiSpare* encodings are the only ones that will display a space/time trade-off because their implementations are the only that depend of a parameter given by the user, v and q respectively. In the first we use $v = 4, 6, 8$ and for the second

$q = 16, 32, 64, n$ (when $q = n$, it means that we calculate each LCP value using only the CSA and Ψ). We also consider a naive implementation for LCP that uses $\log(\max_lcp)$ bits per value of the array. For all the LCP representations we will use Sadakane’s compressed suffix array, when it is necessary. Finally for $DAC/DAC\text{-}Var$ we use González’s implementation for the bitmaps [GGMN05], and for DAC we use $b = 4$.

To compare all the LCP structures presented we carry out two different tests. The first test measures the average time taken to access a random position in the LCP array. For that we tested the different LCP implementations by accessing 100,000 random positions of the LCP . Figure 5.6 shows the space/times achieved.

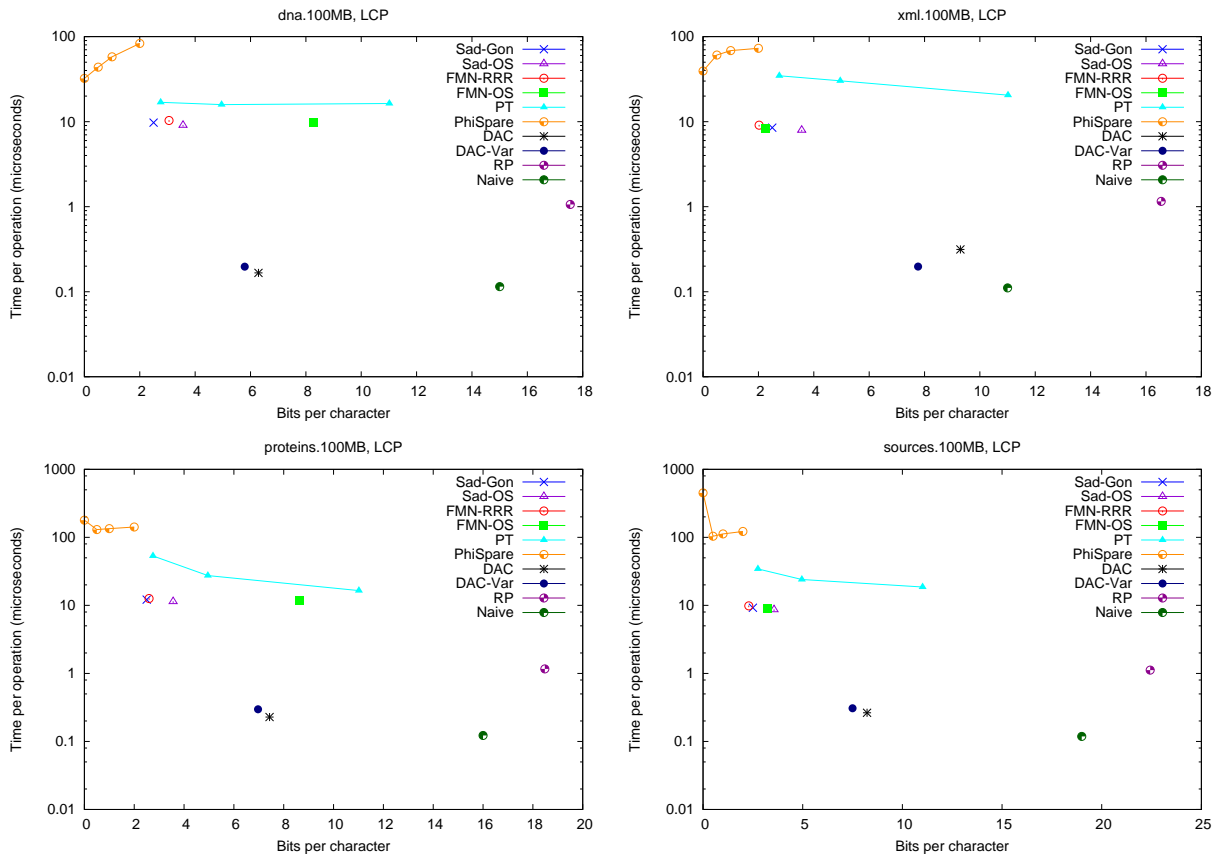


Figure 5.6: Space/time for randomly accessing the LCP array. Note the logscale.

As it can be seen, $DAC/DAC\text{-}Var$ and the representations of the bitvector H (see Section 5.1), Sad and FMN implementations, largely dominate the space-time trade-off map. $PhiSpare$ and PT can use less space but they become impractically slow. Note that $PhiSpare$ encoding is a bad option unless the average LCP in the text is high. This is because, to compute $LCP[i]$ in our

implementation of *PhiSpare*, if $h > 0$ (recall Section 5.4) we must move h positions in the text and obtain the suffix array position with $A^{-1}[A[i] + h]$ and $A^{-1}[A[i - 1] + h]$, and then continue by brute force. The problem is that if h is too small it will be faster to move the h positions by brute force, i.e., using $\Psi^h(A[i])$, than to do it via A and A^{-1} . So for a text with low average *LCP*, it would be desirable to use a higher q so that most *LCP* values are computed by brute force. This explains why *PhiSpare* sometimes worsens in time as it uses more space.

Note that *DAC*, *DAC-Var*, *RP* and the *naive* implementation are faster than the other structures because they not need to access the text or compute $A[i]$. Note also that *RP* takes 30%-60% of the original *LCP* array. The reason why the *naive* implementation uses less space than *RP* is that it uses $\log(\max_lcp)$ bits per entry, while *RP* creates many new symbols, and thus it yields a shorter sequence whose symbols need more bits. Still, the naive representation, while slightly faster than *DAC*, uses too much space in comparison.

The second test is based in how we use the *LCP* array to answer queries like *NSV*, *PSV*, and *RMQ*. This test consists in accessing 100,000 random position but for each such position we also access the next 32 positions in a sequential way. The results obtained in this test are important because most of the times we use *LCP* we will need to do several sequential accesses.

As it can be seen in Figure 5.7, all the *LCP* representations run a little faster on sequential accesses, but only the *naive* and *DAC/DAC-Var* implementations show a marked improvement. For the *naive* implementations this is because most of the entries will be in cache and, in the case of *DAC/DAC-Var* implementation, it is because these representations, as we mentioned in Section 5.5, were improved in this thesis for doing sequential accesses to the *LCP* array.

We can also conclude that the idea of compressing Sadakane's H array (recall Section 5.1) makes little difference. This is attributable to the 27% overhead over the entropy of the *RRR* implementation for *FMN-RRR*, and similarly *FMN-OS* spends $2m$ extra bits [OS07] (recall *rrr* and *darray* structures from Section 2.3).

For the rest of this work we will keep only *DAC* and *DAC-Var*, which give the best time performance within reasonable space, and *FMN-RRR* and *Sad-Gon*, which achieve minimum space and a robust and reasonable performance.

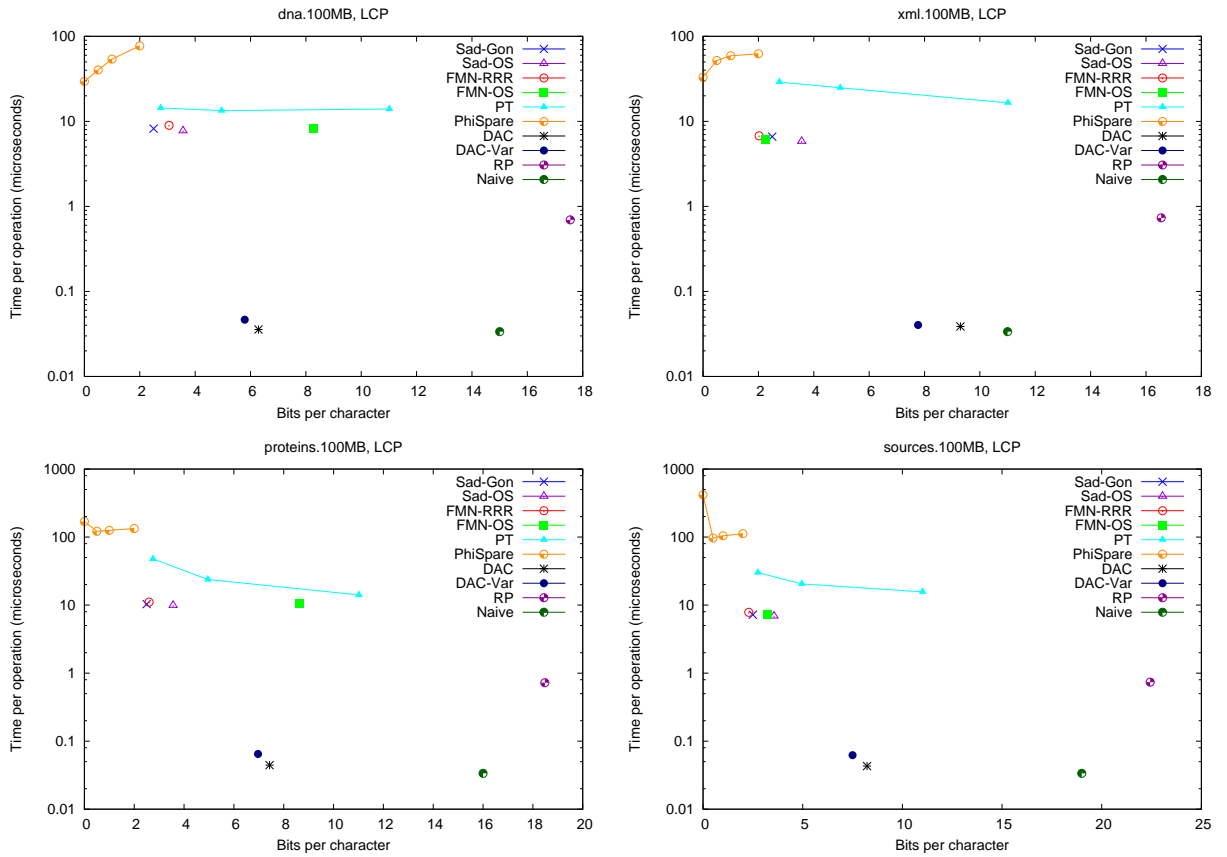


Figure 5.7: Space/time for random sequential accesses to the LCP array.

Chapter 6

Computing NSV/PSV/RMQ (NPR)

In this chapter we study the queries *next smaller value (NSV)*, *previous smaller value (PSV)* and *range minimum query (RMQ)*.

Definition 6.1. Let $S[1, n]$ be a sequence of elements drawn from a set with a total order \preceq (where one can also define $a \prec b \Leftrightarrow a \preceq b \wedge b \not\preceq a$). We define the queries next smaller value, previous smaller value, and range minimum query as follows: $NSV_S(i) = \min\{j, (i < j \leq n \wedge S[j] \prec S[i]) \vee j = n + 1\}$, $PSV_S(i) = \max\{j, (1 \leq j < i \wedge S[j] \prec S[i]) \vee j = 0\}$, and $RMQ_S(i, j) = \operatorname{argmin}_{i \leq l \leq j} S[l]$ (where *argmin* refers to order \preceq), respectively.

Once a representation for *LCP* is chosen, one must carry out these operations on top of it (as they require to access *LCP*). As we already showed in Section 4.3, we make heavy use of these queries to solve most of the navigation operations in our new compressed suffix tree.

6.1 Fischer et al.'s NPR

We first implemented verbatim the theoretical proposals of Fischer et al. [FMN09]. For *NSV*, the idea is akin to the recursive *findclose* solution for compressed trees (see Section 3.2.2). The array is divided into blocks of b values. A position i will be called *near* if $NSV(i)$ is within the same block of i . So the first step when solving *NSV* query will be to scan the values $S[i + 1 \dots b \cdot \lceil i/b \rceil]$, that is, from $i + 1$ to the end of the block, looking for an $S[j] \prec S[i]$. This takes $O(b)$ time and solves the query for *near* positions.

Positions that are not *near* are called *far*. A *far* position i will be called a *pioneer* if $NSV(i)$ is not in the same block of $NSV(j)$, being j the largest *far* position preceding i (the first *far* position in the array is a *pioneer*). It follows that, if i is not a *pioneer* and j is the last *pioneer* preceding i , then $NSV(i)$ is in the same block of $NSV(j) \geq NSV(i)$. Hence, to solve *NSV*(i), we find j and then scan (left to right) the block $S[\lceil NSV(j)/b \rceil - b + 1 \dots NSV(j)]$ for the first

position p where $S[p] \prec S[i]$. This is done in $O(b)$ time.

The remaining problem is how to find efficiently the *pioneer* preceding each *far* position i , and how to store the answers for *pioneers*. The form to solve this is to mark all the *pioneers* in a bitmap $P[1, n]$. Using a structure for *rank/select* (Section 3.2) on P we can find the last pioneer preceding a *far* position i , as $j = \text{select}_1(P, \text{rank}_1(P, i))$. So if we store all the *NSV* answers for the *pioneers* in an array $N[1, n']$, where $n' = O(n/b)$ [Jac89] and $NSV(j) = N[\text{rank}_1(P, j)]$ if j is a *pioneer*, then the time taken to answer *NSV* for any position is $O(b)$, but the space used for storing the *pioneer* answers is $O((n/b) \log n)$.

Instead, they apply the same idea recursively. Instead of storing the answers explicitly in array N , they form a (virtual) *reduced* sequence S' that contains all the *pioneer* values i and their answers $NSV(i)$. Sequence S' is not explicitly stored. Rather, they set up a bitmap $R[1, n]$ where the selected values of S are marked, so they can retrieve any value $S'[i] = S[\text{select}_1(R, i)]$. Because S' is a subsequence of S , it holds that the answers to *NSV* in S' are the same answers mapped from S . Hence $NSV(i)$, for *pioneer* i , can be found by the corresponding recursive query on S' , $NSV(i) = \text{select}_1(R, NSV_{S'}(\text{rank}_1(R, i)))$. They use recursion to solve the *NSV* problem over the new sequence S' . They continue the process recursively for r levels before storing the explicit answer in array $N[1, n_r]$, where $n_r = O(n/b^r)$. Therefore the data structure uses $O(\frac{n \log b}{b} + r \frac{n \log \log n}{\log n} + \frac{n \log n}{b^r})$ bits, where $O(\frac{n \log n}{b^r})$ bits are used to store N , and the rest is the sum of the space used by all the R^i and P^i ($i = 1, \dots, r$), and the time taken to answer *NSV* is in the worst case $O(r \cdot b)$, taking $O(b)$ time per level. In this thesis we only use two levels of recursion because, given the size of the texts we used in the experiments, the use of more levels would mean an increase in the size and the time used by the representation.

The block length b yields a space/time trade-off since, at each level of the recursion, we must obtain $O(b)$ values from *LCP*. *PSV* is symmetric, so another similar structure is needed.

For *RMQ* we apply an existing implementation [FH07] to the *LCP* array, remembering that we do not have direct access to *LCP* but have to use any of the access methods we have developed for it in Chapter 5. This accesses at most 5 cells of *LCP*, yet it requires $3.25n$ bits. In the actual theoretical proposal [FMN09] this is reduced to $o(n)$ but many more accesses to *LCP* would be necessary; we did not implement that verbatim as it has little chances of being practical.

The final data structure, that we call *NPR-FMN*, is composed of the structure to answer *NSV* queries plus the one for *PSV* queries plus the structure to calculate *RMQ*.

6.2 A Novel Practical Solution for NPR

We propose a different solution, inspired by Sadakane and Navarro's succinct tree representation (see Section 3.2.3). We divide *LCP* into blocks of length L , and store the minimum *LCP* value of each block i in an array $m[i]$. The array uses $\frac{n}{L} \log n$ bits. Now we form a hierarchy of blocks. On top of array m , we construct a perfect L -ary tree T_m where the leaves are the elements

of m and each internal node stores the minimum of the values stored in its children. The total space needed for T_m is $\frac{n}{L} \log n(1 + O(1/L))$ bits, so if $L = \omega(\log n)$, the space used is $o(n)$ bits.

To answer $NSV(i)$, we look for the first $j > i$ such that $LCP[j] < p = LCP[i]$, using T_m to find it in time $O(L \log(n/L))$. We first search sequentially for the answer in the same block of i . If it is not there, we go up to the parent of the leaf that represents the block and scan the minima of the right siblings of this leaf. If some of these sibling leaves contain a minimum value smaller than p , then the answer to $NSV(i)$ is within their block, so we go down to their block and find sequentially the leftmost position j where $LCP[j] < p$. If, however, no sibling of the leaf contains a minimum smaller than p , we continue going up the tree and considering the right siblings of the parent of the current node. At some node we find a minimum smaller than p and start traversing down the tree as before, finding at each level the first child of the current node with a minimum smaller than p . PSV is symmetric. Without the need to use more space we can easily define $NSV'(i, d)/PSV'(i, d)$, a generalization of NSV/PSV , which find the next/previous value smaller or equal to d , and its implementation follow the same algorithm presented for NSV/PSV . Note that the heaviest part of the cost in practice is the $O(L)$ accesses to LCP cells at the lowest levels, since the minima in T_m are explicitly stored. Therefore it is important to have a fast sequential access to the LCP .

To calculate $RMQ(x, y)$ we use the same T_m and separate the search into three parts: (a) We calculate sequentially the minimum value in the interval $[x, L\lceil \frac{x}{L} \rceil - 1]$ and its leftmost position in the interval; (b) we do the same for the interval $[L\lfloor \frac{y}{L} \rfloor, y]$; (c) we calculate $RMQ(L\lceil \frac{x}{L} \rceil, L\lfloor \frac{y}{L} \rfloor - 1)$ using T_m . Finally we compare the results obtained in (a), (b) and (c) and the answer will be the one holding the minimum value, choosing the leftmost to break ties. The remaining problem is how we compute (c). This is done by searching for the minimum value between the leaves that are among the $\lceil \frac{x}{L} \rceil$ -th leaf and $(\lfloor \frac{y}{L} \rfloor - 1)$ -th leaf of T_m . Then we go down to the block that represents the minimum value found, and find sequentially its leftmost position. To compute the range minimum query between leaves of T_m we recursively separate the search into three parts: (a') We find sequentially the minimum value and its position between the $\lceil \frac{x}{L} \rceil$ -th leaf and its right siblings; (b') we do the same between the $(\lfloor \frac{y}{L} \rfloor - 1)$ -th leaf and its left siblings; (c') we calculate the minimum value and its position between the leaves $[L\lceil \frac{x}{L} \rceil, L\lfloor \frac{y}{L} \rfloor - 1]$. Finally we compare the results obtained in (a'), (b') and (c') and the answer will be the one holding the minimum value, choosing the leftmost to break ties. To compute (c') we apply recursively the same idea over the next level of T_m .

For each node in T_m we will also store the local position in the children where the minimum occurs, so we do not need to scan the child blocks when we want to know which of these is the leftmost child that contains the minimum value. The extra space incurred is just $\frac{n}{L} \log L(1 + O(1/L))$ bits. The final data structure, if $L = \omega(\log n)$, requires $o(n)$ bits and can compute NSV , PSV and RMQ all using the same auxiliary structure. We call it $NPR-CN$.

6.3 Experimental Comparison

We tested the performance of the different *NPR* implementations by performing 100,000 *NSV* and *RMQ* queries at different random positions in the *LCP* array. We will show the space/time achieved for each implementation, using *Sad-Gon*, *FMN-RRR*, *DAC* and *DAC-Var* *LCP* encodings, over the texts of Table 5.1. We obtained space/time trade-offs by using different block sizes $L = 8, 16, 32$. Note that the times and space used for *RMQ* on *NPR-FMN* are not affected by L . It is important to mention that the spaces shown are those used by the structure (in the case of *CN*), or the sum of the structures (in the case of *FMN*), that support *NSV*, *PSV*, and *RMQ* without considering the space used by the chosen *LCP* representation.

Figures 6.1 and 6.2 show the results obtained with *NPR-FMN* and *NPR-CN* using *Sad-Gon* and *FMN-RRR* encodings of *LCP*. Clearly *NPR-CN* displays the best performance for *NSV* (therefore also for *PSV*), both in space and time. For *RMQ*, one can see that the best time obtained with *NPR-CN* dominates, in time and space, the *NPR-FMN* curve. Note that using *Sad-Gon* encoding of *LCP*, for most of the texts, the times are slightly better than those using *FMN-RRR*. The spaces obtained by *NPR-FMN* plus *Sad-Gon* or *FMN-RRR* encodings of *LCP* are $9.5n$ to $13.5n$ or $9n$ to $14n$ bits, respectively, depending on L and the text, while *NPR-CN* uses $3.7n$ to $7.6n$ or $3.2n$ to $8.1n$ bits, respectively, and only depend on L .

Figures 6.3 and 6.4 show the results obtained with *NPR-FMN* and *NPR-CN* using *DAC* and *DAC-Var* encodings of *LCP*. As expected the times obtained using *DAC* encoding, for most of the texts, are better than the ones obtained using *DAC-Var*. Again *NPR-CN* displays the best performance for *NSV* (therefore also for *PSV*), both in space and time. For *RMQ*, one can see that the results obtained with *NPR-FMN* always dominates, in time, the *NPR-CN* curve. The problem is that the extra space used by *NPR-FMN* in addition to *DAC* or *DAC-Var* implementations of *LCP*, which is between $13.3n$ to $20.3n$ bits using *DAC* and $12.8n$ to $18.8n$ bits using *DAC-Var*, is too high to be considered a good alternative. On the other hand, if we use *NPR-CN* we can afford to use *DAC* or *DAC-Var* maintaining a competitive time to answer *RMQ* queries while using reasonable space, between $7.5n$ to $14.4n$ with *DAC* and $7n$ to $12.9n$ bits with *DAC-Var*. Thus *NPR-CN* will be our preferred option to be used in our final structure no matter what encoding of *LCP* we choose. Another reason to choose *NPR-CN* is that it can support *NSV'/PSV'*, which will allow us to solve operation LAQ_T (see Chapter 7) without adding more memory as opposed to the technique presented in Section 4.3.

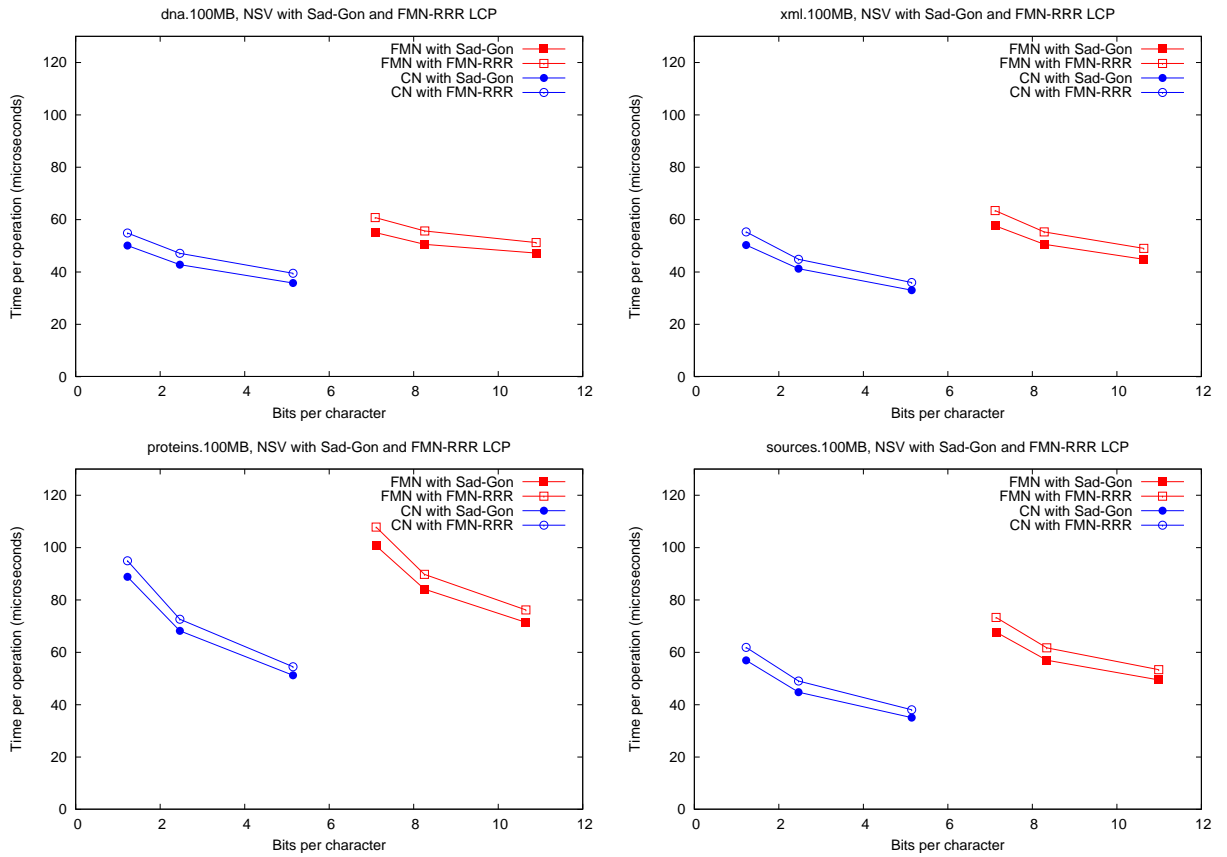


Figure 6.1: Space/time for the operation NSV using Sad-Gon and FMN-RRR.

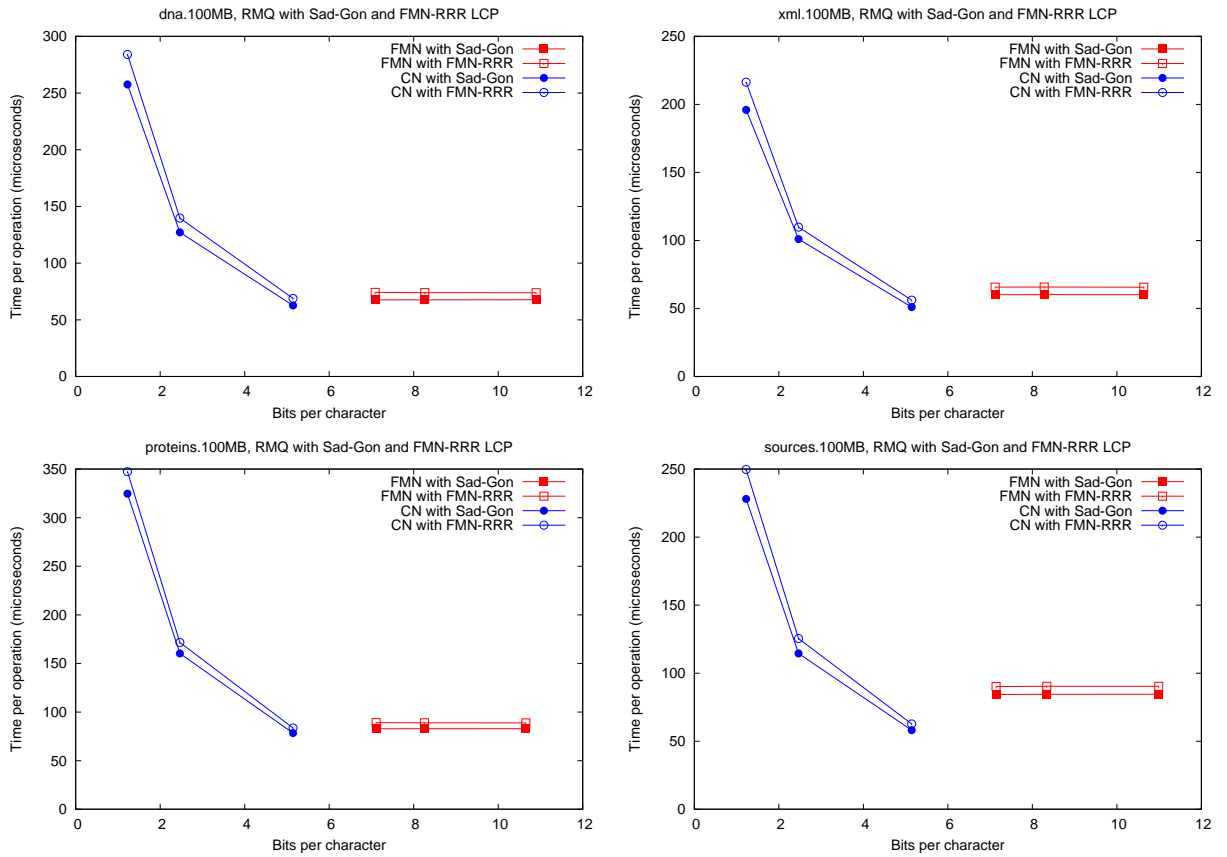


Figure 6.2: Space/time for the operation *RMQ* using *Sad-Gon* and *FMN-RRR*.

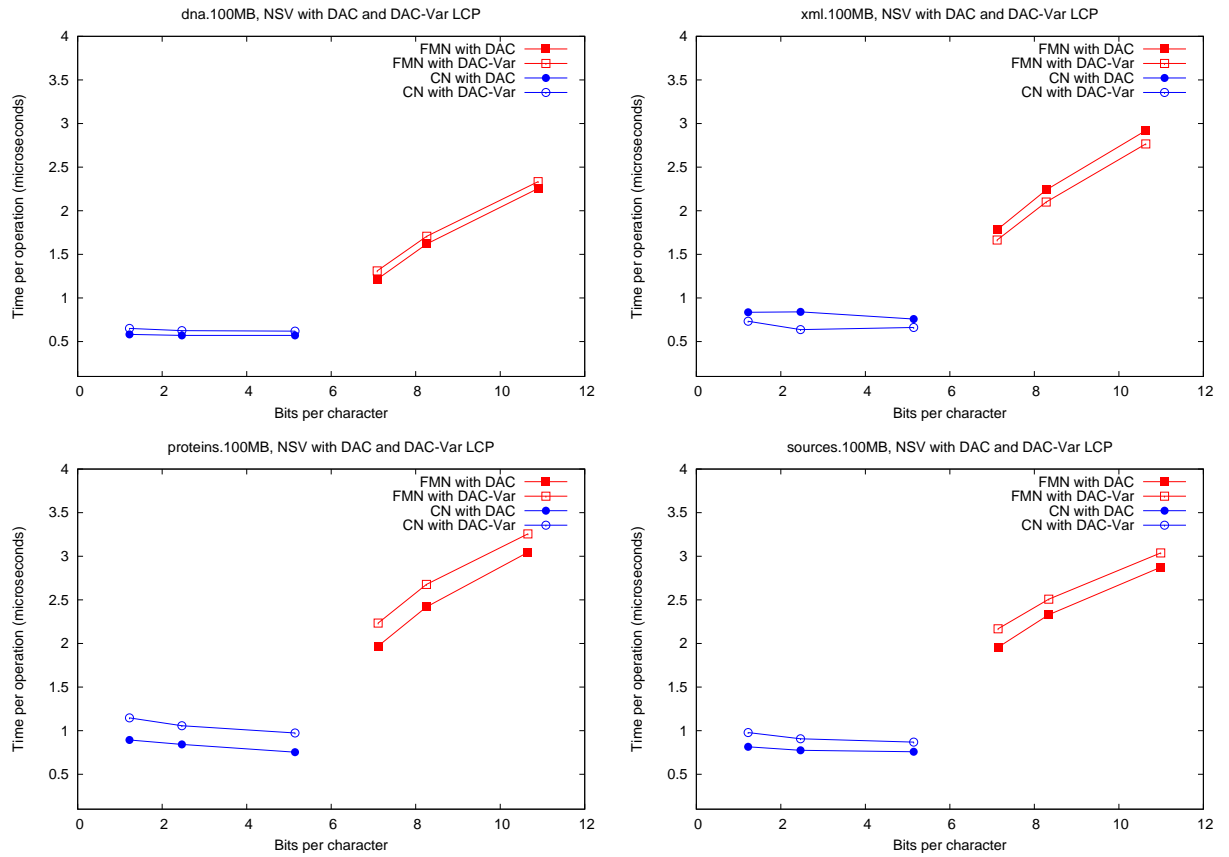


Figure 6.3: Space/time for the operation NSV using DAC and DAC-Var.

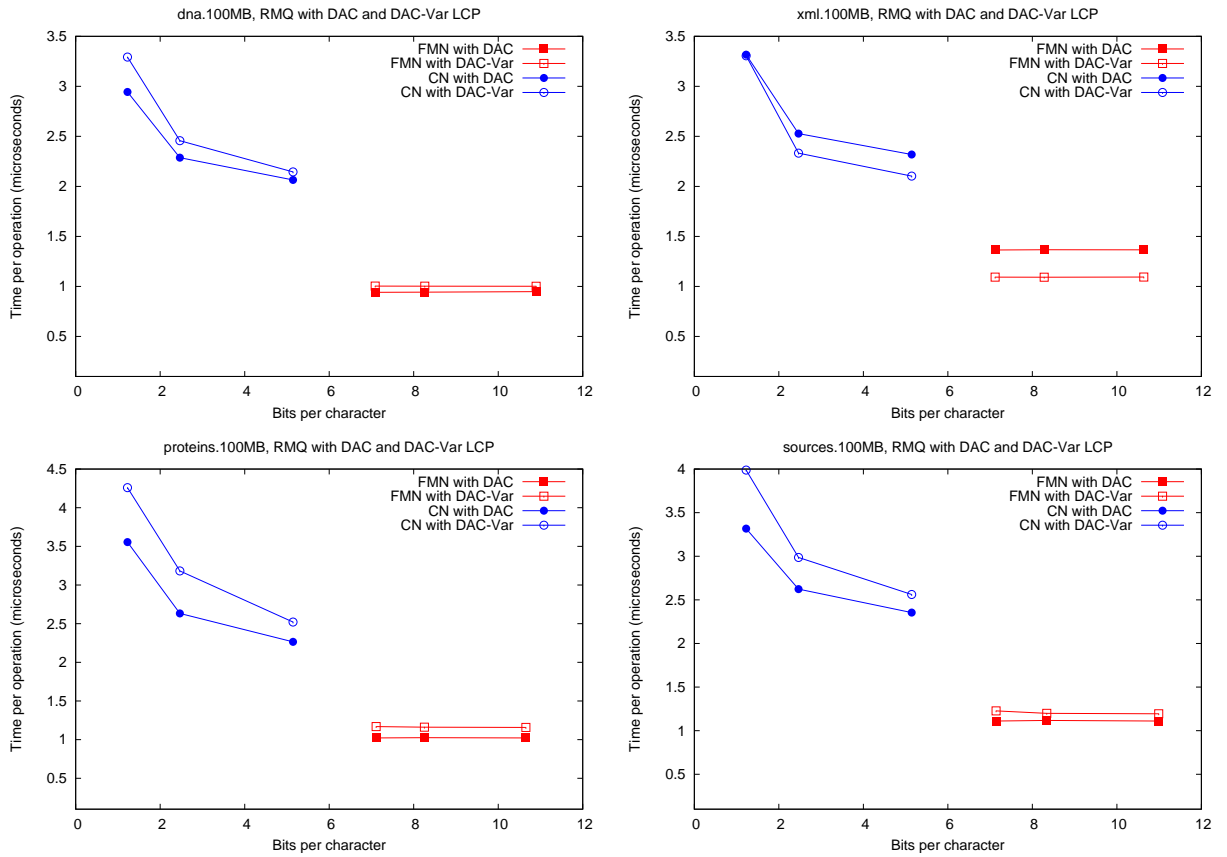


Figure 6.4: Space/time for the operation *RMQ* using *DAC* and *DAC-Var*.

Chapter 7

Our Compressed Suffix Tree

Our *CST* implementation applies our *NPR-CN* algorithms of Chapter 6 on top of some *LCP* representation from those chosen in Chapter 5. Using this we solve most of the tree traversal operations by using the formulas provided in Section 4.3. In some cases, however, we have deviated from the theoretical algorithms for practical considerations.

NSibling(v): First obtain the parent $[w_l, w_r]$ of v , then check whether $v_r = w_r$ (in which case there is no next sibling), then check whether $w_r = v_r + 1$ (in which case the next sibling is leaf $[w_r, w_r]$), and finally answer $[v_r + 1, z - 1]$, where $z = RMQ(v_r + 2, w_r)$. This *RMQ* is aimed at finding the end of the next sibling of the next sibling, but can fail if we are near the end. Instead, in this thesis we replace it by the faster $z = NSV'(v_r + 1, LCP[v_r + 1])$ (see Section 6.2).

SLink ^{i} (v): Return the value resulting of applying i times *SLink* recursively.

Child(v, a): The children are ordered by letter. We extract the children sequentially using *FChild* and *NSibling*, to find the one descending by the correct letter, yet extracting the *Letter* of each is expensive (see end of Section 4.2). Thus we first find all the children sequentially and then binary search the correct letter among them, thus reducing the use of *Letter* as much as possible.

TDepth(v): We move to the root using *Parent*, as there is no alternative practical solution in the proposal.

LAQ_S(v, d): Instead of the slow complex formula given in Section 4.3, we use *NSV'* (and *PSV'*): $LAQ_S([v_l, v_r], d) = [PSV'(v_l + 1, d), NSV'(v_r, d) - 1]$. This is a complex operation we are supporting with extreme simplicity.

LAQ_T(v, d): There is no practical solution in the original proposal. In this thesis we proceed as follows to achieve the cost of d *Parent* operations, plus LAQ_S operations, all of which

are reasonably cheap. Since $SDepth(v) \geq TDepth(v)$, we first try $v' = LAQ_S(v, d)$, which is an ancestor of our answer; let $d' = TDepth(v')$. If $d' = d$ we are done; else $d' < d$ and we try $v'' = LAQ_S(v, d + (d - d'))$. We compute $d'' = TDepth(v'')$ (which is measured by using $d'' - d'$ *Parent* operations until reaching v') and iterate until finding the right node. So the total number of *Parent* operations is d and the LAQ_S are fewer.

7.1 Comparing the CST Implementations

We compare all the CST implementations: Välimäki et al.’s [VGDM07] implementation of Sadakane’s compressed suffix tree [Sad07a] (*CST-Sadakane*); Russo’s implementation of Russo et al.’s “fully-compressed” suffix tree [RNO08b] (*FCST*); and our best variants, called *Our CST* in the plots. Depending on their *LCP* representation, the latter are suffixed with *Sad-Gon*, *FMN-RRR*, *DAC*, and *DAC-Var*. We do not compare all operations. For example we omit *Root* and *Ancestor* because they are trivial in all implementations; *Locate* and *Count* because they depend only on the underlying compressed suffix array (which is mostly orthogonal, thus *Letter* is sufficient to study it); *SLinkⁱ* because it is usually better to do *SLink* i times. Given that LAQ_S and LAQ_T are not implemented in the practical alternative CSTs, we will only present the performance of our implementations for these operations.

We typically show space/time trade-offs for all the structures, where the space is measured in bits per character (recall that these CSTs replace the text, so this is the overall space required). The times are averaged over a number of queries on random nodes. We use four types of node samplings, which make sense in different typical suffix tree traversal scenarios: (a) Collecting the nodes visited over 10,000 traversals from a random leaf to the root (used for *Parent*, *SDepth*, and *Child* operations); (b) same previous sampling but keeping only nodes of depth at least 5 (for *Letter*); (c) collecting the nodes visited over 10,000 traversals from the parent of a random leaf towards the root via suffix links (used for *SLink* and *TDepth*); (d) taking 10,000 random leaf pairs (for *LCA*).

We show the results of each operation over all texts presented in Table 5.1. The standard deviation divided by the average is in the range [0.21,2.56] for *CST-Sadakane*, [0.97,2.68] for *FCST*, [0.65,1.78] for *Our CST Sad-Gon*, [0.64,2.50] for *Our CST FMN-RRR*, [0.59,0.75] for *Our CST DAC*, and [0.63,0.91] for *Our CST DAC-Var*. The standard deviation of the estimator is thus at most 1/100th of that.

7.2 Experimental Comparison

Figures 7.1 and 7.2 show the performance of the operations *Parent* and *TDepth* respectively. As can be seen in the graphs, Sadakane's CST is about 5 to 10 times faster than our best results for *Parent*. Given that our implementation computes *TDepth* by brute force, our best times are about 100 to 1000 slower than Sadakane's results. This big difference in time is because Sadakane's CST stores explicitly the tree topology, taking only a fraction of a microsecond to answer queries like *Parent* and *TDepth*.

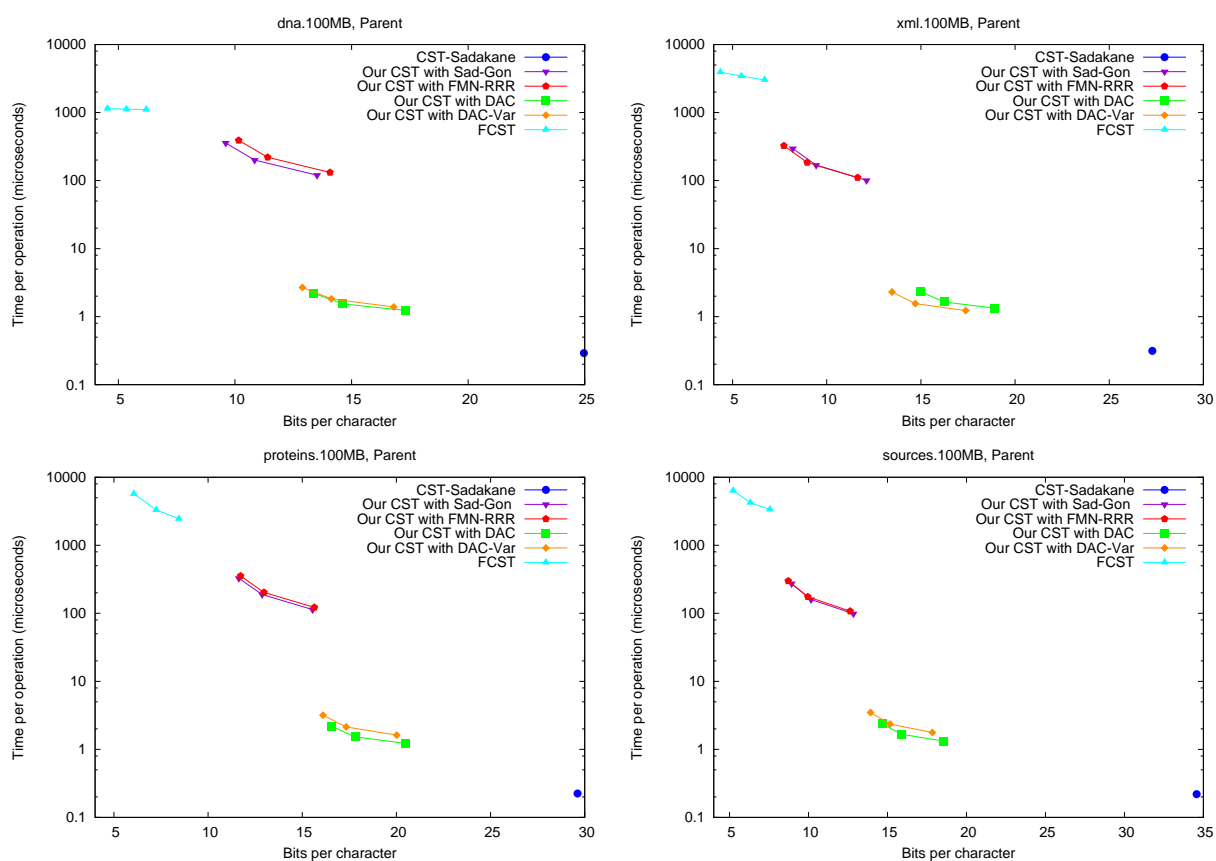


Figure 7.1: Space/time trade-off performance for the operation *Parent*. Note the logscale.

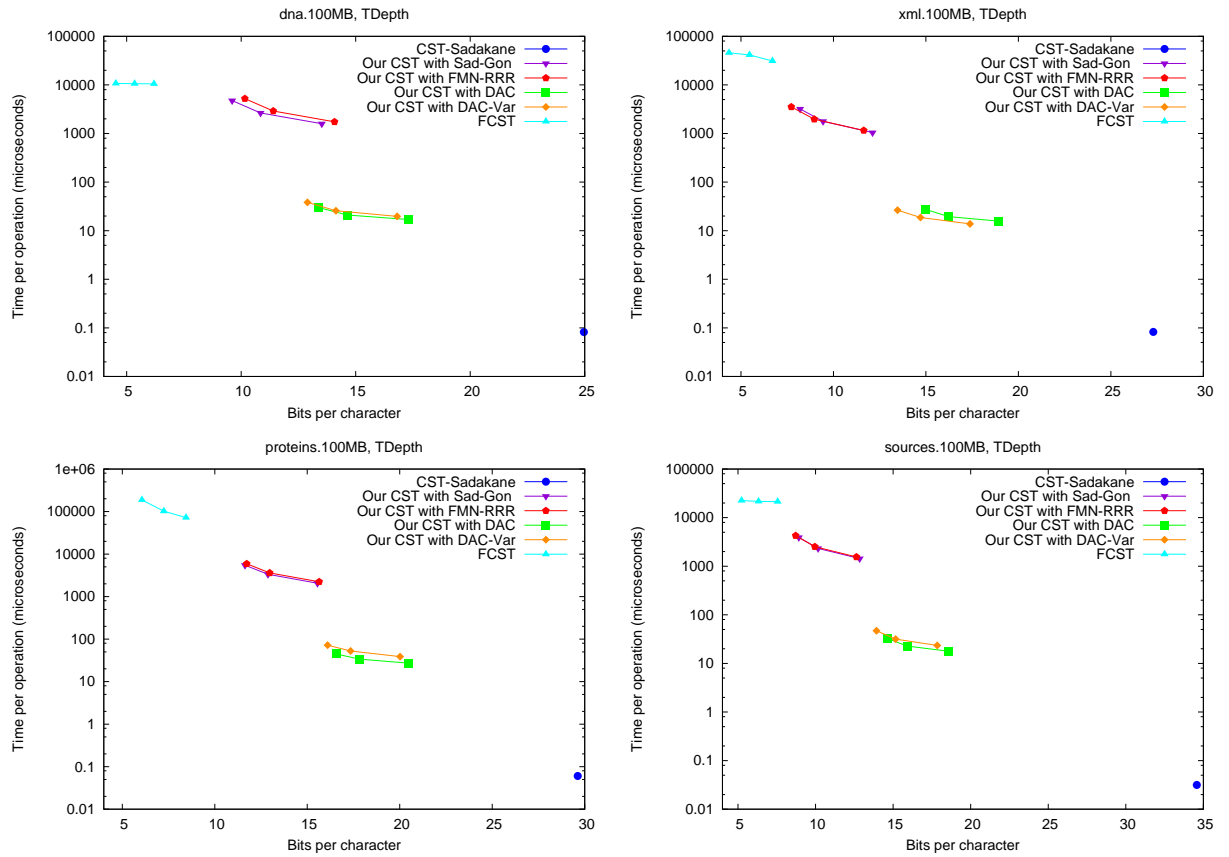


Figure 7.2: Space/time trade off performance for the operation $TDepth$. Note the logscale.

Figures 7.3 to 7.5 show the performance of the operation *SLink*, *LCA*, and *SDepth*. We note that the *FCST* is more efficient on operations *LCA* and *SDepth*, yet it is still slower than our slower variant. We remark that our faster solutions, for these operations, outperforms all the other structures being 10 times faster than Sadakane CST and about 100 to 1000 times faster than the *FCST*.

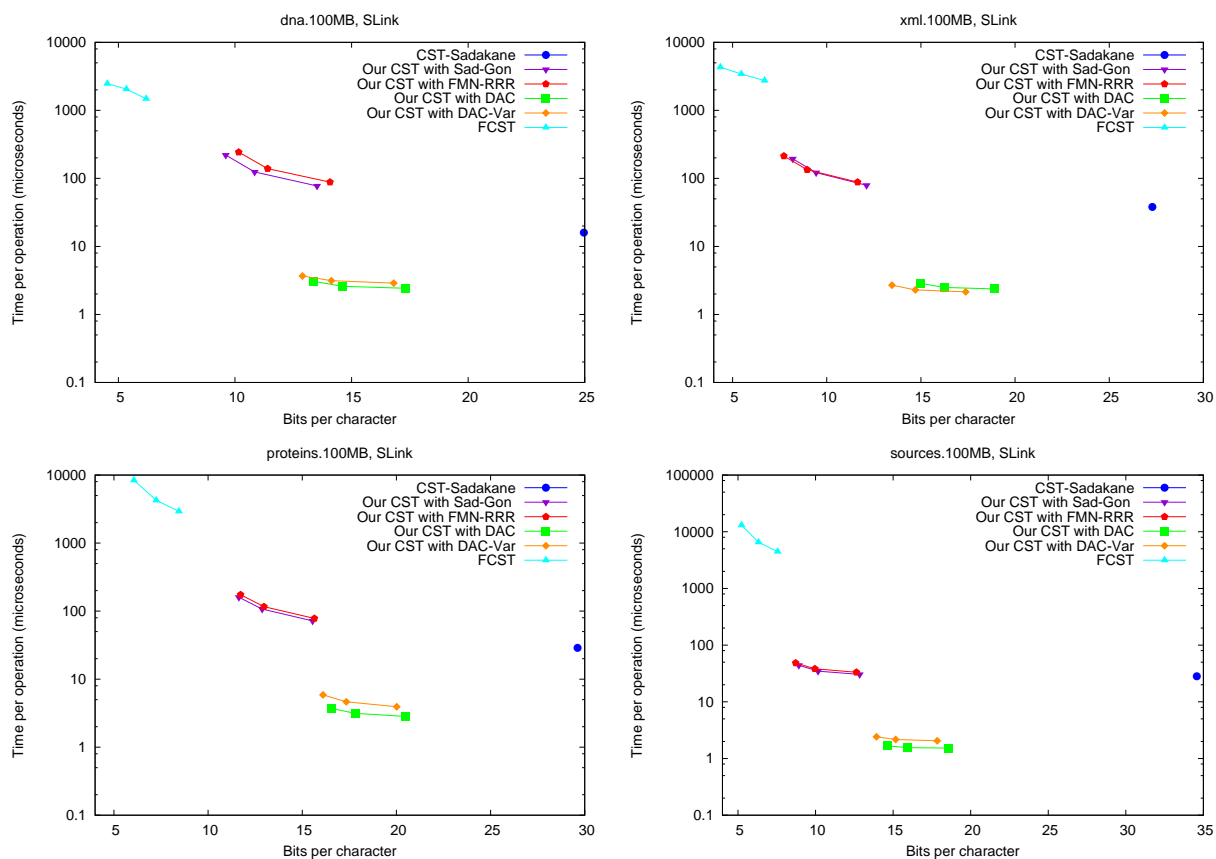


Figure 7.3: Space/time trade-off performance for the operation *SLink*. Note the logscale.

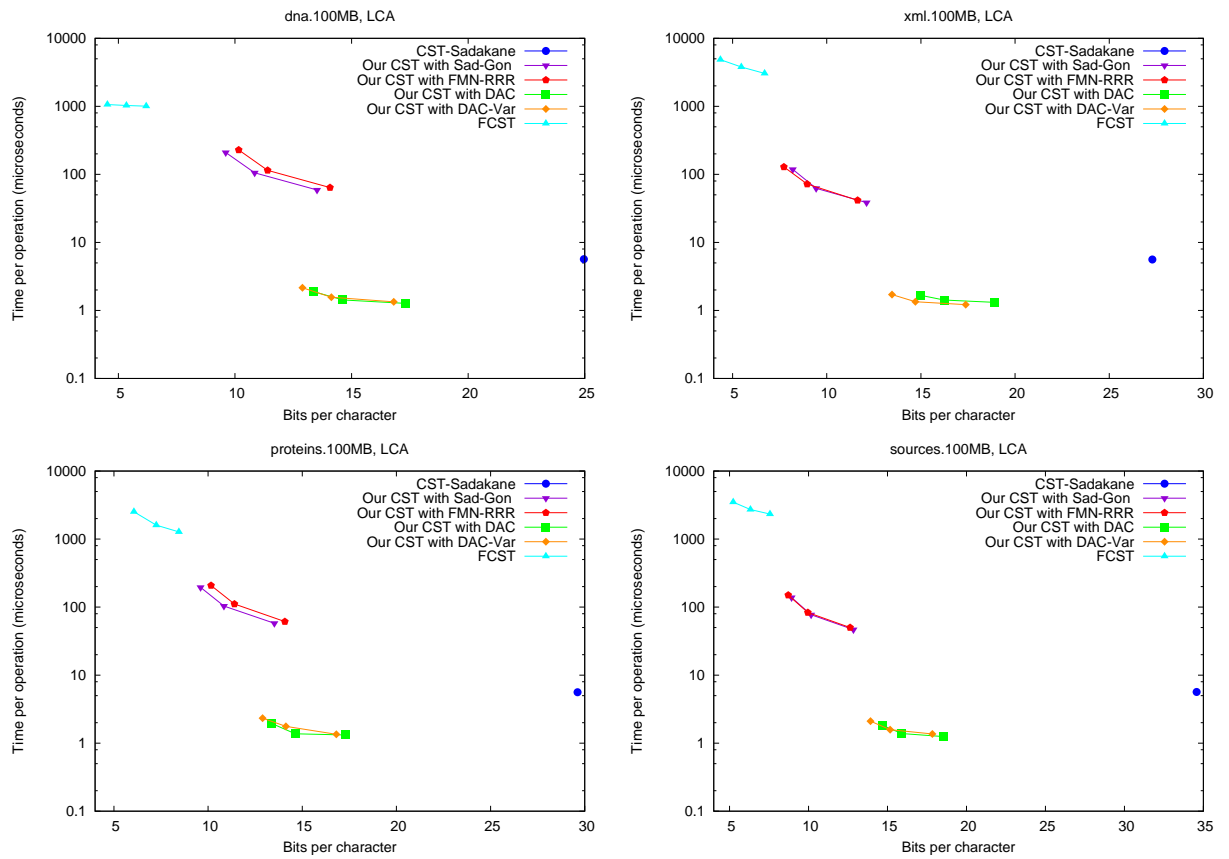


Figure 7.4: Space/time trade-off performance for the operation *LCA*. Note the logscale.

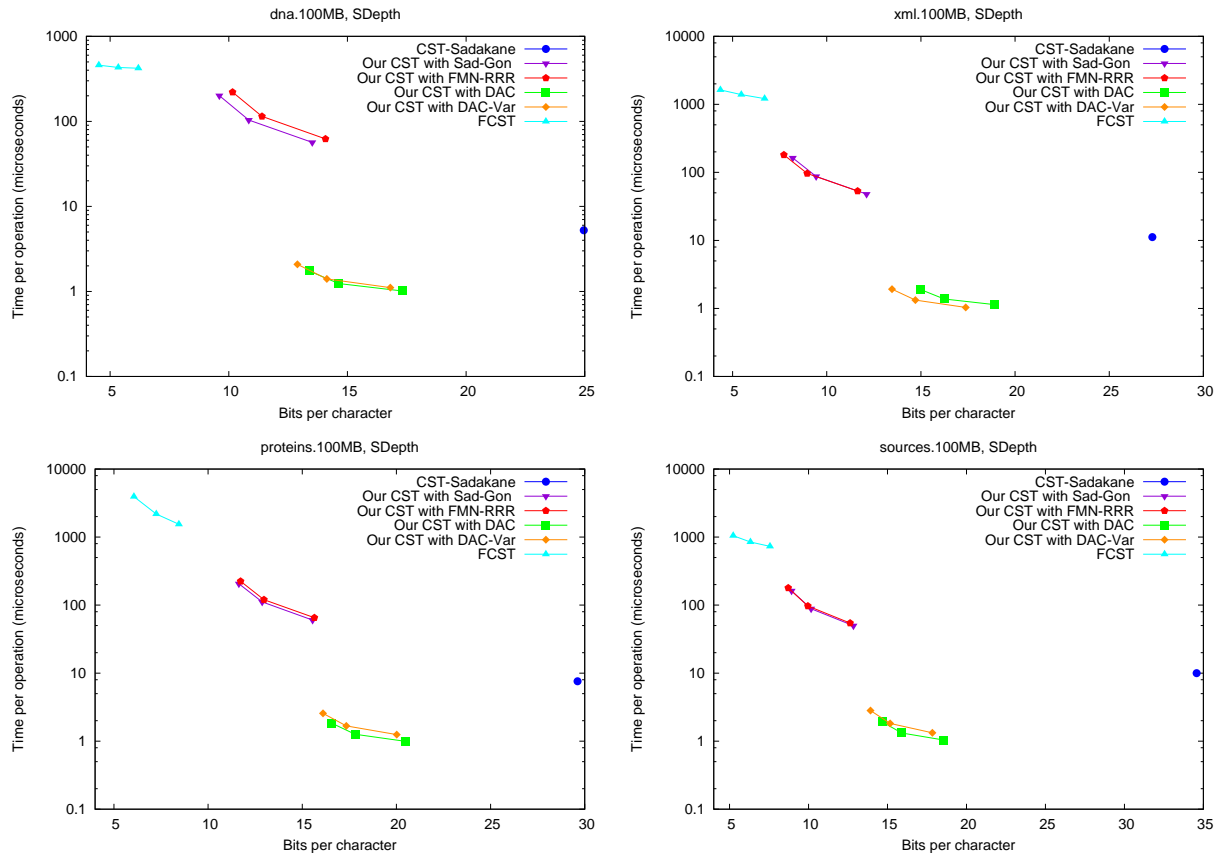


Figure 7.5: Space/time trade-off performance for the operation $SDepth$. Note the logscale.

Figure 7.6 shows operation *Child*, where we descend by a random letter from the current node. Our times are higher than for other operations as expected, yet the same happens to all the implementations. As we can notice in the graphs all the implementations have a different performance for each text and is not possible to conclude anything consistent about which implementation is better. It is important to note that FCST is quite competitive compared to our slow variants.

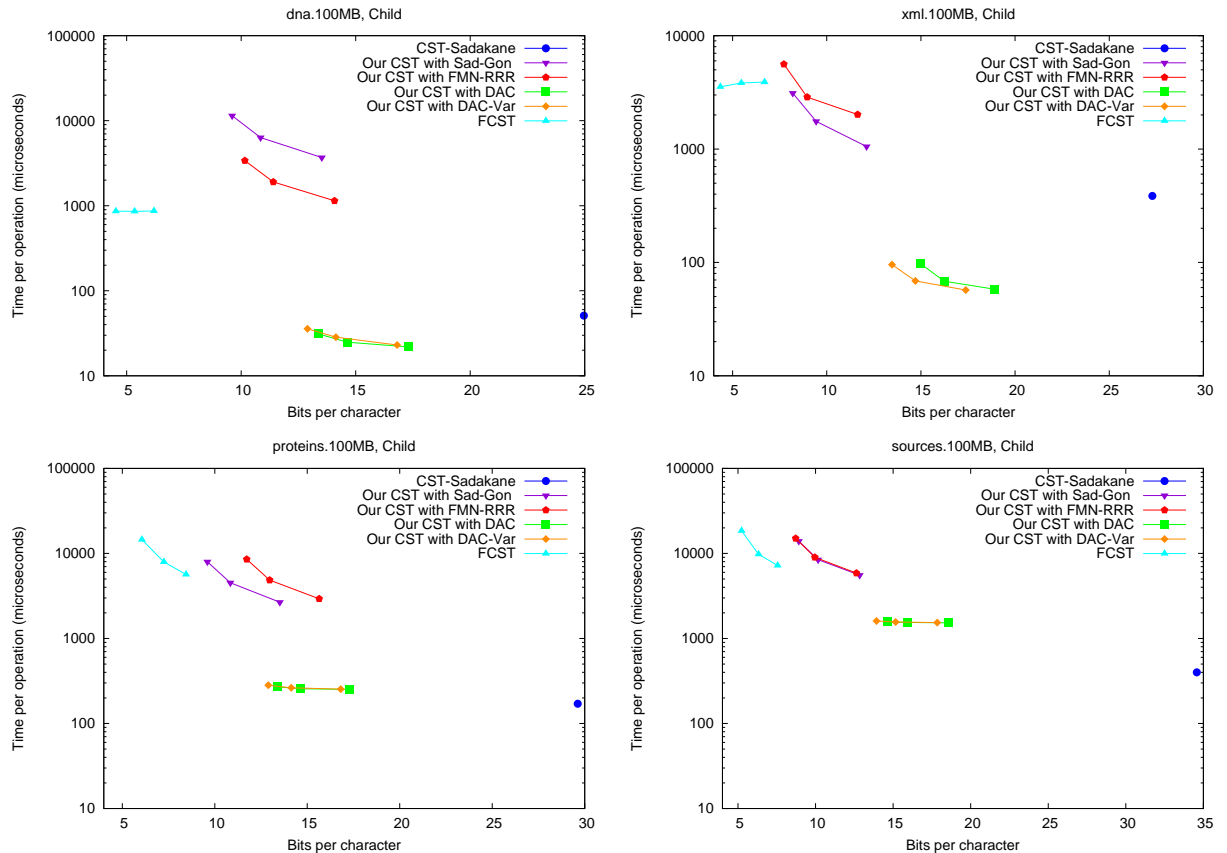


Figure 7.6: Space/time trade-off performance for the operation *Child*. Note the logscale.

Figure 7.7 shows the operation $Letter(i)$, as a function of i . It depends only on the CSA structure, and requires either applying $i - 1$ times Ψ , or applying once A and A^{-1} . The former choice is preferred for the FCST and the latter in Sadakane's CST. For our CST, using Ψ iteratively was better for these i values, as the alternative take around 70 microseconds.

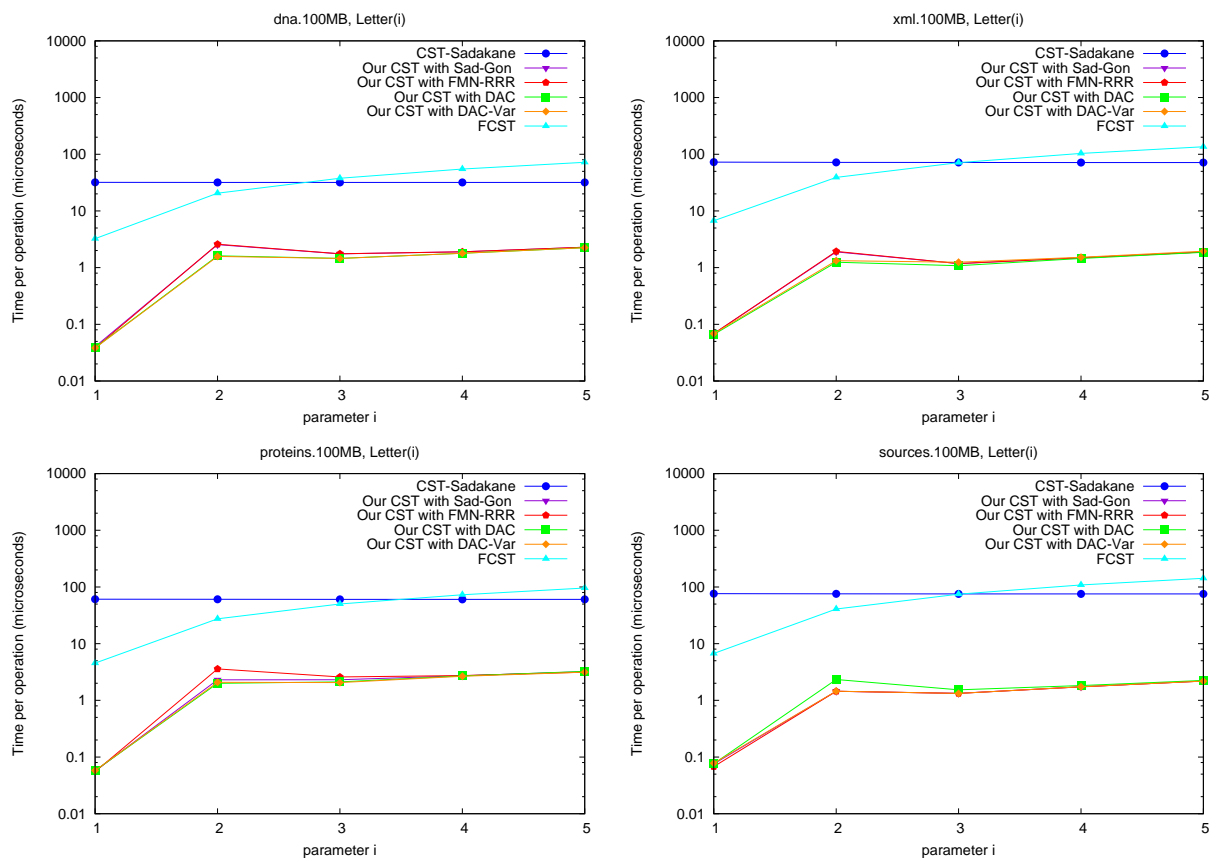


Figure 7.7: Space/time trade-off performance for the operation $Letter$. Note the logscale.

Finally Figure 7.8 shows a basic suffix tree traversal algorithm: the classical one to detect the longest repetition in a text. This traverses all of the internal nodes using *FChild* and *NSibling* and reports the maximum *SDepth*. Although Sadakane’s CST takes advantage of locality, our “large and fast” variants are better using half the space. Our “small and slow” variant, instead, requires a few hundred microseconds as expected, yet the FCST has a special implementation for full traversals and it is competitive with our slow variant.

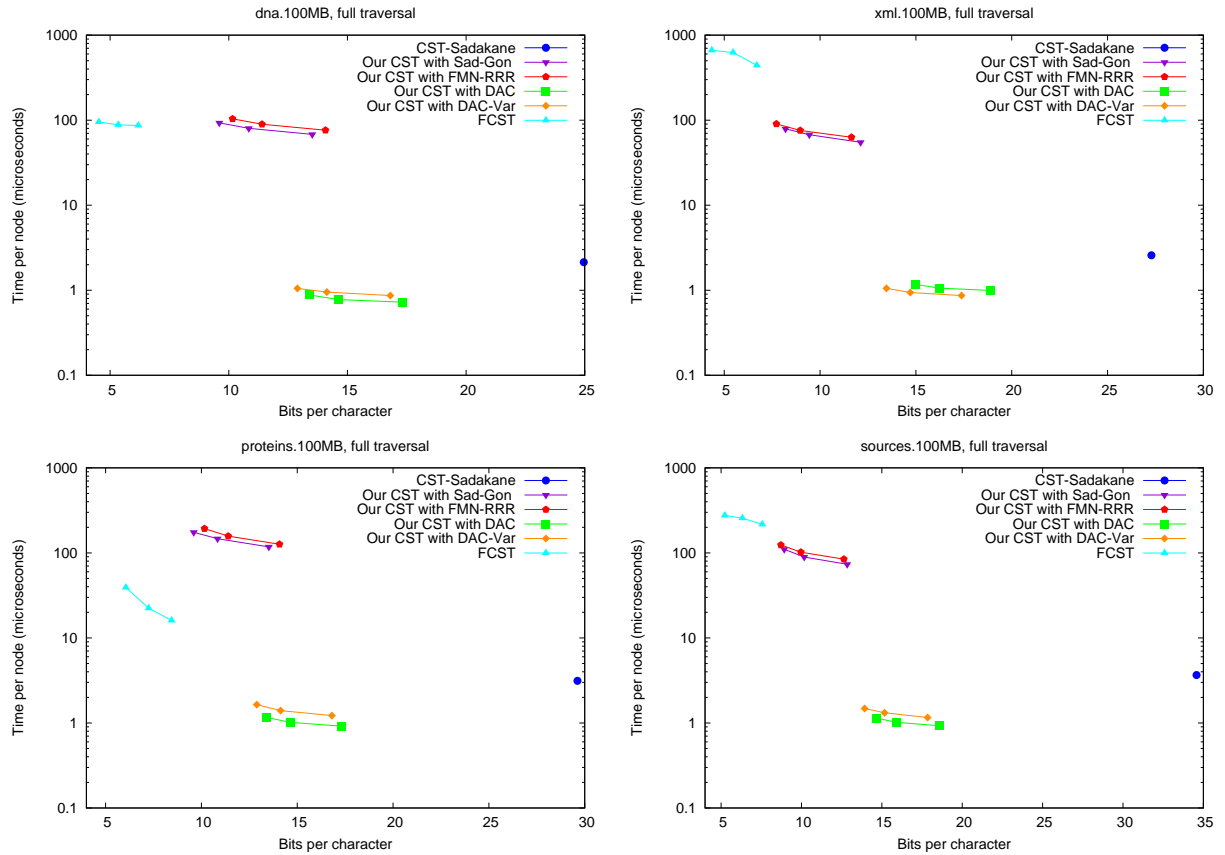


Figure 7.8: Space/time trade-off performance for a full traversal of the tree. Note the logscale.

Figures 7.9 and 7.10 show the performance of the operations $LAQ_S(d)$ and $LAQ_T(d)$ respectively. The times are presented as a function of parameter d , where we use $d = 4, 8, 16, 32, 64$. We query the nodes visited over 10,000 traversals from a random leaf to the root (excluding the root). As expected the performance of LAQ_S is similar to that of *Parent*, and the performance of LAQ_T is close to the time it would take to do d times *Parent*. Note that as we increase the value of d the times tend to a constant. This is because, as we increase d , a greater number of queried nodes are themselves the answer to LAQ_S or LAQ_T queries.

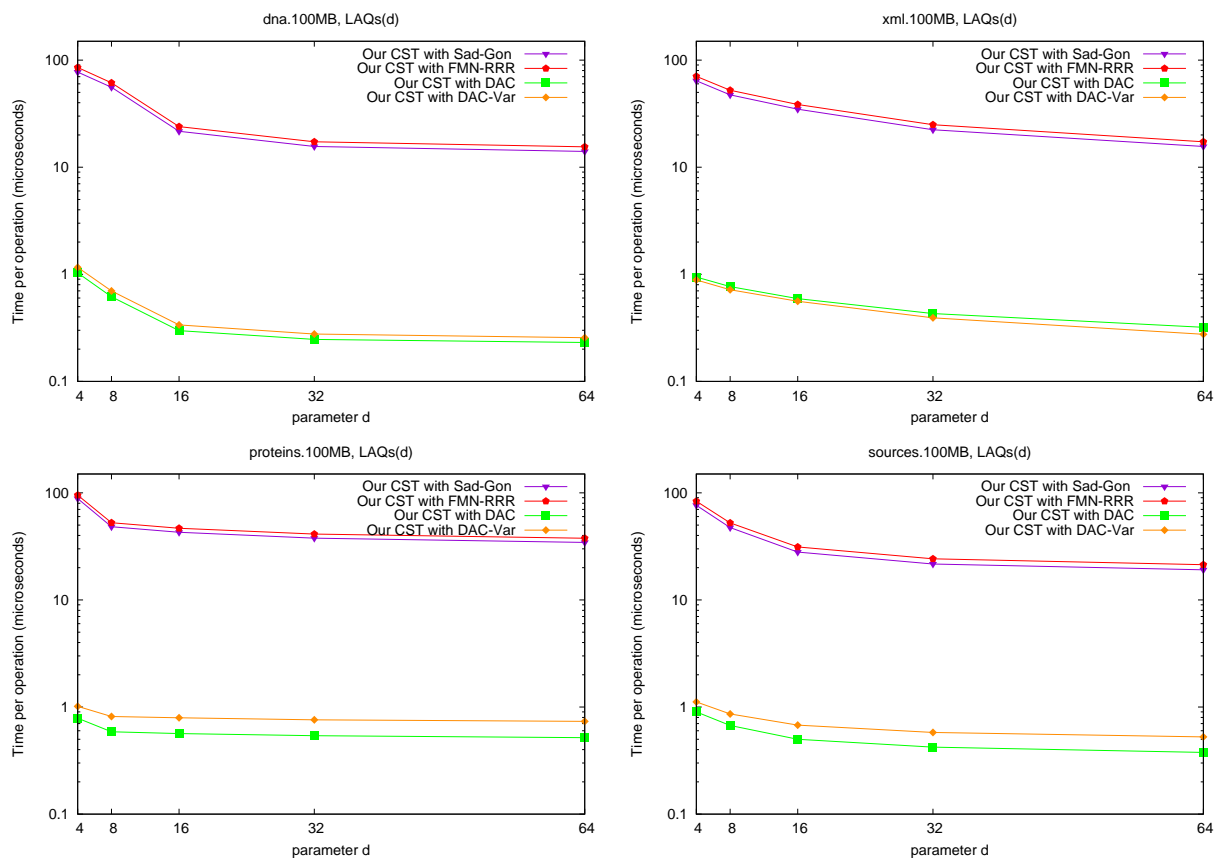


Figure 7.9: Space/time trade-off performance for the operation $LAQ_S(d)$. Note the logscale.

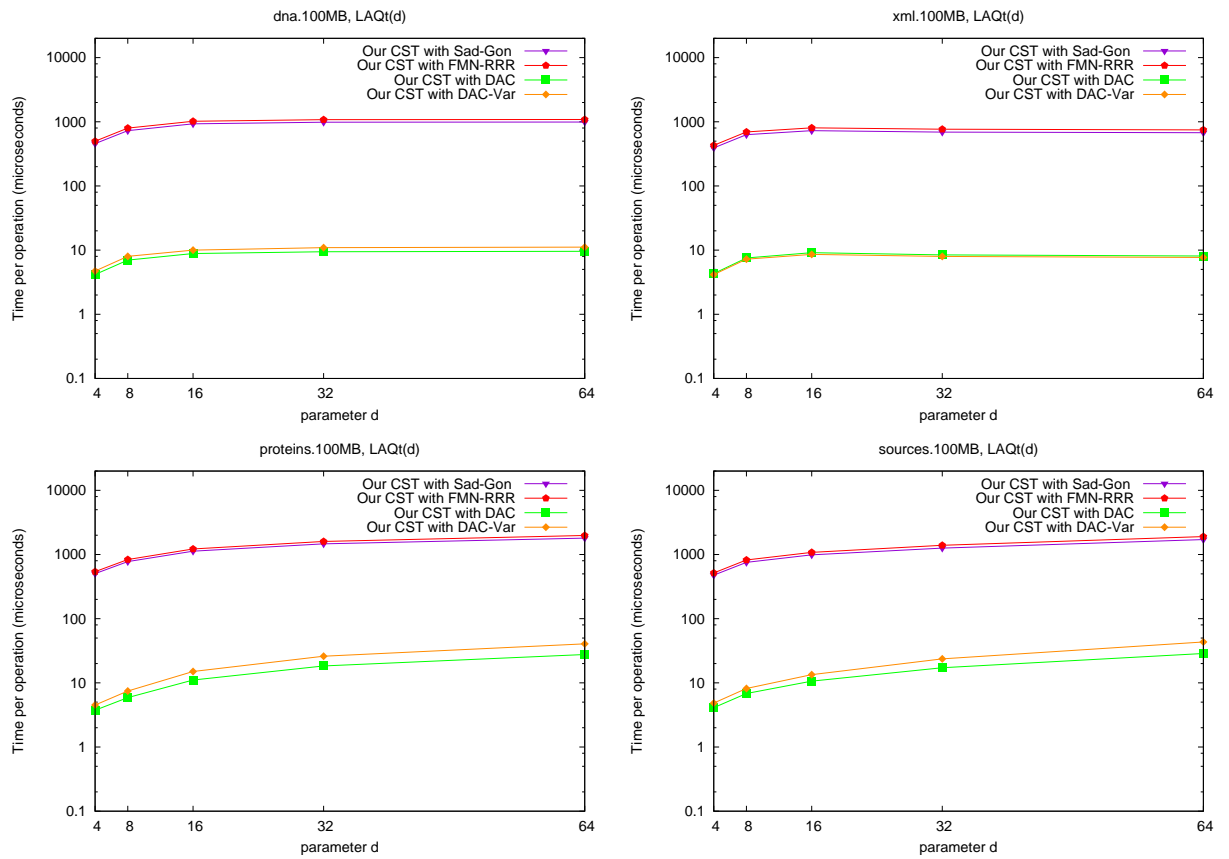


Figure 7.10: Space/time trade-off performance for the operation $LAQ_T(d)$. Note the logscale.

Chapter 8

Epilogue

8.1 Breaking News

This is a very rapidly changing field. Since the core work of this thesis was completed, a few months ago, new ideas and improvements have continued to emerge. These have not been covered in the thesis because they are too recent and have no practical implementation yet. Here we briefly describe some of these novel ideas, which certainly can yield new practical space/time trade-offs for CSTs.

8.1.1 Wee LCP

In a recent paper [Fis10b] Fischer improves Fischer et al.’s compressed suffix tree [FMN09] (recall Section 4.3), requiring only $(1 + \frac{1}{\epsilon})nH_k + o(n)$ bits of space and maintaining the same time complexity for all the operations. This improvement is achieved by replacing the *LCP* representation proposed (see Section 5.2) with a new one.

This new *LCP* representation is built on top of the bitvector H of Sadakane’s *LCP* representation (see Section 5.1). Recall that in Sadakane’s proposal, $LCP[i] = select_1(H, j + 1) - 2j - 1$, where $j = A[i]$. Given that $select_1$ is the only operation needed over H , Fischer presents a way to answer it using less space while maintaining the same time for computing $LCP[i]$.

The scheme is inspired by Clark’s *select* structure [Cla98]. Fischer divides the range of arguments for $select_1$ into subranges of size $k = \lfloor \log^2 n \rfloor$, and stores in $N[i]$ the answer to $select_1(H, ik)$. This table $N[1, \lceil \frac{n}{k} \rceil]$ needs $O(\frac{n}{\log n})$ bits, and divides H into blocks of different sizes, each containing k 1’s (apart from the last).

A block is called *long* if it spans more than k^2 positions in H , and *short* otherwise. For the *long* blocks, he stores all the answers of $select_1$ -queries explicitly in a table P , which requires $O(n/\log n)$ bits. Then he divides the *short* blocks into sub-ranges of size $\lambda = \lfloor \log^2 k \rfloor$, storing in

$N'[1, \lceil \frac{n}{\lambda} \rceil]$ the answers to $select_1(H, i\lambda)$, for $i \in \{1, \lceil \frac{n}{\lambda} \rceil\}$, but this time only the answer relative to the beginning of the block where i occurs. Note that table N' needs $O(n/\log \log n)$ bits. Finally N' divides the blocks into miniblocks, each containing λ 1-bits.

Miniblocks are called *long* if they span more than $s = \log^\delta n$ bits (where $0 < \delta \leq 1$ is an arbitrary constant), and *short* otherwise. For *long* miniblocks, the answers to all *select*-queries are stored explicitly in a table P' , relative to the beginning of the corresponding block, so that table P' requires $O(\frac{n \log^3 \log n}{\log^\delta n})$ bits.

To answer $LCP[i]$ we first need to compute $select_1(H, j)$, for $j = A[i]$, using the above structure. Let $a = select_1(H, \lfloor j/\lambda \rfloor \lambda)$ be the beginning of j 's mini-block in H . Likewise, compute the beginning of the next mini-block as $b = select_1(H, \lfloor j/\lambda \rfloor \lambda + \lambda)$. Now if $b - a > s$, then the mini-block where the j -th one occurs is *long*, and the answer can be looked up using the precomputed tables N, N', P , and P' . Otherwise, return the value a as an approximation to the actual value of $select_1(H, i)$. Then use the text T to compute $LCP[i]$ in additional $O(\log^\delta n)$ time. To this end, let $j' = A[i - 1]$. The unknown value $LCP[i]$ equals the length of the longest common prefix of suffixes $T_{j,n}$ and $T_{j',n}$, so we need to compare these suffixes. However, we do not have to compare letters from scratch, because we already know that the first $m = \max(a - 2j - 1, 0)$ characters of these suffixes match. So we start the comparison at $T[j + m]$ and $T[j' + m]$, and compare as long as they match. Because $b - a \leq s$, we will reach a mismatch after at most $s = O(\log^\delta n)$ character comparisons.

This proposal is competitive in theory, but given the results presented in Chapter 5, we believe that, in practice, the time for computing $LCP[i]$ will be much higher than the obtained with Sadakane's or our CST. This is because the improvement presented by Fischer requires to compute $A[i]$ and $A[i - 1]$, which is slow, and then to access several text symbols using Ψ and the *CSA*. In Section 5.7 we have shown that the representations that needed accesses to the text had the worst time performance.

8.1.2 Range Minimum Queries without LCP Accesses

Recently, Fischer [Fis10a] showed how to preprocess an array L into a scheme of size $2n + o(n)$ bits that allows one to answer range minimum queries on L in constant time. This space is asymptotically optimal in the important setting where access to L is not permitted after the preprocessing step. As we showed in Section 6.3, one problem of our compressed suffix tree is that *RMQ* queries over *LCP* depend entirely on the *LCP* access performance, and moreover several accesses are needed. Therefore, using this novel *RMQ* proposal seems to be a good idea for our implementations. Especially, the "small and slow" ones could stay equally small but not that slow.

The basis for this new succinct structure is the *2d-Min-Heap* [Fis10a]. A *2d-Min-Heap* of L (\mathcal{M}_L) is a labeled and ordered tree with vertices v_0, \dots, v_n , where v_i is labeled with i for all $0 \leq i \leq n$. For $1 \leq i \leq n$, the parent node of v_i is v_j iff $j < i$, $L[j] < L[i]$, and $L[k] \geq L[i]$ for all

$j < k \leq i$. The order of the children is chosen such that their labels are increasing from left to right. This *2d-Min-Heap* of L has the following properties:

- The node labels correspond to the preorder numbers of \mathcal{M}_L .
- Let i be a node in \mathcal{M}_L with children x_1, \dots, x_k . Then $L[i] < L[x_j]$ for all $1 \leq j \leq k$.
- Again, let i be a node in \mathcal{M}_L with children x_1, \dots, x_k . Then $L[x_j] \leq L[x_{j-1}]$ for all $1 < j \leq k$.
- For arbitrary nodes i and j , $1 \leq i < j \leq n$, let l denote the *LCA* of i and j in \mathcal{M}_L . Then if $l = i$, $RMQ_L(i, j)$ is given by i , and otherwise, $RMQ_L(i, j)$ is given by the child of l that is on the path from l to j .

Note that this last property yields the rightmost minimum in the query range if this is not unique. However, it can be easily arranged to return the leftmost minimum by adapting the *2d-Min-Heap*, if this is desired. To achieve the optimal $2n + o(n)$ bits, they represent the *2d-Min-Heap* \mathcal{M}_L using *DFUDS* (see Section 3.3.2) plus an implementation of a restricted *RMQ* over the *DFUDS* sequence, where consecutive values differ by ± 1 . For further details of how the *2d-Min-Heap* works we encourage the reader to read the original paper [Fis10a].

8.1.3 CST++

Ohlebusch et al. have just presented a new representation for the CST [OFG10]. They use a succinct data structure of size $3n + o(n)$ bits that allows them to answer *NSV/PSV/RMQ* queries over an array $L[1, n]$ in constant time, without accessing L , plus a representation of the suffix array and a new approach to compress the *LCP* array. Based on this preprocessing scheme, they present a new compressed suffix tree with $O(1)$ time for most navigation operations, just like Sadakane's CST (which needs $6n$ extra bits instead of $3n$).

Once implemented, it would be interesting to compare experimentally, in terms of space and time, their new *LCP* encoding, *NSV/PSV/RMQ* succinct data structure, and compressed suffix tree (CST++), against all the alternative representations presented in this thesis.

8.2 Conclusion and Future Work

We have presented new practical compressed suffix tree (CST) implementations with full functionality that offer very relevant space/time trade-offs, which are between the two rather extreme existing variants. This opens the door to a number of practical suffix tree applications, particularly relevant to bioinformatics.

Our CSTs can operate within 8–12 bits per character (bpc, that is, at most 50% larger than the plain byte-based representation of the text, and replacing it) while requiring a few hundred

microseconds for most operations (the “small and slow” variants *Sad-Gon* and *FMN-RRR*); or within 13–16 bpc and carry out most operations within a few microseconds (the “large and fast” variants *DAC/DAC-Var*). In contrast, the FCST [RNO08b] requires only 4–6 bits per character (which is, remarkably, as little as half the space required by the plain text representation), but takes the order of milliseconds per operation (which is close to disk access times); and Sadakane’s CST [Sad07a] takes usually a few tens of microseconds per operation but requires 25–35 bits per character, which is close to uncompressed suffix arrays (not uncompressed suffix trees, though).

We remark that, for many operations, our “fast and large” variant uses half the space of Sadakane’s CST implementation and it is many times faster. Exceptions are operations *Parent* and *TDepth*, where Sadakane’s CST stores the explicit tree topology, and thus takes a fraction of a microsecond. On the other hand, our CST carries out operations *LAQ_S* in a similar time as *Parent*, and *LAQ_T(d)* in about d times the time of *Parent*. These operations are much more complicated for the alternative compressed suffix trees; in fact they had not been implemented before.

We have also, in Chapter 3, carried out a rather exhaustive comparison of the best techniques to represent general trees of n nodes in little space. We focused on the most succinct schemes, which use $2n + o(n)$ bits. It turns out that the recent so-called fully-functional (FF) representation [SN10] offers an excellent combination of space usage, time performance, and functionality. It implements a large set of simple and sophisticated operations, and in most cases it outperforms all the other structures in time and space. For example, with less than 20% overhead over the minimum $2n$ bits of space, it carries out all the operations within the microsecond (up to 2 microseconds for a few of them). Our study demonstrates that approaching the $2n$ bits, while offering basic navigation is not so difficult with simple engineered schemes, whereas the most striking aspect of the *FF* representation is the wide functionality it offers within this space.

Finally, it is important to mention that all our implementations have been made public in *Google Code* (<http://code.google.com/p/libcds/>), to foster their use in diverse applications and their comparison to other practical proposals to come.

This work pointed out interesting lines of research for the future. A obvious topic of future work is to implement the proposals of Section 8.1 and experimentally study what new practical space/time trade-offs for CSTs can be achieved.

One thing that we have not considered was the space and time taken by the construction of the structures. As future work, we plan to experimentally compare the construction process of our CST with the existing ones, in terms of space and time, and consider the construction within compressed space, as it has already been done for CSAs [HS03, HSS03, MN08].

Another important topic of future work is to achieve a practical succinct dynamic suffix tree representation, where nodes can be added to and deleted from the tree. In this aspect only theoretical proposals exist [CHLS07, RNO08a].

In Gusfield’s book [Gus97] several applications of suffix trees to bioinformatic problems are

presented. Many of them can be solved using the navigation operations of the CST presented in this thesis. Yet, there are others where it is necessary to store extra information in the internal nodes (which changes across the process), or the number of repetitions of some special substrings, etc. Moreover, for some problems a dynamic CST is needed. Given that the extra information stored in the nodes can change across the process, it is vital to represent it using dynamic compact data structures [CHLS07, GN08, MN08]. Currently there is no study about how these problems can be solved in succinct space, thus it is open for future work to enhance our CST, or to implement new solutions, to solve these problems with compressed data structures.

Bibliography

- [ACNS10] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97. SIAM Press, 2010.
- [AKO04] M. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.
- [Apo85] A. Apostolico. *The myriad virtues of subword trees*, pages 85–96. Combinatorial Algorithms on Words. NATO ISI Series. Springer-Verlag, 1985.
- [BBK04] D. Blandford, G. Blelloch, and I. Kash. An experimental analysis of a compact graph representation. In *Proc. 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 49–61, 2004.
- [BDM⁺05] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
- [BFC00] M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th Latin American Theoretical Informatics Symposium (LATIN)*, LNCS 1776, pages 88–94, 2000.
- [BFC04] M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.
- [BGMR07] J. Barbay, A. Golynski, I. Munro, and S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theoretical Computer Science*, 387(3):284–297, 2007.
- [BHMR07] J. Barbay, M. He, I. Munro, and S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.
- [BK03] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Annual Conference on Combinatorial Pattern Matching (CPM)*, pages 55–69, 2003.

- [BLN09] N. Brisaboa, S. Ladra, and G. Navarro. Directly addressable variable-length codes. In *Proc. 16th String Processing and Information Retrieval Symposium (SPIRE)*, LNCS 5721, pages 122–130, 2009.
- [BSV93] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993.
- [BW94] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.
- [CHLS07] H. Chan, W. Hon, T. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2):article 21, 2007.
- [CKL06] R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: structures for faster text indexing. In *Proc. 33rd International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 4051, pages 358–369, 2006.
- [CL00] C. Colbourn and A. Ling. Quorums from difference covers. *Information Processing Letters*, 75(1-2):9–12, 2000.
- [Cla98] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Ontario, Canada, 1998.
- [CN08] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th String Processing and Information Retrieval Symposium (SPIRE)*, LNCS 5280, pages 176–187, 2008.
- [CN10] R. Cánovas and G. Navarro. Practical compressed suffix trees. In *Proc. 9th International Symposium on Experimental Algorithms (SEA)*, LNCS 6049, pages 94–105, 2010.
- [DRR06] O. Delpratt, N. Rahman, and R. Raman. Engineering the LOUDS succinct tree representation. In *Proc. 5th Workshop on Experimental Algorithms (WEA)*, pages 134–145. LNCS 4007, 2006.
- [Eli75] P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–20, 1975.
- [Far97] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.
- [FGNV09] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13:article 12, 2009.

- [FH07] J. Fischer and V. Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *Proc. 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE)*, LNCS 4614, pages 459–470, 2007.
- [FH10] J. Fischer and V. Heun. Range median of minima queries, super cartesian trees, and text indexing. In *Proc. International Workshop on Combinatorial Algorithms (IWOCA)*, pages 239–252, 2010.
- [Fis10a] J. Fischer. Optimal succinctness for range minimum queries. In *Proc. 9th Latin American Theoretical Informatics Symposium (LATIN)*, LNCS 6034, pages 158–169, 2010.
- [Fis10b] J. Fischer. Wee LCP. *Information Processing Letters*, 110(8-9):317–320, 2010.
- [FLMM05] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. 46th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 184–196, 2005.
- [FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, page 390, 2000.
- [FM08] A. Farzan and I. Munro. A uniform approach towards succinct representation of trees. In *Proc. 11th Scandinavian Workshop on Algorithmic Theory (SWAT)*, LNCS 5124, pages 173–184, 2008.
- [FMMN07] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.
- [FMN09] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.
- [Fre60] E. Fredkin. Trie memory. In *Communications of the ACM* 3, pages 490–500, 1960.
- [GGG⁺07] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. Rao. On the size of succinct indices. In *Proc. 15th European Symposium on Algorithms (ESA)*, pages 371–382. LNCS 4698, 2007.
- [GGMN05] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. 4th Workshop on Experimental Algorithms (WEA)*, pages 27–38, 2005.
- [GGV03] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

- [GHSV06] A. Gupta, W. Hon, R. Shah, and J. Vitter. Compressed dictionaries: Space measures, data sets, and experiments. In *Proc. 5th Workshop on Experimental Algorithms (WEA)*, pages 158–169, 2006.
- [GN07] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.
- [GN08] R. González and G. Navarro. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science*, 410:4414–4422, 2008.
- [GRR04] R. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. 15th Symposium on Discrete Algorithms (SODA)*, pages 1–10, 2004.
- [GRRR06] R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.
- [Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [GV05] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *SIAM Journal on Computing*, pages 397–406, 2005.
- [HMR07] M. He, I. Munro, and S. Rao. Succinct ordinal trees based on tree covering. In *Proc. 34th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 4596, pages 509–520, 2007.
- [HS03] W. Hon and K. Sadakane. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th International Symposium on Algorithms and Computation (ISAAC)*, LNCS 2906, pages 240–249, 2003.
- [HSS03] W. Hon, K. Sadakane, and W. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 251–260, 2003.
- [Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.
- [JSS07] J. Jansson, K. Sadakane, and W. Sung. Ultra-succinct representation of ordered trees. In *Proc. 18th Symposium on Discrete Algorithms (SODA)*, pages 575–584, 2007.
- [KA03] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Conference on Combinatorial pattern matching (CPM)*, pages 200–210, 2003.

- [KA07] P. Ko and S. Aluru. Optimal self-adjusting trees for dynamic string data in secondary storage. In *Proc. 14th String Processing and Information Retrieval Symposium (SPIRE)*, LNCS 4726, pages 184–194, 2007.
- [KLA⁺01] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2001.
- [KMP09] J. Kärkkäinen, G. Manzini, and S. Puglisi. Permuted longest-common-prefix array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5577, pages 181–192, 2009.
- [Knu73] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.
- [KR03] J. Kärkkäinen and S. Rao. *Algorithms for Memory Hierarchies*, chapter 7: Full-text indexes in external memory, pages 149–170. LNCS 2625. Springer, 2003.
- [KS03] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 943–955, 2003.
- [KSB06] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.
- [KSPP03] D. Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Conference on Combinatorial pattern matching (CPM)*, pages 186–199, 2003.
- [Kur99] S. Kurtz. Reducing the space requirements of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.
- [LM00] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. of the IEEE*, 88(11):1722–1732, 2000.
- [LV88] G. Landau and U. Vishkin. Fast string matching with k-differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988.
- [LY08] H. Lu and C. Yeh. Balanced parentheses strike back. *ACM Transactions on Algorithms*, 4(3):article 28, 2008.
- [Man01] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
- [McC76] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 32(2):262–272, 1976.

- [MM93] U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.
- [MN05] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
- [MN07a] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 214–226, 2007.
- [MN07b] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.
- [MN08] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 32, 2008. 38 pages.
- [MR01] I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
- [MR04] I. Munro and S. Rao. Succinct representations of functions. In *Proc. 31th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 3142, pages 1006–1015, 2004.
- [MRR01] I. Munro, V. Raman, and S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.
- [Mun96] I. Munro. Tables. In *Proc. 16th Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.
- [Nav09] G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics*, 13: article 2, 49 pages, 2009.
- [NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
- [OFG10] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *Proc. 17th String Processing and Information Retrieval Symposium (SPIRE)*, 2010. To appear.
- [OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.
- [PST07] S. Puglisi, W. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4, 2007.
- [PT08] S. Puglisi and A. Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *Proc. 19th International Symposium on Algorithms and Computation (ISAAC)*, pages 124–135, 2008.

- [RNO08a] L. Russo, G. Navarro, and A. Oliveira. Dynamic fully-compressed suffix trees. In *Proc. 19th annual symposium on Combinatorial Pattern Matching (CPM)*, pages 191–203, 2008.
- [RNO08b] L. Russo, G. Navarro, and A. Oliveira. Fully-Compressed Suffix Trees. In *Proc. 8th Latin American Theoretical Informatics Symposium (LATIN)*, LNCS 4957, pages 362–373, 2008.
- [RRR02] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k -ary trees and multisets. In *Proc. 13th Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.
- [Sad00] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th International Conference on Algorithms and Computation (ISAAC)*, pages 410–421, 2000.
- [Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
- [Sad07a] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
- [Sad07b] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.
- [SF96] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley-Longman, USA, 1996.
- [SN10] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010.
- [SWK⁺02] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proc. 28th International Conference on Very Large Data Bases (VLDB)*, pages 974–985, 2002.
- [Szp96] W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM Journal on Computing*, 22:1176–1198, 1996.
- [Ukk95] E. Ukkonen. Constructing suffix trees on-line in linear time. *Algorithmica*, 14(3):249–260, 1995.
- [VGDM07] N. Välimäki, W. Gerlach, K. Dixit, and V. Mäkinen. Engineering a compressed suffix tree implementation. In *Proc. 6th Workshop on Experimental Algorithms (WEA)*, pages 217–228, 2007.
- [Wei73] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

- [WZ99] H. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42:193–201, 1999.