Statistical and Repetition-based Compressed Data Structures

Autor: Alberto Ordóñez Pereira

Tesis doctoral UDC / 2014

Directores:
Gonzalo Navarro Badino
Nieves Rodríguez Brisaboa

Departamento de Computación

PhD thesis supervised by
Tesis doctoral dirigida por

Gonzalo Navarro Badino
Departamento de Ciencias de la Computación
Universidad de Chile
Blanco Encalada 2120 Santiago (Chile)
Tel: +56 2 6892736
Fax: +56 2 6895531
gnavarro@dcc.uchile.cl

Nieves Rodríguez Brisaboa
Departamento de Computación
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1243
Fax: +34 981 167160
brisaboa@udc.es

Gonzalo Navarro Badino y Nieves Rodríguez Brisaboa, como directores, acreditamos que esta tesis cumple los requisitos para optar al título de doctor internacional y autorizamos su depósito y defensa por parte de Alberto Ordóñez Pereira cuya firma también se incluye.

Á miña familia.

Acknowledgements

The first acknowledgment words are, undoubtedly, for my thesis directors: Nieves and Gonzalo. Nieves was always there willing to help whenever it was necessary and I cannot be more thankful for that. Thanks also for making everything easier along this period. With people like you, everything becomes very easy. Gonzalo is a big reference for me. Both his capacity and patience amaze me. But not only that. His quality as a person, his hospitality, his advices, and his highly caloric barbecues, are only some of the reasons because I will always be grateful.

I also want to acknowledge the effort of all the committee members and external reviewers for having taking their time and for their highly valuable advices. Thanks to Philip Bille, Simon Gog, Susana Ladra, Veli Mäkinen and Giovanni Manzini.

Thanks also to all the Database Laboratory members, especially to the person who started it all. Thanks to Luis for offering me the opportunity of joining this group and for being there whenever I needed it. Thanks also to Susana, Edu, Óscar, Jose, Fari, Diego, and to many others who have always been there whenever I needed help or advice.

A special mention is deserved by all people, now friends, that I have met along my research visits. There is nothing better than traveling along the world and find people that make you feel like if you were at home. Roberto Know, Francisco Claude, Travis Gagie, Simon Puglisi, Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and probably to many others that I am not mentioning here, thanks for your hospitality and friendship.

I cannot forget my old friends, those who have always been there and that, probably, will remain there forever. This would not be possible without those moments of disconnection and laughter we have lived together. A piece of this thesis also belongs to you.

My family has been fundamental. It all started with you, and I am not only speaking in strictly biological terms. You knew exactly how to guide me in life. You did not show me the path, but you did something much more interesting. You taught me how to find it. You have always been an example for me, and I cannot be prouder of you. My father and mother, Manuel and Elsa, my brother Javier, my grandparents María, Alberto, Sara, and Plácido; my second parents Eliseo and

Felisa, my pseudo-sister Rosamna, and to all the rest, thanks.

Finally, I have to pay tribute to Loida, my fellow and wife. Undoubtedly, this thesis would not be possible without you. Your generosity, your comprehension, your patience, and your willingness to help at any time, have been the pillars that have held me. I hope to live enough years to give you back everything you gave me. Do not ever change (well, you can get your driving license, but the rest let it as it is ;-).

Agradecimientos

Las primeras palabras de agradecimiento van dirigidas a mis dos directores: Nieves y Gonzalo. Nieves siempre ha estado ahí para lo que he necesitado, y no puedo estarle más agradecido por todas las facilidades que me ha dado durante este período. Con personas así, todo se vuelve más sencillo. Gonzalo es todo un referente para mí. Tanto su capacidad como su paciencia no dejan de sorprenderme. Pero no solo eso. Su calidad como persona, su hospitalidad, su honestidad, sus consejos, y sobre todo sus asados altos en calorías hacen que no pueda ser de otra manera.

También quiero agradecer el esfuerzo a todos los miembros del tribunal y revisores externos por su tiempo y consejos de gran valor. Gracias a Philip Bille, Simon Gog, Susana Ladra, Veli Mäkinen y Giovanni Manzini.

También debo darles las gracias a todos los miembros del Laboratorio de Bases de Datos. Especialmente a la persona que lo empezó todo. Luís fue el primero en fijarse en mí y el que siempre estuvo ahí para lo que hiciese falta. Gracias también a Susana, Edu, Óscar, Jose, Fari, Diego, y muchos otros que han estado siempre dispuestos a ayudar, y no debo menos que reconocérselo.

También merecen especial mención aquellas personas, ahora amigos, que conocí durante mis estancias en el extranjero y conferencias. No hay nada mejor que recorrer el mundo y conocer a gente que te hace sentir como si estuvieras en casa. Roberto Konow, Francisco Claude, Travis Gagie, Simon Puglisi, Veli Mäkinen, Djamal Belazzougui, Fabio Cunial y probablemente muchos otros que seguramente me estoy dejando en el tintero. Gracias por vuestra hospitalidad y amistad.

No puede uno olvidarse de los amigos de siempre, esos que siempre han estado ahí y que, seguramente, seguirán estando sin esperar nada a cambio. Todo habría sido muy diferente sin esos momentos de desconexión y risas que vivimos juntos. Un pedazo de esta tesis (pequeño que tampoco quiero que se crezcan) también es suya. Ya invitaréis a algo.

Fundamental ha sido mi familia. Con ella empezó todo, y no estoy hablando en términos estrictamente biológicos, aunque también. Siempre me han sabido guiar. No me enseñaron el camino sino que hicieron algo mucho más importante, me enseñaron a encontrarlo. Su ejemplo ha sido fundamental y no puedo estar más orgulloso de ellos. Mi padre y mi madre, Manuel y Elsa, mi hermano Javier, mis

abuelos María, Alberto, Sara y Plácido, mis segundos padres Eliseo y Felisa, mi pseudo-hermana Rosamna, y a todos los demás, gracias de corazón.

Y por último, aunque no menos importante sino que todo lo contrario, debo rendir homenaje a Loida, mi compañera de camino y esposa. No tengo la menor duda de que esta tesis no habría sido posible sin ella. Su generosidad, su comprensión, su paciencia y su disposición a ayudarme tanto cuando las cosas iban bien como mal, han sido los pilares que han aguantado de mí. Espero que me lleguen los años que me quedan para devolverte todo lo que me has dado. No cambies nunca (bueno, sácate el carnet de conducir, pero lo demás déjalo estar ;-).

Agradecementos

As primeiras liñas van, como non, dirixidas ós meus diretores de tese: Nieves e Gonzalo. Nieves sempre estivo ahí para o que precisei e non lle podo estar máis agradecido por elo, así como por todas as facilidades que me dou durante este periodo. Con persoas así, todo se volve moito máis sinxelo. Gonzalo é todo un referente para min. Tanto a sÃža capacidade coma a sÃža paciencia nunca deixarán de sorpenderme. Pero no só iso. A súa calidade como persoa, hospitalidade, os seus consellos e sobre todo os seus asados con alto aporte calórico fan que non poida ser doutra maneira.

Tamén lle quero mostrar o meu agradecemento ós membros do tribunal e revisores externos polo seu tempo e consellos de gran valor. Gracias Philip Bille, Simon Gog, Susna Ladra, Veli Mäkinen e Giovanni Manzini.

Gracias tamén a todos os membro do Laboratorio de Bases de Datos. E en especial á persoa que o empezou todo. Luís foi o primeiro en fixarse en min e sempre estivo ahí para o que fixese falta. Pero non só él, Susana, Edu, Óscar, Jose, Fari, Diego e moitos outros que sempre sacaron tempo de onde non o tiñan para botarme unha man, e non podeo menos que recoñecerllelo.

Tamén mercen especial atención aquelas persoas, agora amigos, que coñecín durante as miñas estancias no estranxeiro. Non hai nada mellor que recorrer o mundo e coñecer a xente que te fai sentir coma se estiveses na casa. Roberto Konow, Francisco Claude, Travis Gagie, Simon Puglisi, Veli Mäkinen, Djamal Belazzougui, Fabio Cunial e probablemente moitos outros que me estou deixando no tinteiro, gracias pola vosa hospitalidade e amizade.

Non podo tampouco olvidarme dos amigos de sempre, eses que sempre estiveron ahí e que seguramente sigan estando sin esperar nada a cambio. Nada sería posible sen eses meomentos de desconexión e risas que sen eles non sería posible. Un cacho desta tese tamén é vosa (pero pequeno que tampouco quero que vos veñades arriba). Xa invitaredes a algo.

Fundamental foi a miña familia. Con ela empezou todo, e non estou falando en térmos estrictamente biolóxicos, que tamén. Sempre me souperon guiar. Non me enseñaron o camiño senon que fixeron algo moito máis importante, enseñáronme a atopalo. O seu examplo foi fundamental e non podo estar máis orgullosos deles do

que o estou. Ós meus pais, Manuel e Elsa, ó meu irmán Javier, ós meus avós María, Alberto, Sara e Plácido, ós meus segundos pais Eliseo e Felisa, á miña pseudo-irmá Rosamna, e a todos os demáis, gracias de corazón.

E por último pero non menos importante, senon que todo o contrario, debo rendirlle homenaxe a Loida, a miña compañeira de camiño e muller. Non teño a menor dúbida de que esta tese non sería posible sin ela. A súa xenerosidade, comprensión, paciencia e a súa disposición a axudarme tanto nos bos coma nos malos momentos, foron os pilares que me sostiveron durante estes anos. Espero que me cheguen os anos para poder devolverche todo o que me diches. Non cambies nunca (buneo, sácate o carnet de conducir, pero o demáis deixao estar ;-).

Abstract

In this thesis we present several practical compressed data structures that address open problems related to statistically-compressible and highly repetitive databases.

In a the first part, we focus on statistical-based compressed data structures, targeting the problem of managing large alphabets. This problem arises when typical sequence-based compression is used as a basis for compressed data structures representing more general structures like grids and graphs. Concretely, (a) we provide space-efficient solutions to represent prefix-free codes when the alphabet is large; (b) we also present a new wavelet-tree based data structure to solve `rank` and `select` queries that obtains zero-order compression and outperforms previous wavelet tree implementations on large alphabets.

In the second part of this thesis, we focus on highly repetitive datasets. We present (c) a very space efficient grammar-based compressed data structure to solve `rank` and `select` on these scenarios; (d) the first *LZ77*-space bounded compressed data structure that solves `rank` and `select` queries in $O(1)$ time and is in practice almost as fast as statistically-compressed structures; and (e) the first practical version of grammar-compressed tree topologies, obtaining unprecedented results in the representation of repetitive trees.

Additionally, we apply our new solutions to several problems of interest: point grids, inverted indexes, self-indexes, XPath systems, and compressed suffix trees of highly repetitive inputs, displaying various space-time tradeoffs of interest.

Resumen

En esta tesis presentamos varias estructuras de datos comprimidas de naturaleza práctica, centradas en problemas abiertos relacionados con bases de datos estadísticamente compresibles y bases de datos cuyo contenido es altamente repetitivo.

En la primera parte, nos centramos en las estructuras de datos comprimidas para bases de datos estadísticamente compresibles, más concretamente, en problemas relativos al manejo de alfabetos grandes. Este tipo de problemas aparecen cuando usamos técnicas clásicas de compresión estadística en estructuras de datos comprimidas para secuencias, y éstas a su vez se aplican a problemas tales como la representación de grillas de puntos o grafos. Concretamente, (a) presentamos soluciones muy eficientes en términos de espacio para representar códigos libres de prefijo cuando el alfabeto el grande; (b) y también presentamos una nueva estructura de datos comprimida basada en *wavelet trees* para resolver consultas `rank` y `select` que obtiene compresión de orden cero y mejora las implementaciones previas de *wavelet trees* en alfabetos grandes.

En la segunda parte de esta tesis, nos centramos en las bases de datos altamente repetitivas. Presentamos (c) una estructura de datos comprimida basada en gramáticas para resolver consultas `rank` y `select` en este tipo de contextos y que usa muy poco espacio; (d) la primera estructura de datos comprimida que obtiene espacio proporcional al de un compresor *LZ77* y resuelve consultas `rank` y `select` en tiempo $O(1)$, siendo en la práctica casi tan rápido como las estructuras de datos basadas en compresión estadística; (e) la primera estructura de datos práctica que utiliza gramáticas para comprimir topologías de árboles, obteniendo resultados sin precedentes para la representación de árboles repetitivos.

Adicionalmente, mostramos varias aplicaciones en las que las estructuras de datos que proponemos a lo largo de la tesis resultan de utilidad. Desde representaciones de grillas de puntos, índices invertidos, auto-índices, sistemas *XPath*, hasta árboles de sufijos comprimidos para colecciones altamente repetitivas, mostrando diferentes resultados de interés tanto en términos de tiempo como de espacio.

Resumo

Nesta tese presentamos varias estruturas de datos comprimidas de natureza práctica, centradas en problemas abertos no ámbito das bases de datos estatisticamente compresibles e das bases de datos altamente repetitivas.

Na primeira parte da tese, centrámonos nas estruturas de datos comprimidas para as bases de datos estatisticamente compresibles. Máis concretamente en problemas relativos ó manexo de alfabetos grandes. Este tipo de problemas aparecen cando usamos técnicas de compresión estatística en estruturas de datos comprimidas para secuencias, e esta á sua vez se utilizan para aplicacións tales como a representación de grellas de puntos ou para a representación de grafos. Concretamente, (a) presentamos solucións que son moi eficientes en termos espaciais para representar códigos libres de prefixo cando o alfabeto é grande; e (b) tamén presentamos unha nova estructura de datos comprimida baseada en *wavelet trees* para resolver consultas `rank` e `select` que obtén compresión de orde cero e mellora as implementacións previas de *wavelet trees* para alfabetos grandes.

Na segunda parte da tese, centrámosnos nas bases de datos con contido altamente repetitivo. Presentamos (c) unha estrutura de datos comprimida baseada en gramáticas que usa moi pouco espazo e resolve eficientemente consultas `rank` e `select` en este tipo de contextos repetitivos; (d) a primeira estrutura de datos comprimida que obtén espazo proporcional ó que obtén un compresor *LZ77* e resolve consultas `rank` e `select` en tempo $O(1)$, sendo na práctica tan rápido coma as estruturas de datos baseadas en compresión estatística; (e) a primeira estrutura de datos práctica que utiliza gramáticas para comprimir topoloxías de árbores, obtendo uns resultados sin precedentes para a representación de árbores repetitivos.

Adicionalmente, mostramos varias aplicacións nas que as estruturas de datos que propoñemos ó longo da tese resultan de utilidade: representacións de grellas de puntos, índices invertidos, auto-índices, sistemas *XPath* e árbores de sufixos comprimidos para colecións altamente repetitivas, mostrando diferentes resultados de interese, tanto en termos de espazo coma de tempo.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Contributions 4
1.3 Structure of the Thesis 6

I Statistical Encodings **9**

2 Previous Concepts on Statistical Encodings **11**
2.1 Model of Computation 11
2.2 Information Theory, Entropy, and Encodings 11
2.3 Empirical Entropy 13
2.4 Entropy Bounded Encodings 14
2.4.1 Shannon-Fano Encoding 14
2.4.2 Huffman Encoding 15
2.4.2.1 Canonical Huffman 16
2.5 Encoding Schemes for Integers 17
2.5.1 Encodings for Small Integers 19
2.5.2 Encodings for Large Integers 20
2.6 Rank, Select, and Access on Bitmaps 20
2.7 Rank, Select and Access on Sequences 23
2.7.1 Wavelet Trees 23
2.7.2 Huffman-Shaped Wavelet Trees 25
2.7.3 GMR Representation 26
2.7.4 Alphabet Partitioning 28
2.8 Directly Addressable Codes 29
2.9 Static Succinct Tree Representations 30
2.10 Text 32

3 Efficient Representation of Prefix Codes **33**
3.1 Related Work . . . 35
3.2 Representing Optimal Codes . . . 37
3.3 Additive Approximation . . . 39
3.4 Multiplicative Approximation . . . 40
3.5 Experimental Results . . . 44
3.5.1 Implementations . . . 44
3.5.1.1 Classical Huffman codes . . . 45
3.5.1.2 Hu-Tucker codes . . . 45
3.5.2 Experimental Setup . . . 46
3.5.3 Representing Optimal Codes . . . 47
3.5.4 Length-Limited Codes . . . 49
3.5.5 Approximations . . . 51
3.6 Discussion . . . 53

4 The Compressed Wavelet Matrix **57**
4.1 Related Work . . . 59
4.1.1 Pointerless Wavelet Trees . . . 59
4.1.1.1 The Strict Variant . . . 59
4.1.1.2 The Extended Variant . . . 61
4.1.2 The Wavelet Matrix . . . 62
4.1.2.1 The Strict Variant . . . 63
4.1.2.2 The Extended Variant . . . 64
4.1.2.3 Construction . . . 65
4.1.3 Pointerless Huffman Shaped Wavelet Trees . . . 65
4.2 The Compressed Wavelet Matrix . . . 67
4.3 Experimental Results . . . 72
4.3.1 Datasets . . . 72
4.3.2 Measurements . . . 73
4.3.3 Results on Sequences . . . 73
4.3.3.1 Space . . . 74
4.3.3.2 Time . . . 75
4.3.3.3 Bottom Line . . . 76
4.4 Discussion . . . 77

II Repetition-based Data Structures **81**

5 Previous Concepts on Repetitive Scenarios **83**
5.1 Why Repetition-based Data Structures? . . . 83
5.2 Kolmogorov Complexity . . . 84
5.3 Lempel-Ziv Parsings . . . 84
5.3.1 LZ77 . . . 85

5.3.2 LZ78 . . . 85
5.3.3 LZ-End . . . 86
5.4 Grammar Compression . . . 86
5.5 Rank, Select, and Access on Repetitive Scenarios . . . 87
5.5.1 Rank, Select, and Access on Repetitive Bitmaps . . . 88
5.5.2 Rank, Select, and Access on Repetitive Sequences . . . 90
5.6 Repetitive Datasets . . . 90

6 Grammar Compressed Sequences 93
6.1 Efficient `rsa` for Sequences on Small Alphabets . . . 94
6.2 Efficient `rsa` for Sequences on Large Alphabets . . . 95
6.2.1 `AP` with `GCC` in Practice . . . 96
6.3 Experimental Results . . . 96
6.3.1 Setup and Datasets . . . 96
6.3.2 Parameterizing the Data Structures . . . 96
6.3.3 Performance on Small Alphabets . . . 98
6.3.4 Performance on Large Alphabets . . . 101
6.4 Discussion . . . 103

7 Block Trees for Sequences 105
7.1 Block Graphs . . . 106
7.2 Block Trees . . . 107
7.2.1 Block Trees Structure . . . 107
7.2.1.1 Analysis . . . 108
7.2.2 Construction . . . 110
7.2.3 Queries on a Block Tree . . . 111
7.2.3.1 Access and Extract . . . 112
7.2.3.2 Rank . . . 112
7.2.3.3 Select . . . 114
7.3 Block Trees for Sequences for Large Alphabets . . . 114
7.4 Block Trees in Practice . . . 115
7.5 Experimental Results . . . 116
7.5.1 Performance on Bitmaps . . . 116
7.5.2 Performance on Sequences With Small Alphabets . . . 117
7.6 Discussion . . . 121

8 Grammar Compressed Trees 127
8.1 Related Work . . . 129
8.2 Grammar Compressed Tree . . . 132
8.2.1 GCT Structure . . . 132
8.2.1.1 Storing the Rules R . . . 132
8.2.1.2 Storing Information on the Rules . . . 134
8.2.1.3 Storing the Array C . . . 135

8.2.2 Basic Operations . . . 135
8.2.3 Operations *fwd* and *bwd* . . . 137
8.2.4 Operation *RMQ* . . . 139
8.2.5 Mapping with Leaves . . . 140
8.3 Experimental Results . . . 141
8.3.1 Environmental Set-up and Datasets . . . 141
8.3.2 Parameterizing the Data Structures . . . 142
8.3.3 Space Performance . . . 142
8.3.4 Time Performance . . . 142
8.4 Discussion . . . 144

III Applications 151

9 Representation of Point Grids 153
9.1 Representing Grids with Wavelet Trees . . . 154
9.2 Experimental Results . . . 156
9.3 Discussion . . . 158

10 Inverted Indexes 161
10.1 Inverted Indexes with `rsa` Data Structures . . . 162
10.2 Experimental Results . . . 162
10.3 Discussion . . . 165

11 Self-Indexes on Highly Repetitive Sequences 167
11.1 Statistically-bounded Self-Indexes . . . 168
11.1.1 Compressed Suffix Arrays . . . 168
11.1.1.1 Compressing Ψ . . . 169
11.1.1.2 Compressing the Suffix Array . . . 169
11.1.1.3 Compressing the Inverse of the Suffix Array . . . 172
11.1.2 FM-Indexes . . . 172
11.2 Self-Indexes on Highly Repetitive Scenarios . . . 173
11.3 Grammar and Block-Tree FM-Indexes . . . 174
11.4 Experimental Results . . . 174
11.5 Discussion . . . 175

12 XPath on Repetitive XML 181
12.1 SXSI on Highly Repetitive Scenarios . . . 182
12.2 Experimental Results . . . 183
12.3 Discussion . . . 184

13 GCST: Grammar Compressed Suffix Tree **185**
13.1 Current Compressed Suffix Trees 187
13.2 Grammar Compressed Suffix Tree 189
13.3 Experimental Results 189
13.3.1 Space Usage 189
13.3.2 Space-Time Performance of Operations 190
13.3.3 Discussion 206
13.4 Discussion 213

IV Thesis Summary **215**

14 Conclusions and Future Work **217**
14.1 Conclusions 217
14.2 Future work 218

Appendices **221**

A Publications and Other Research Results **223**

B Resumen del Trabajo Realizado **225**
B.1 Estructura de la Tesis y Contribuciones 228
B.2 Trabajo Futuro 232

Bibliography **234**

List of Figures

2.1 Relation between classes of encodings. 13
2.2 Example of a Huffman tree. 16
2.3 Example of two Huffman trees with the same average code length $\mathcal{L} = 2$. 18
2.4 Example of a Huffman tree and an equivalent *canonical* Huffman tree 18
2.5 Example of a standard wavelet tree (`WT`) over a sequence. 24
2.6 Example of a Multi-ary wavelet tree (`MWT`) over a sequence 26
2.7 Example of a wavelet tree (`WT`) versus its Huffman-shaped version (`WTH`) 27
2.8 Alphabet Partitioning example. 28
2.9 Example of a DAC 30
2.10 Example of several ordinal succinct tree representations 31

3.1 An arbitrary canonical prefix code and the result of sorting the source symbols at each level 37
3.2 An example of Milidiú and Laber's algorithm 41
3.3 An example of the multiplicative approximation 42
3.4 Code representation size versus compression and decompression time for table based representations and ours proposal 48
3.5 Comparison of the length-restricted approaches measured as their additive redundancy 50
3.6 Relation between the size of the model and the average code length . 52
3.7 Space/time performance of the approximate and exact approaches on compression and decompression 55

4.1 Examples of a wavelet tree and its pointerless version 60
4.2 Example of a pointerless wavelet tree and a wavelet matrix 64
4.3 An example of a pointer-based canonical Huffman wavelet tree (`WTH`) versus its pointerless representation 68
4.4 Example of a sequence of canonical codes along wavelet matrix levels, showing that the leaves do not span a contiguous area. 69
4.5 Example of a Huffman Tree and a compressed wavelet matrix 71

4.6 Running time per `access` query over the four datasets. 78
4.7 Running time per `rank` query over the four datasets. 79
4.8 Running time per `select` query over the four datasets. 80

5.1 Example of an *LZ77*, *LZ78*, and *LZ-End* parsings 85
5.2 Data structures resulted of executing the *RePair* algorithm on a sequence 88
5.3 Expansion tree of a grammar rule 89

6.1 Comparison of `rank` and `select` performance of `GCC.N` and `GCC.C`. . . 99
6.2 Space-time tradeoffs for `rank` and `select` queries over small alphabets 100
6.3 Space-time tradeoffs for `rank` and `select` queries over moderate and large alphabets 102

7.1 Example of a Block Graph 107
7.2 Example of a Block Tree 109
7.3 Example of `rank` mappings in a Block Tree 113
7.4 Space-time tradeoffs for `access` queries on bitmaps 118
7.5 Space-time tradeoffs for `rank` queries on bitmaps 119
7.6 Space-time tradeoffs for `select` queries on bitmaps 120
7.7 Space-time tradeoffs for `access` queries over small alphabets 122
7.8 Space-time tradeoffs for `rank` queries over small alphabets 123
7.9 Space-time tradeoffs for `select` queries over small alphabets 124

8.1 A DAG representation of a tree T 128
8.2 *TreeRepair* applied to tree T 129
8.3 Example of an ordinal tree and its balanced parentheses representation 131
8.4 Compressed dictionary representation for grammars 133
8.5 Tree structure built on top of the C array 136
8.6 General scheme of the algorithm for $fwd(p, d)$ operation 138
8.7 Space breakdown of the `GCT` 143
8.8 Space-time tradeoffs for operation *fChild*. 145
8.9 Space-time tradeoffs for operation *tDepth*. 146
8.10 Space-time tradeoffs for operation *nSibling*. 147
8.11 Space-time tradeoffs for operation *parent*. 148
8.12 Space-time tradeoffs for operation *tAncestor*. 149
8.13 Space-time tradeoffs for operation *LCA*. 150

9.1 Example of a grid of 12×12 with only a marked square per row and column. 154
9.2 Running time per `count` query over the three datasets. 159
9.3 Running time of `report` query over the three datasets. 160

10.1 Space-time tradeoffs for inverted index operations 164

11.1 Suffix tree example with all the components necessary for CSAs and FM-Indexes. 171
11.2 Space-time tradeoffs for operation `count` with $m = 2$. 176
11.3 Space-time tradeoffs for operation `count` with $m = 4$. 177
11.4 Space-time tradeoffs for operation `count` with $m = 8$. 178
11.5 Space-time tradeoffs for operation `count` with $m = 16$. 179

13.1 Space breakdown of the `GCST` 191
13.2 Space-time tradeoffs for operation *fChild*. 196
13.3 Space-time tradeoffs for operation *tDepth*. 197
13.4 Space-time tradeoffs for operation *nSibling*. 198
13.5 Space-time tradeoffs for operation *parent*. 199
13.6 Space-time tradeoffs for operation *tAncestor*. 200
13.7 Space-time tradeoffs for operation *LCA*. 201
13.8 Space-time tradeoffs for operation *sLink*. 203
13.9 Space-time tradeoffs for operation *sDepth*. 204
13.10 Space-time tradeoffs for operation *LAQs*. 205
13.11 Space-time tradeoffs for operation *letter*. 207
13.12 Space-time tradeoffs for operation *child*. 208
13.13 Space-time tradeoffs for finding the maximal substrings 211
13.14 Approximate space figures for the different CSTs 212
13.15 Construction times for the different indexes 213

List of Tables

2.1 Examples of variable length encodings 20
2.2 List of operations in *ordinal* trees. 32

3.1 Main statistics of the texts used to evaluate prefix code representations 46
3.2 Rough minimum size of various model representations for prefix-free codes 47

5.1 Statistics of the repetitive datasets 92

12.1 Results of the XPath system for highly repetitive XML 184

13.1 Typical suffix tree operations 188
13.2 Operation time ranges for the `GCST` and orders of magnitude of difference with alternative CSTs 209

List of Algorithms

1 Building a $\mathcal{D}$-ary Huffman Tree 17
2 Obtaining codes from a Huffman Tree 19
3 Standard Wavelet Tree Algorithms 25
4 Alphabet Partition Algorithms 29
5 Pointerless Wavelet Tree Algorithms (Strict Variant) 61
6 Pointerless Wavelet Tree Algorithms (Extended Variant) 63
7 Wavelet Matrix Algorithms (Strict Variant) 65
8 Wavelet Matrix Algorithms (Extended Variant) 66
9 Range Search Algorithms on a Wavelet Tree 155
10 Range Search Algorithms on Pointerless Wavelet Trees 156
11 Range Search Algorithms on the Wavelet Matrix 157
12 Reporting the positions of a term in a document with a `rsa` inverted index 162
13 Reporting the list of documents that contains two terms in a `rsa` inverted index 163
14 Searching for a pattern in a suffix array using the Ψ function 170
15 Extracting a suffix given the suffix array position 170
16 Counting the occurrences of a pattern in an FM-Index 173

Chapter 1

Introduction

1.1 Motivation

The amount of stored data has increased exponentially over the past few decades, and it seems this tendency will last. Databases are being flooded with tons of data, coming from many different sources and with very different properties. For example, text databases resulting from Web crawling processes or document digitalization have grown faster than the capacity of many organizations to store them. The challenge is exacerbated by applications like versioning control systems or software repositories, in which we want to access the history or versions of a document. This implies having to store all versions of a document, with the tremendous space impact it may induce. We have also DNA databases, in which we store genomes of many individuals of the same species. This is a challenge since DNA databases have grown at the same pace as the costs of sequencing have decreased, that is, faster than the improvements in hardware capabilities. Storing and querying the structure of social networks or Web graphs is also a big challenge due to the number of nodes and connections involved. Geographic Information Systems (GIS) are other examples of applications in which the volume of data generated is massive.

However, not always the increase in the amount of data is the problem, since the other side of the coin is the storage capacity. With the widespread adoption of small and mobile devices (smartphones, tables, or network sensors), it is also common to work with reduced processing and storage capacity. These devices should be able to manage as much information as possible but in an space-constrained environment, without the possibility to resort to secondary storage. In addition, transferring data in compressed form saves their network bandwidth and battery life.

Fortunately, most data involved on these applications and scenarios is not of random nature. Instead, it can be usually modeled according to some statistical model, or at least it has some properties that make it predictable. And predictable,

in Information Theory, means *compressible*. Since the publication of Shannon's thesis, a number of *compressed representations* for a wide variety of data sources have been designed. Finding compressed representations which use as little space as possible to represent data is a first order need, not only for the obvious economical reasons related to saving disk space. Data compression is also fundamental to speed up data transmission through networks, which speeds up computations in cluster-based environments, or to save energy in mobile devices by minimizing the size of messages sent through wireless networks. However, the handicap of these compressed representations is that they only offer compression. If we want to carry out a search inside the compressed data, we have to decompress, and then to carry out the search in the bare data. This means that, in case we need complex functionalities, some of the benefits of compressed representations may vanish.

In order to deal with this problem, the advent of *compressed data structures* (CDS) was a relief. A *compressed data structure* is a data structure that not only cares about space but also about functionality. The space performance is still the lighthouse, but they also provide some operations that are directly performed on the compressed data. This means, for instance, not having to decompress a whole compressed document to carry out a search in it, or to access a random portion of the document. Instead, compressed data structures provide search capabilities within the compressed space. Note this is actually a tremendous step forward since, (a) we have the same benefits of classical compressed representations, (b) we save time if we avoid decompressing the whole data to just access to a portion, (c) we speed up searches since we do not need to search the whole bare data, (d) we speed up computations since we can operate directly on the compressed data, which means the chances of fitting the whole structure in RAM memory or even into cache increase, and hence the processing time shortens, and (e) we save energy.

However, compressed data structures are much younger than classical compressed representations, which means we actually have more open problems than solutions. We need more space and time efficient compressed data structures to solve indexed pattern matching, to deal with computational biology problems, to solve document retrieval problems, to improve search engines, to represent hierarchical information, to improve searches on XML, to improve communication networks, to improve the performance in cluster-based environments, to provide search capabilities in constrained-memory environments like mobile devices or sensors, to improve geographic information systems, and a long etcetera.

In this thesis we face some of these problems having in mind the nature of data we are processing, distinguishing between *statistically-compressible* and *highly repetitive* datasets.

Statistical compressors take advantage of the distribution of symbols they are compressing to assign shorter codes to more frequent symbols. It is a very robust research area, since it has been active for many decades, and we know many lower bounds that tell us how far can we go. However, and despite of its maturity, there still

exist many problems not yet solved or not even considered. Statistical compression was initially thought for sequences which came from an infinite source of information and in which the number of different symbols (the alphabet) was finite. And not only finite, but in many cases considered small. However, with the advent of compressed data structures, the sources of information turned out to be finite (a document, for instance), and the alphabet could not be considered always small, becoming a problem in many cases due to its size. For instance, if we consider a text as a sequence of words and not of letters, then the alphabet size increases dramatically. If we model an $n \times n$ grid of points as a sequence of coordinates, then the alphabet size may become as large as the sequence. Adjacency lists in information retrieval systems or Web graphs usually display the same problem. In all these scenarios, just representing the alphabet and its statistical model is a serious problem, and how to deal with that has not been considered yet. The first part of this thesis focuses on proposing new compressed data structures that deal with statistically compressible datasets in which the alphabet is very large.

On the other hand, a *highly repetitive* dataset is that in which we have many copies or near-copies of the same document. This happens, for instance, in software repositories or versioning systems, in which we expect many versions of the same document, but with only slight modifications between successive versions. DNA databases are another example, since we have to store many genomes of individuals of the same species, being known that two individuals of the same species share a large portion of their genetic material, which results in highly repetitive databases. The problem is that statistical compressors do not perform well on highly repetitive datasets since they are not able of capturing repetitiveness. Many compressed representations targeted at highly-repetitive datasets have been proposed, but the number of compressed data structures is much more limited, and not many lower bounds are known. An obvious explanation is that highly-repetitive datasets have only been possible due to the exponential increase in storage capacity we experienced along the last decade. That is, it is a very young research area, and then the number of proposals is very limited. Actually, most compressed data structures for highly repetitive datasets are either not practical or, those which are practical, are rather space and time inefficient compared to the data entropy. This prevents compressed data structures from being used in practice, resulting in that most systems are still using classical compressed representations when they have to deal with highly repetitive databases. This is a big limitation since, for instance, by using compressed representations, version control systems can just store the differences and retrieve any version, but performing searches on the versioned collections by using compressed data structures would be much more interesting, but at the same time, much more challenging. The second part of this thesis is devoted to the proposal of efficient and practical compressed data structures for highly repetitive databases. This, apart from all benefits already described, may open the door to new functionalities and applications not achievable with classical compressed representations.

1.2 Contributions

In this section we describe the specific contributions presented along the thesis. All of them relate to new practical compressed data structures for *statistically-compressible* and *highly repetitive* databases. Beyond the interest the solutions may have by themselves, since they address open research problems, we also provide a set of practical applications in which our proposals are of interest. These applications are real world problems which we evaluate and for which we quantify the magnitude of the impact of our succinct data structures. In short terms, our contributions may be summarized as follows:

1. The first contribution of this thesis addresses the problem of space-efficient representation of optimal and suboptimal prefix-free codes. We present several implementations of previous ideas to experimentally show they were not only interesting from a theoretical perspective, but they could also be practical after undergoing a proper algorithm engineering process. The main idea behind this contribution is based on a previous work [BNO12] in which we explored the use of alphabetic codes to represent large models compactly. Later, we showed [NO13] that a permutation-based representation of the original Huffman codes was more efficient. Then, we extended this work with a previous publication by Gagie et al. [GNN10] that addressed the representation of sub-optimal prefix-free codes. The result work appeared in the *IEEE Transactions on Information Theory* journal [GNNO15]. My contribution was the implementation, engineering, and evaluation of the optimal and suboptimal schemes.

2. Our second contribution is the *compressed wavelet matrix*, an alternative sequence representation for large alphabets that retains all the properties of compressed *wavelet trees* but is significantly faster in practice. This contribution relies on a previous work [CN12] in which the uncompressed *wavelet matrix* was proposed. However, it turns out that we cannot apply the encodings we use to obtain zero-order compression on wavelet trees to wavelet matrices due to the reordering of bits induced by the latter. Thus, we derive an alternative code assignment scheme based on the Kraft inequality that is also optimal and compatible with *wavelet matrices*. By doing so, in theory we obtain zero-order compression, and in practice we obtain a data structure that on large alphabets is space-time dominant over the other implementations of wavelet trees over large alphabets.

   My contribution was a new optimal encoding scheme for *compressed wavelet matrices*, the proof of its correctness, its implementation, and the experimental evaluation. This work was published in the *Information Systems* journal [CNO15].

3. Our third contribution is a data structure to represent highly repetitive sequences. Recent applications need to represent this kind of sequences but classical statistical compression has proven to be ineffective in terms of space, since it is not able of capturing repetitiveness. We therefore introduce two grammar-based representations for highly repetitive sequences. The first, which we dubbed `GCC`, from *Grammar Compression with Counters*, is tailored for sequences with small alphabets and matches optimal space bounds available in the state of the art while performing very well in practice. The second is combination of the `GCC` with alphabet-partition-like techniques that excels in practice when the alphabet size of the sequence is large.

   I have been the main contributor in this development, including its conception, implementation, engineering, and the experimental evaluation. This work was published in *Proc. of the* 21^{th} *International Symposium on String Processing and Information Retrieval (SPIRE)* [NO14b] and its journal version was submitted to the *Information Systems* journal [ONB15].

4. Our fourth contribution is the first *LZ77*-bounded sequence representation that solves `access`, `rank`, and `select` in $O\left(\log_r \frac{n \lg \sigma}{z \lg n}\right)$ time using $O\left(\sigma z r \lg n \log_r \frac{n \lg \sigma}{z \lg n}\right)$ bits of space, z being the number of phrases in an *LZ77*-parsing of an input string $S[1, n]$ over an alphabet $\Sigma = [1, \sigma]$. We can also obtain $O(1)$ time for `access`, `rank`, and `select` using $O(\sigma z n^{\epsilon})$ space ($\epsilon < 1$). We dubbed our solution `BT`, from *Block Tree*, and beyond its theoretical properties, we show it also excels in practice when applied on sequences with small alphabets.

   I contributed to both theoretical and practical aspects of the data structure, focusing mostly on turning the theoretical proposal into a practical and competitive data structure. This includes the whole process of implementation, engineering, and experimental evaluation. This work was published in *Proc. of the 2015 Data Compression Conference (DCC)* [BGG$^+$15].

5. Our fifth contribution relates with trees and highly repetitive scenarios. It turns out that in this context, and depending on the application, many tree isomorphisms may show up in the tree topology. These isomorphisms may be well exploited by grammar compression techniques, but no practical fully-functional solutions exploit them appropriately. Our contribution, which we dub `GCT`, from *Grammar Compressed Tree*, is the first implementation of a fully-functional grammar compressed tree topology. We present the algorithms to carry out the most common operations, as well as a fully experimental evaluation in which we show its practical performance.

   A similar idea was independently presented by Bille et al. [BLR$^+$11] in sketchy form, and recently presented with full details [BLR$^+$15]. While they aim

at obtaining good theoretical properties, our focus has been the practical performance.

I have been the main contributor in this development, including its conception, implementation, engineering, and experimental evaluation. This work was initially published in *Proc. of the* 13^{th} *International Symposium on Experimental Algorithms (SEA 2014)* [NO14a]. Then it was selected among the best papers of the conference to be invited to a special issue of *ACM Journal of Experimental Algorithmics*, where it was finally accepted for publication [NO15].

6. Finally, we explored several applications of the data structures presented along the thesis that may be of interest by themselves. All of them are real world problems for which our proposals may have an impact. Concretely, our contributions in this aspect are:

   (a) New algorithms to support orthogonal range queries on the *wavelet matrix*. We propose and evaluate these algorithms by comparing them with their wavelet tree version, showing we match their space but outperform them in query time.

   (b) An experimental evaluation of inverted indexes when they are simulated using `rank` and `select` succinct data structures. Classical inverted indexes typically outperform our proposal, although the potential functionality we can offer is richer than that of classical approaches.

   (c) New FM-Indexes for highly repetitive sequences that use our succinct data structures to represent sequences. We compare our proposals with the state of the art, showing that we generally obtain the most space-efficient implementation, but being slower than some current implementations.

   (d) An XPath query system tailored to highly repetitive inputs, which is basically a re-engineering of a well known system for regular inputs. We build a prototype and we show some preliminary experimental results that suggest that using our compressed data structures we can build very space efficient XPath systems for highly repetitive datasets.

   (e) The *Grammar Compressed Suffix Tree* (`GCST`), one of the most space and time efficient suffix trees for highly repetitive datasets. We provide a fully experimental evaluation comparing it with the most space efficient compressed suffix trees from the state of the art, showing that we obtain competitive space and time performance.

1.3 Structure of the Thesis

This thesis is organized in four parts: the first for statistically-based representations; the second for highly repetitive or repetition-based structures; the third for

applications of the data structures presented along the thesis; and the fourth for summarizing. Concretely,

Part I starts with Chapter 2, in which we describe basic data structures and algorithms we intensively use along the thesis. Chapter 3 presents our first contribution and shows how to efficiently represent optimal and sub-optimal prefix-free codes, providing also a full experimental evaluation of these proposals. Chapter 4 presents our second contribution, the *compressed wavelet matrix*, for which we provide an experimental evaluation comparing this proposal with the best state of the art competitors.

Part II starts with a Chapter 5, dedicated to previous concepts of interest on highly repetitive scenarios. Chapter 6 presents our third contribution, which are several data structures to support `rank`, `select`, and `access` on grammar compressed sequences. This chapter also includes an experimental evaluation that compares our proposals with the state of the art. Chapter 7, describes our fourth contribution, the first *LZ77*-bounded sequence representation. We provide the algorithms to construct the data structure and to support queries, as well as a theoretical analysis. We also provide an implementation of our proposals, which we compare with the best state of the art competitors. Chapter 8 presents our fifth contribution, explaining how to grammar compress tree topologies, describing the algorithms to support the expected functionalities, and providing an experimental evaluation of our proposal.

Part III contains several applications in which we can apply the data structures presented along the thesis. Concretely, Chapter 9 presents new orthogonal range query algorithms implemented on a *wavelet matrix* with an experimental evaluation of these algorithms. Chapter 10 shows how to use repetition-based and statistically-compressed sequence representations to simulate *inverted indexes*. Although the algorithms are not new, we use this chapter to show the space-time performance of our proposals when compared with real inverted index implementations. Chapter 11 shows how to build the most space-efficient self-index for highly repetitive sequences. We use our proposals to solve `rank` and `select` queries on highly repetitive collections and we provide an experimental comparison. Chapter 12 shows how to adapt an XPath system to support queries on highly repetitive scenarios using a fraction of the space of the original proposal. We also provide an experimental evaluation. Finally, Chapter 13, presents a new *compressed suffix tree* for highly repetitive inputs. We experimentally evaluate it, showing it is a worth-considering solution for highly repetitive inputs.

Part IV contains Chapter 14, which presents our conclusions about the contributions proposed in this thesis, addressing also future research lines derived from this work.

Appendix A enumerates the publications and other research results derived from this thesis.

Appendix B presents a summary of the thesis in Spanish.

Part I

Statistical Encodings

Chapter 2

Previous Concepts on Statistical Encodings

2.1 Model of Computation

We assume the word-RAM model of computation. This model supposes we have available a constant number of registers we can use to operate and to address the RAM memory. The registers are of $w = \Theta(\lg n)$ bits[1], usually meaning that we can address at least n bits of RAM. The RAM memory is split into contiguous words of length w bits. Each operation in a register, from arithmetic (+,-,*,/) to binary operations, as well as for memory accesses (reads and writes) can be carried out in $\Theta(1)$ time.

2.2 Information Theory, Entropy, and Encodings

Information Theory is a branch of Computer Science that deals with the quantification of information and how to use that measure to efficiently transmit messages through a communication channel. Shannon's work [SW49] settled the basis of the field, providing many useful concepts that are still used today.

Suppose we are given a source of information that emits symbols $x \in \mathcal{X}$ with probability $p(x)$ (or p_x). This can be mathematically modeled as a discrete random variable X that takes values in $\mathcal{X}$ with probability mass function $p(x) = Pr\{X = x\}$, $x \in \mathcal{X}$, and $\mathcal{X}$ being a countable set. Shannon [SW49] defined the concept of *entropy* (*Shannon-entropy*) as a function $\mathcal{H}(X)$ or just $\mathcal{H}$ that measures the uncertainty about X. This function should satisfy the following properties:

[1] We use $\lg x$ to denote $\log_2 x$.

1. "$\mathcal{H}$ should be continuous in the $p(x)$".
2. "If all the $p(x)$ are equal, $p(x) = \frac{1}{|\mathcal{X}|}$, then $\mathcal{H}$ should be a monotonic increasing function on $|\mathcal{X}|$".
3. "If a choice be broken down into two successive choices, the original $\mathcal{X}$ should be the weighted sum of the individual values of $\mathcal{H}$".

Shannon [SW49] also proved the only function with these properties is:

$$\mathcal{H}(X) = -\sum_{x \in \mathcal{X}} p(x) \lg p(x)$$

with $0 \lg 0 = 0$.

An encoding C for a random variable X is a function that maps symbols in $\mathcal{X}$ to $\mathcal{D}^*$ ($C : \mathcal{X} \to \mathcal{D}^*$). An element $C(x) \in \mathcal{D}^*$is the code of x according to the encoding C. A *fixed-length* encoding is that in which every assigned code has the same length: $|C(x)| = |C(y)|, \forall x \in X$ ($|C(x)|$ is the length in bits of code $C(x)$). On the other hand, a *variable-length* encoding removes that restriction, permitting codes with different lengths.

In both cases, we need C to be injective ($x \neq y \Rightarrow C(x) \neq C(y)$), otherwise we cannot decode a symbol univocally. This kind of encodings are also known as *nonsingular*. However, this property suffices to decode a single symbol unambiguously, but not a concatenation of encoded symbols.

An *extension* C^* of an encoding C is a mapping from a sequence of symbols to its sequence of codes: $C^*(x_1, x_2, \ldots, x_n) = C(x_1)C(x_2)\ldots C(x_n)$. We say that an encoding C is *univocally decodable* if $C^*(x_1, x_2, \ldots, x_n)$ is *nonsingular*. Roughly speaking, an *univocally decodable* encoding means that from $C^*(x_1, x_2, \ldots, x_n)$ we can obtain back the original message $x_1, \ldots, x_n$. However, we may have to read the whole $C^*(x_1, x_2, \ldots, x_n)$ stream to obtain the code associated with x_1. In other words, it may be hard to tell where the code associated with a symbol finishes.

We say an encoding is *instantaneous*, *prefix-free*, or just a *prefix* encoding if we can determine x as soon as we have read the last bit of $C(x)$. This necessarily means there is no code $C(x)$ that is a prefix of another code $C(x')$. Note also that if C is *prefix-free*, then $C^*(x_1, x_2, \ldots, x_n)$ is *univocally decodable*. It is worth mentioning that for all *univocally decodable* code we can always find a *prefix-pree* code with the same average code length but being easier to decode. This is why *prefix-free* codes are preferred over just *univocally decodable* codes. Figure 2.1 depicts the relation between these classes of encodings.

The aim of many transmission algorithms is to find an *encoding* C such that (a) the *average code length* is minimized, and (b) C is *prefix-free*. The first requirement (a) means that if we are paying a price for each transmitted bit, we assign codes to symbols in order to minimize the cost of transmission. Being X the source of information, the function to minimize is:

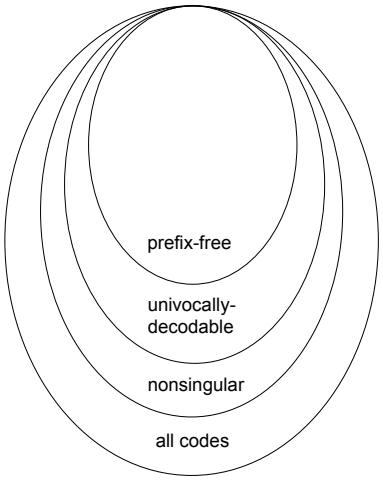


Figure 2.1: Relation between classes of encodings.

$$\mathcal{L}(X, C) = \sum_{x \in \mathcal{X}} p(x) \cdot |C(x)|$$

$\mathcal{L}(X, C)$ is known as the *average code length* for X according to the encoding C, satisfying that $\mathcal{L}(X, C) \geq \mathcal{H}(X)$.[2] Actually, $\mathcal{H}(X)$ is a lower bound on the average code length for any *univocally decodable* code. Additionally, to achieve the second requirement (b), the encoding C must also be prefix-free. Any prefix-free code must satisfy the *Kraft-McMillan inequality*:

$$\sum_{x \in \mathcal{X}} 2^{-|C(x)|} \leq 1.$$

Another important aspect about *encodings* is how to efficiently store and access them. The *encoding model* is a data structure that stores an encoding in such a way that (a) given a symbol x, we can obtain its code $C(x)$, and (b) given a code $C(x)$, we obtain back x. The size of the model, as well as the time necessary to access it, may be as important as other properties of the encoding.

2.3 Empirical Entropy

If the source of information is not an infinite source but just a message $S[1, n]$ drawn over $\Sigma = [1, \sigma]$, then $p(x)$ can be redefined as the probability of occurrence of x in S. Then, the *Shannon-entropy* is redefined as the *zero-order empirical entropy*:

[2] We omit the arguments of $\mathcal{L}$ if they can be inferred unambiguously from the context.

$$\mathcal{H}_0(S) = \sum_{x \in \Sigma} p(x) \lg \frac{1}{p(x)}$$

$\mathcal{H}_0$ is also a function that measures the uncertainty about S when considering only the probability of occurrence of each symbol. Note that in the worst-case $p(x) = p(y) \forall x, y \in \Sigma$, and $\mathcal{H}_0(S) = \lg \sigma$, which is reached when using fixed length encoding of $\lfloor \lg(\sigma) \rfloor + 1$ bits.

Although obtaining H_0 bits per symbol may be low enough in many cases, there exist applications in which we can go further. Higher-Order models measure the information or uncertainty about a symbol by considering which symbols precede it in the sequence. This is a reasonable approach, for instance, when modeling natural language sequences, where symbols group together to form words, being some words more frequent than others, and hence, the chance of predicting the next symbol of a word given the k preceding symbols is also higher. This intuitive idea is known as *k-order empirical entropy* of an input sequence S and is mathematically defined as:

$$\mathcal{H}_k(S) = \sum_{C = s_1 \ldots s_k} \frac{|S_C|}{n} \mathcal{H}_0(S_C)$$

where S_C is a string formed by collecting k symbols that follows each occurrence of the context $C = s_1 \ldots s_k$ in S.

2.4 Entropy Bounded Encodings

Given a random variable X taking values in $\Sigma = [1, \sigma]$, in this section we will show how to obtain prefix-free encodings space-bounded by the *Shannon* or *empirical* entropy of X. The algorithms we present here assign a code with an integral number of bits to each symbol (we cannot assign fractions of bits unless we use more sophisticated approaches like Arithmetic encodings [Sal07]). We focus in two algorithms: *Shannon-Fano* and *Huffman*.

2.4.1 Shannon-Fano Encoding

A Shannon-Fano encoding (SF) uses the *entropy* definition $\mathcal{H}(X) = \sum_\sigma p(x) \lg \frac{1}{p(x)}$ to obtain the code for each symbol in Σ. It turns out that the optimal code length for x is exactly $\lg(1/p(x))$ [SW49]. Then, using code lengths $\lceil \lg(1/p(x)) \rceil$ yields an average code length which uses at most 1 bit over the entropy $\mathcal{H}(X)$:

$$\mathcal{L}(X, SF) = \sum_{x \in \mathcal{X}} p(x) \left\lceil \lg \frac{1}{p(x)} \right\rceil < \sum_{x \in \mathcal{X}} p(x) (\lg \frac{1}{p(x)} + 1) = \mathcal{H}(\mathcal{X}) + 1.$$

Note that this way of assigning code lengths to symbols satisfies the *Kraft-McMillan inequality*, which ensures that a prefix-free encoding with that properties exists:

$$\sum_{x\in\mathcal{X}} 2^{-|C(x)|} = \sum_{x\in\mathcal{X}} 2^{-\left\lceil \lg \frac{1}{p(x)} \right\rceil} \leq \sum_{x\in\mathcal{X}} 2^{-\lg \frac{1}{p(x)}} = \sum_{x\in\mathcal{X}} p(x) = 1.$$

Note also that if the source of information is a sequence and not an infinite source, $\mathcal{H}$ becomes $\mathcal{H}_0$. This implies that the minimum probability of occurrence of a symbol x is $p(x) \geq 1/n$, and then the maximum length of a code is $\left\lceil \lg \frac{1}{p(x)} \right\rceil = O(\lg n)$.

Even though a *Shannon-Fano* encoding obtains an average code length which is less than 1 bit over the entropy $\mathcal{H}(X)$, it may be actually far from the entropy (concretely up to 2 bits [Hor77]). For instance, suppose we are given a random variable $X = \{a, b\}$ with mass probability function $P(X = a) = 0.999999$ and $P(x = b) = 1 - P(x = a)$. Then, $|C(a)| = \left\lceil \lg \frac{1}{P(x=a)} \right\rceil = 1$ and $|C(b)| = \left\lceil \lg \frac{1}{P(x=b)} \right\rceil = 20$, while the optimal code length for b is clearly 1 bit. This is an intrinsic problem of *Shannon-Fano* encodings, which was later overcomed by David Huffman[Huf52].

2.4.2 Huffman Encoding

The *Huffman* algorithm [Huf52] obtains a *prefix-free* encoding H for X such that $\mathcal{L}(X, H) = \sum_{x\in\Sigma} p(x)|C(x)|$ is minimal among all *prefix-free* codes. Besides, this encoding satisfies $\mathcal{H}(X) \leq \mathcal{L}(X, H) < \mathcal{H}(X) + 1$, which, in the worst case, is one bit less over $\mathcal{H}(X)$ than *Shannon-Fano* encodings.

Huffman's algorithm [Huf52] builds what is known as the Huffman tree to assign optimal codes to symbols. It is a $|\mathcal{D}|$-ary tree in which every node (but possibly the root) has $|\mathcal{D}|$ children. Each leaf node corresponds to a symbol and has associated its probability of occurrence. Each internal node of the tree has associated the sum of probabilities of all the leaves it covers. By construction, the Huffman algorithm ensures that if a leaf is less deep than another in the tree, then the probability of occurrence of the former is higher or equal than the second. Every child pointer of each node is labeled with a different symbol in $[0, |\mathcal{D}| - 1]$.

Once built, the Huffman tree permits to obtain the code associated with each symbol by traversing the tree from the root to the leaf that contains the symbol, annotating the child label we follow at each node. The code length associated with a symbol is given by the tree depth of the leaf representing the symbol. Figure 2.2 shows an example of binary Huffman tree for a given set of symbols and probabilities of occurrence.

Although there exist many algorithms to build a Huffman tree efficiently [MK95], Algorithm 1 shows probably one of the simplest implementations. It builds a $\mathcal{D}$-ary Huffman tree from a given $P\langle p_1, \ldots, p_\sigma\rangle$ using $O(\sigma)$ additional space. It runs in $O(\sigma)$ time if P is sorted (in increasing order by p_x to make Algorithm 1 correct),

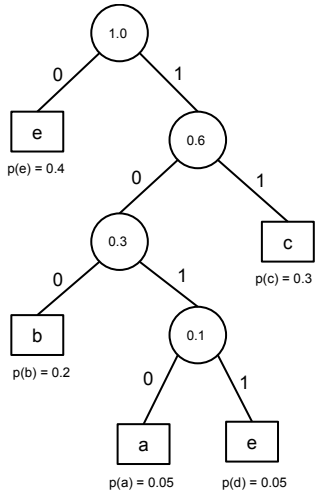


Figure 2.2: Example of a Huffman tree.

otherwise in $O(\sigma \lg \sigma)$ since we need to sort P. The algorithm proceeds as follows. Being Q_1 and Q_2 two initially empty queues, it first creates the tree leaves and inserts them into queue Q_1 in the same order they are in P. Then, it extracts from Q_1 a total of $2 \leq r' \leq \mathcal{D}$ nodes such that $(\sigma - r')/(\mathcal{D} - 1)$ is an integer number [Huf52], merges these nodes, and inserts the resulting node in the initially empty queue Q_2. This step is to ensure the next loop can always carry out merges of $|\mathcal{D}|$ nodes. After this step, each iteration of the *while* loop merges those $|\mathcal{D}|$ nodes with lower probability p_x from Q_1 and Q_2, inserting the resulting node (returned by function **MergeNodes**) into Q_2. This process is repeated until $|Q_1| + |Q_2| = 1$, ending with Q_2 containing the root of the Huffman tree.

2.4.2.1 Canonical Huffman

The average code length $\mathcal{L}$ of a Huffman encoding is determined by the depth of each symbol in the Huffman tree. We could actually permute symbols which are at the same depth still obtaining the same value for $\mathcal{L}$, as Figure 2.3 shows. This means that once code lengths are known, codes can be assigned to symbols in several ways. Among all of them, *canonical* Huffman [SK64, Sal07] is of special interest due to its many advantages[SK64, Sal07]. Basically, a *canonical* Huffman encoding ensures that codes with the same length are consecutive numbers, as we can see in Figure 2.4.

The algorithm to compute a D-ary canonical Huffman code [SK64] starts from the code length assignments produced by the standard Huffman algorithm (for the same arity), and produces a particular Huffman tree with the same code lengths.

Algorithm 1 Given the probability of occurrence of each symbol ($P\langle p_1, p_2, \ldots, p_\sigma\rangle$)) in increasing order and $\sigma \geq 2$, it returns a $\mathcal{D}$-ary Huffman tree [Huf52] associated with that distribution. Q_1 and Q_2 are two queues which can **Push** and **Pop** in $O(1)$ time. Function **PopMin**(Q_1, Q_2) **Pop**s the element with minimum probability p_x among Q_1 and Q_2.

Huffman$(P\langle p_1, p_2, \ldots, p_\sigma\rangle)$
  $Q_1, Q_2 \leftarrow \emptyset$
  for $x \in [1, \sigma]$ **do**
    $n_x.child[i] = null, \forall i \in [1, |\mathcal{D}|]$
    $n_x.p_x \leftarrow P[x]$
    $n_x.id \leftarrow x$
    Q_1.**Push**(n_x)
  end for
  Let $2 \leq r' \leq \mathcal{D}$ s.t. $(\sigma - r')/(\mathcal{D}-1)$ be an integer
  $n_{new} \leftarrow$ **MergeNodes**(Q_1, Q_2, r')
  Q_2.**Push**(n_{new})
  while $|Q_1| + |Q_2| > 1$ **do**
    $n_{new} \leftarrow$ **MergeNodes**$(Q_1, Q_2, |\mathcal{D}|)$
    Q_2.**Push**(n_{new})
  end while
  return Q_2.**Pop**()

MergeNodes(Q_1, Q_2, r)
  $n_{new}.p_x \leftarrow 0$
  for $i \in [1, r]$ **do**
    $v \leftarrow$ **PopMin**(Q_1, Q_2)
    $n_{new}.p_x \leftarrow n_{new}.p_x + v.p_x$
    $n_{new}.child[i] = v$
  end for
  $n_{new}.child[i] = null, \forall i \in [r+1, |\mathcal{D}|]$
  return n_{new}

First, it computes ℓ_{min} and ℓ_{max}, the minimum and maximum code lengths, and array $nCodes[\ell_{min}, \ell_{max}]$, where $nCodes[\ell]$ is the number of codes of length ℓ. Then, the algorithm assigns the codes as follows:

1. $first[\ell_{min}] = 0^{\ell_{min}}$ (i.e., ℓ_{min} 0s) is the first code of length ℓ_{min}.
2. All the codes of a given length ℓ are consecutive numbers, from $first[\ell]$ to $last[\ell] = first[\ell] + nCodes[\ell] - 1$.
3. The first code of the next length $\ell' > \ell$ that has $nCodes[\ell'] > 0$ is $|\mathcal{D}|^{\ell'-\ell} + (last[\ell] + 1)$.

Note that rule 2 ensures that all codes of a given level are consecutive numbers starting at $first[\ell]$. Note also $last[\ell]$ is the last code associated with a symbol of length ℓ. Rule 3 guarantees that the set of produced codes is prefix-free.

2.5 Encoding Schemes for Integers

Given a sequence $X[1, n]$ of integers, there exist a plenty of techniques that deal with how to compactly represent X while we access each element of X efficiently.

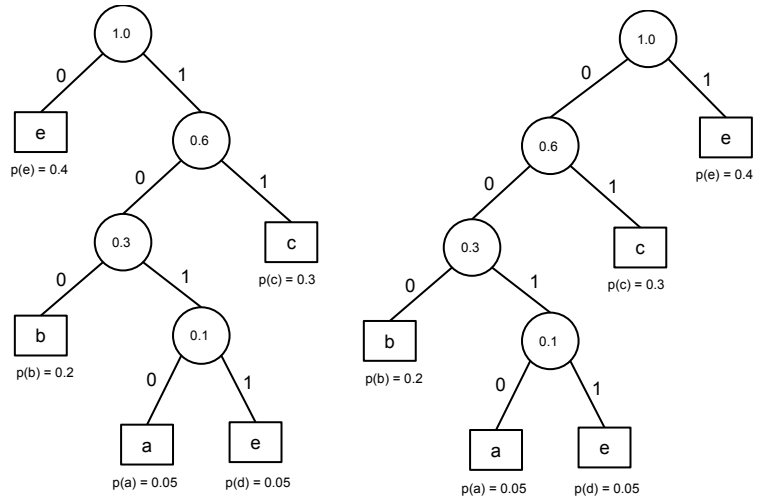
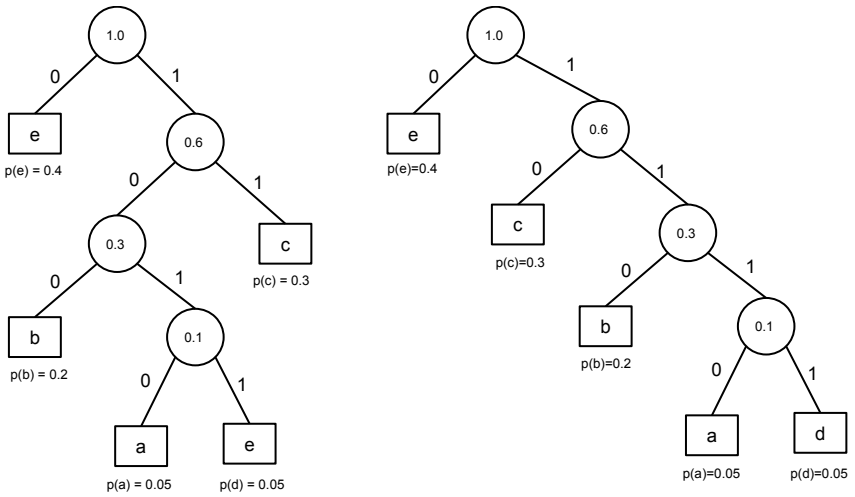


Figure 2.3: Example of two Huffman trees with the same average code length $\mathcal{L} = 2$.



(a) Huffman tree

(a) *Canonical* Huffman tree

Figure 2.4: Example of a Huffman tree (a) and an equivalent *canonical* Huffman tree (b).

Algorithm 2 Given a Huffman Tree T resulting from Algorithm 1, it reports triplets of *(symbol,code,code-length)*. It initially calls **ReportCodesHuffman**$(T, 0, 0)$.

ReportCodesHuffman$(t, code, len)$
  if isLeaf(t) **then**
    output$(t.id, code, len)$
  else
    for $i \in [1, |\mathcal{D}|]$ **and** $t.child[i] \neq null$ **do**
      ReportCodesHuffman$(t.child[i], code * |\mathcal{D}| + i, len + 1)$
    end for
  end if

Using a Huffman encoding for X may reduce the space from n words to $O(n\mathcal{H}_0(X))$ bits but it requires to store and use the *Huffman model*, which implies space and time overheads. On the other hand, we may focus in other properties of X rather than the *entropy* to obtain an space efficient representation. For instance, if X contains only small numbers, we can use an instantaneous encoding of $\lfloor \lg M \rfloor + 1$ bits for each number ($M = \max_{X[i]}(X)$), avoiding the use of any *model*.

In this section we address several instantaneous encodings, some of them particularly focused on representing small numbers like Unary-Codes, Gamma-Codes, and Delta-Codes. We also target an integer representation, dubbed VByte[WZ99], that allows us to space-efficiently represent sequences of larger integers.

2.5.1 Encodings for Small Integers

Some of the most representative techniques for small integers are unary-codes, γ-codes, and δ-codes. Table 2.1 shows how these techniques, which are explained next, encode the numbers in the interval $[1, 10]$.

Unary-codes are a kind of variable length encoding that represents an integer x as a binary sequence $unary(x) = 0^{x-1}1$, that is, a 0 repeated $x - 1$ times followed by a 1. Note $|unary(x)| = x$ and that this encoding does not permit to represent the 0 value. Besides, it is only adequate when x is very small.

Gamma-codes (γ-codes) is only adequate for small numbers. Given an integer x, its γ-code corresponds to the concatenation of the number of bits of x using unary codes and the number x without its most significant bit. That is, $\gamma(x) = unary(|x|)x\langle|x| - 1, 1\rangle$, where $x\langle|x| - 1, 1\rangle$ are the $|x| - 1$ least significant bits of x.

Delta-codes (δ-codes) become of interest when an integer x cannot be represented efficiently by a γ-encoding because it is too large. It is defined as follows: $\delta(x) = \gamma(|x|)x\langle|x| - 1, 1\rangle$. That is, we replace the unary-encoding used by γ-codes to tell the length of the integer by a γ-encoding.

Symbol	Unary-code	γ-code	δ-code
1	0	0	0
2	10	100	1000
3	110	101	1001
4	1110	11000	10100
5	11110	11001	10101
6	111110	11010	10110
7	1111110	11011	10111
8	11111110	1110000	11000000
9	111111110	1110001	11000001
10	1111111110	1110010	11000010

Table 2.1: Examples of variable length encodings for integers in the range $[1, 10]$.

2.5.2 Encodings for Large Integers

Although δ-codes become more efficient as the magnitude of the number grows, there exist some techniques that are specifically designed for large integers. These techniques, that are completely unacceptable for small integers, have some properties, like space efficiency and fast decoding, that make them more suitable for this new scenario. A particularly useful technique is known as VByte-codes [WZ99].

A *Variable Byte* [WZ99] is a byte-aligned integer representation addressed for larger numbers. The aim of this encoding is not only to be space efficient but also to obtain fast decoding by obtaining a byte-aligned variable length solution. The key point is to split each element of X into an integer number of *byte*-length chunks.

Thus, given an integer x, a VByte-encoding divides x into chunks of 7 bits. Each chunk is stored in a byte setting the most significant bit to 0 or 1 depending on whether the chunk is the least significant or not (respectively). Thus, $VByte(x) = b_1 b_2 \dots b_k$, where $k = \lceil |x|/7 \rceil$, and each $b_i = x\langle i * 7, (i-1) * 7 + 1 \rangle$, padding with 0 to the left if necessary, and setting $b_i\langle 8 \rangle = 1$ iff $i = k$.

Note finally that the extension of a VByte encoding is $VByte^*(X) = VByte(x_1)VByte(x_2)\dots VByte(x_n)$.

2.6 Rank, Select, and Access on Bitmaps

A `bitmap` or `bitsequence` is a sequence $B[1, n]$ where $B[i] \in \{0, 1\}$, $1 \leq i \leq n$, which supports the following operations:

- `access`(B, i) returns $B[i]$ for $1 \leq i \leq n$.

- $\texttt{rank}_v(B, i)$ reports the number of occurrences of $v \in \{0, 1\}$ in $B[1, i]$, with $1 \leq i \leq n$, $\texttt{rank}_v(B, 0) = 0$, and $\texttt{rank}_v(B, i) = \texttt{rank}_v(B, n)$ if $i > n$.

- $\texttt{select}_v(B, i)$ reports the position of the ith occurrence of $v = \{0, 1\}$ in $B[1, n]$, being $1 \leq i \leq \texttt{rank}_v(B, i)$ and $\texttt{select}_v(B, 0) = 0$.

We typically denote solutions that support these three operations as `rsa` data structures (from `rank`, `select`, and `access`).

The number of applications in which the use of bitmaps is involved is as heterogeneous as countless. From full-text indexes to succinct tree representations, the nature of applications is so variable that different representations with different space-time tradeoffs become necessary. A subset of these solutions are intensively used along this thesis and explained next.

The first proposal that addressed the problem of computing `rsa` queries on bitmaps was due to Jacobson[Jac89] and was initially developed as a tool to space-efficiently represent trees and graphs. Given a binary sequence $B[1, n]$, he proposed to add a sublinear space term on top of B to support `rank` queries in $O(1)$ time, becoming the total space for the whole structure $n+o(n)$ bits. This data structure on top of B basically consists of a two-level directory built as follows. We define a super-block size $s = \lg^2 n/2$ and we store an array $R_b[i] = \texttt{rank}_1(B, i * s)$, $1 \leq i \leq n/s$. This adds a total of $(n/s) \lg n = O(n/\lg n) = o(n)$ bits of overhead and permits to solve `rank` queries in $O(s)$ time. To obtain $O(1)$ `rank`, we need to add another level of sampling. We divide each super-block into blocks of length $b = s/\lg n = \lg n/2$ and we store a vector $R_b[i] = \texttt{rank}_1(B, b * i) - \texttt{rank}_1(B, \lceil i/s \rceil\, s)$, $1 \leq i \leq n/b$. The space for R_b adds to $(n/b)\lg(s + 1) = O(n \lg\lg n/\lg n) = o(n)$ bits. With this two-level data structure we can solve `rank` in $O(b)$ time since we can compute in $O(1)$ the `rank` answer for each block beginning but we still need to spend $O(b)$ time to scan the block itself. To overcome this problem we use a lookup-table that stores the `rank` answers for all possible chunks of length b. This lookup-table adds up $O(2^b b \lg b) = O(\sqrt{n} \lg n \lg\lg n) = o(n)$ bits. Therefore, the total space of this bitmap representation is n bits (the binary sequence itself) plus the space to store R_s, R_b, and the lookup-table, adding up for a total of $n + o(n)$ bits.

Later, Clark and Munro [Cla96, Mun96] augmented this proposal to also support `select` queries in $O(1)$ using an additional overhead of $O(n/\lg\lg n) = o(n)$ bits of space. Along the thesis, we refer to plain bitmap representations as `CM`.

All these data structures for bitmaps seen so far have in common that they use additional data structures on top of B to solve `rsa` queries efficiently. That is, all of them are orthogonal to the compressibility properties of B. Pagh [Pag01] and later Raman et al.[RRR07] were the first to explore this line and the first to statistically compress a binary sequence still solving `rsa` queries efficiently.

Concretely, the proposal Raman et al. [RRR07] suggests to conceptually divide the binary sequence into blocks of size b bits. Each block B_i covers from positions $(i-1)b+1$ to ib. A block belongs to class c_i if it contains exactly c_i 1s. As each block

has length at most b, we have at most b different classes. However, as classes contain different numbers of 1s, each class c_i has at most $\binom{b}{c_i}$ different reshuffles of its bits. Each block B_i is identified as a pair (c_i, o_i), where c_i identifies its class while $0 \leq o_i \leq \binom{b}{c_i}$ identifies each of the $\binom{b}{c_i}$ reshuffles of that class. We then store a sequence $C[1, \lceil n/b \rceil]$ to store the array of classes each block belongs to. As each cell requires $\lceil \lg(b+1) \rceil$ bits, it adds up for a total of $\lceil n/b \rceil \lceil \lg(b+1) \rceil = (n/b) \lg b + O(n/b)$ bits. Additionally, we need a vector O to store each o_i component. Knowing that each o_i requires $\left\lceil \lg \binom{b}{c_i} \right\rceil$ bits, the total space for the sequence O becomes:

$$\sum_{i=1}^{\lceil n/b \rceil} \left\lceil \lg \binom{b}{c_i} \right\rceil < \sum_{i=1}^{\lceil n/b \rceil} \lg \binom{b}{c_i} + \lceil n/b \rceil =$$

$$\lg \prod_{i=1}^{\lceil n/b \rceil} \binom{b}{c_i} + \lceil n/b \rceil \leq \lg \binom{b}{m} + \lceil n/b \rceil \leq n\mathcal{H}_0(B) + \lceil n/b \rceil$$

where $m = \sum_{i=1}^{\lceil n/b \rceil} c_i$ [Pag01].

Additionally, we need to add several sublinear data structures to solve `rsa` queries in $O(1)$ time, obtaining a final space bound of $n\mathcal{H}_0(B) + o(n)$ bits and $O(1)$ time for each operation [RRR07]. This data structure is called `RRR` along this thesis.

As we showed along this section, in theory we can solve `rank` and `select` queries in $O(1)$ time using $o(n)$ extra bits. However, the space overhead for `select` is $O(n/\lg\lg n)$, larger than that of `rank`, which is $O(n \lg\lg n/\lg n)$. Golynski [Gol07] also obtained $O(1)$ `select` time using $O(n \lg\lg n/\lg n)$ bits, but the constant is too large to be practical. Therefore, and despite of existing solutions that obtain $O(1)$ time and $o(n)$ space in theory, in practice we typically use non-constant time variants for `select`. The result is that, commonly, `select` operations are slower than `rank`. Therefore, and in order to close this gap, several practical improvements were proposed along the past few years [NP12, GP14].

We also know due to Pătraşcu and Viola [PV10] that, if we want $O(1)$ query time, then the sublinear space overhead cannot be less than $O(n/polylog(n))$ bits. The problem is that even this space overhead may be too much in some cases. For instance, if $B[1, n]$ is a very sparse bitmap, that is, if the number of 1s (m) is $m << n$, then $\mathcal{H}_0(B)$ is very small compared with $o(n)$, which starts to dominate. Then, for very sparse bitmaps, there exist a proposal due to Okanohara and Sadakane [OS07] that achieves $\mathcal{H}_0(B) + O(m)$ bits and solves `rank` in $O(\lg(n/m))$ and `select` in $O(1)$ time. Alternatively, one can encode the differences between 1s or 0s positions with δ-codes (see Section 2.5) and add a sub-linear space overhead on top to solve `rsa` queries efficiently [KN13]. This last technique is called `DELTA` along this thesis.

2.7 Rank, Select and Access on Sequences

A *sequence* data structure $S[1,n]$ is a generalization of the bitmap concept in which each $S[i] \in \Sigma = [1,\sigma]$, $1 \leq i \leq n$. We also extend the meaning of the three basic operations a *bitmap* should support as follows:

- $\texttt{access}(S,i)$ returns $S[i]$ for $1 \leq i \leq n$.
- $\texttt{rank}_v(S,i)$ reports the number of occurrences of $v \in \Sigma$ in $S[1,i]$, with $1 \leq i \leq n$, $\texttt{rank}_v(B,0) = 0$, and $\texttt{rank}_v(B,i) = \texttt{rank}_v(B,n)$ if $i > n$.
- $\texttt{select}_v(S,i)$ reports the position of the ith occurrence of $v \in \Sigma$ in $S[1,n]$, being $1 \leq i \leq \texttt{rank}_v(S,i)$ and $\texttt{select}_v(S,0) = 0$.

Additionally, some sequence representations support $\texttt{extract}(S,i,j)$ operation, which returns $S[i,j], 1 \leq i \leq j \leq n$ more efficiently than with $j-i+1$ calls to `access`.

A naive implementation of a sequence consist of using a bitmap $B_i[1,n]$ for each symbol $i \in \Sigma$, setting $B_i[j] = 1$ iff $S[j] = i$. $\texttt{rank}_v(S,i)$ will become $\texttt{rank}_1(B_v,i)$ while $\texttt{select}_v(S,i) = \texttt{select}_1(B_v,i)$. Therefore, the total time to solve `rank` and `select` queries on S becomes that of solving the same operations on each B_i. The total space becomes $n\sigma(1+o(1))$ bits in case of using uncompressed bitmaps or $nH_0(S) + O(n)$ if the bitmaps are compressed with Okanohara and Sadakane's technique [OS07]. However, `access` operation is not well supported since it implies to check all bitmaps in the wost case. Hence, the worst-case performance for this operation $O(\sigma t_{access})$, being t_{access} the time to support `access` on B_i bitmaps. We dub this solution as `MATRIX`.

Although this solution may be of interest in some scenarios where the space is not a limitation and `access` is uncritical, most of them require a more efficient solution both in terms space and time for the three operations.

In the literature there exist a plenty of sequence representations with different space/time tradeoffs. Some of them are explained next.

2.7.1 Wavelet Trees

A wavelet tree (`WT`) [GGV03] for sequence $S[1,n]$ over alphabet $[1..\sigma]$ is a complete balanced binary tree, where each node handles a range of symbols. The root handles $[1..\sigma]$ and each leaf handles one symbol. Each node v handling the range $[\alpha_v,\omega_v]$ represents the subsequence $S_v[1,n_v]$ of S formed by the symbols in $[\alpha_v,\omega_v]$, but it does not explicitly store S_v. Rather, internal nodes v store a bitmap $B_v[1,n_v]$, so that $B_v[i] = 0$ if $S_v[i] \leq \alpha_v + 2^{\lceil \lg(\omega_v - \alpha_v)\rceil - 1}$ and $B_v[i] = 1$ otherwise. That is, we partition the alphabet interval $[\alpha_v,\omega_v]$ into two roughly equal parts: a "left" one, $[\alpha_v, \alpha_v + 2^{\lceil \lg(\omega_v-\alpha_v)\rceil-1})$ and a "right" one, $[\alpha_v + 2^{\lceil \lg(\omega_v-\alpha_v)\rceil-1}, \omega_v]$. These are

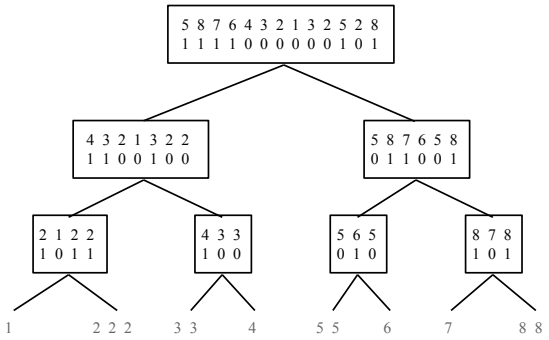


Figure 2.5: Example of a standard wavelet tree (`WT`) over a sequence.

handled by the left and right children of v. No bitmaps are stored for the leaves. Figure 2.5 gives an example.

The tree has height $\lceil \lg \sigma \rceil$, and it has exactly σ leaves and $\sigma - 1$ internal nodes. If we regard it level by level, we can see that it holds, in the B_v bitmaps, up to n bits per level (all the leaves in each level appear to the right of internal nodes). Thus it stores at most $n\lceil \lg \sigma \rceil$ bits. Storing the tree pointers, and pointers to the bitmaps, requires $O(\sigma \lg n)$ further bits, if we use the minimum of $\lg n$ bits for the pointers.

To access $S[i]$, we start from the root node ν, setting $i_\nu = i$. If $B_\nu[i_\nu] = 0$, this means that $S[i] = S_\nu[i_\nu] \leq 2^{\lceil \lg \sigma \rceil - 1}$ and that the symbol is represented in the subsequence S_{ν_l} of the left child ν_l of the root. Otherwise, $S_\nu[i_\nu] > 2^{\lceil \lg \sigma \rceil - 1}$ and it is represented in the subsequence S_{ν_r} of the right child ν_r of the root. In the first case, the position of $S_\nu[i_\nu]$ in S_{ν_l} is $i_{\nu_l} = \texttt{rank}_0(B_\nu, i_\nu)$, whereas in the second, the position in S_{ν_r} is $i_{\nu_r} = \texttt{rank}_1(B_\nu, i_\nu)$. We continue recursively, extracting $S_v[i_v]$ from node $v = \nu_l$ or $v = \nu_r$, until we arrive at a leaf representing the alphabet interval $[a, a]$, where we can finally report $S[i] = a$.

Therefore, the maximum cost of operation `access` is that of $\lceil \lg \sigma \rceil$ binary `rank` operations on bitmaps B_v.

The process to compute $\texttt{rank}_a(S, i)$ is similar. The difference is that we do not descend according to whether $B_v[i]$ equals 0 or 1, but rather according to the bits of $a \in [1, \sigma]$: the highest bit of a tells us whether to go left or right, and the lower bits are used in the next levels. When moving to a child u of v, we compute $i_u = \texttt{rank}_{0/1}(B_v, i_v)$ to be the number of times the current bit of a appears in $B_v[1, i_v]$. When we arrive at the leaf u handling the range $[a, a]$, the answer to $\texttt{rank}_a(S, i)$ is i_u.

Finally, to compute $\texttt{select}_a(S, j)$ we must proceed upwards. We start at the leaf u that handles the alphabet range $[a, a]$. So we want to track the position of $S_u[j_u]$, $j_u = j$, towards the root. If u is the left child of its parent v, then the

Algorithm 3 Standard wavelet tree algorithms: On the wavelet tree of sequence S rooted at ν, **acc**(ν, i) returns $S[i]$; **rnk**(ν, a, i) returns $\texttt{rank}_a(S, i)$; and **sel**(ν, a, j) returns $\texttt{select}_a(S, j)$. The left/right children of v are called v_l/v_r.

```
acc(v, i)
  if ω_v − α_v = 0 then
    return α_v
  end if
  if B_v[i] = 0 then
    i ← rank_0(B_v, i)
    return acc(v_l, i)
  else
    i ← rank_1(B_v, i)
    return acc(v_r, i)
  end if
```

```
rnk(v, a, i)
  if ω_v − α_v = 0 then
    return i
  end if
  if a ≤ 2^⌈lg(ω_v−α_v)⌉−1 then
    i ← rank_0(B_v, i)
    return rnk(v_l, a, i)
  else
    i ← rank_1(B_v, i)
    return rnk(v_r, a, i)
  end if
```

```
sel(v, a, j)
  if ω_v − α_v = 0 then
    return j
  end if
  if a ≤ 2^⌈lg(ω_v−α_v)⌉−1 then
    j ← sel(v_l, a, j)
    return select_0(B_v, j)
  else
    j ← sel(v_r, a, j)
    return select_1(B_v, j)
  end if
```

corresponding position at the parent is $S_v[j_v]$, where $j_v = \texttt{select}_0(B_v, j_u)$. Else, the corresponding position is $j_v = \texttt{select}_1(B_v, j_u)$. When we finally arrive at the root ν, the answer to the query is j_ν.

Thus the maximum cost of query $\texttt{rank}_a(S, i)$ is $\lceil \lg \sigma \rceil$ binary `rank` operations (just like $\texttt{access}(S, i)$), and the maximum cost of query $\texttt{select}_a(S, i)$ is $\lceil \lg \sigma \rceil$ binary `select` operations. Algorithm 3 gives the pseudocode (the recursive form is cleaner, but recursion can be easily removed).

Although when talking about wavelet trees we will focus on the binary case, it is worth mentioning that we can generalize the concept of `WT` to the multi-ary case: Instead of recursively dividing the vocabulary into two halves, we can split it into r disjoint sets. This is known as Multi-ary `WT` or `MWT`. Now the internal `MWT` nodes store sequences drawn over alphabet $[1, r]$ instead of bitmaps, and the height is reduced to $\lceil \log_r \sigma \rceil$. An example is shown in Figure 2.6 and the `rsa` algorithms can be easily modified to consider this new scenario.

2.7.2 Huffman-Shaped Wavelet Trees

Given the frequencies of the σ symbols in $S[1, n]$ and as explained in Section 2.4.2, the Huffman algorithm [Huf52] produces an optimal variable-length prefix-free encoding. If symbol $a \in [1, \sigma]$ appears n_a times in S, then the Huffman algorithm will assign it a codeword of length ℓ_a so that the sum $L = \sum_a n_a \ell_a$ is minimized. The output size of Huffman compression can be bounded by $\sum_a n_a \ell_a < n(\mathcal{H}_0(S) + 1)$ bits, which is off the optimum by less than 1 bit per symbol.

Building a balanced wavelet tree is equivalent to using a fixed length encoding. Instead, by giving the wavelet tree the shape of the Huffman tree, the total number of bits stored is exactly the output size of the Huffman compressor [GGV03, Nav12]: The leaf of a is at depth ℓ_a, and each of the n_a occurrences induces one bit in the

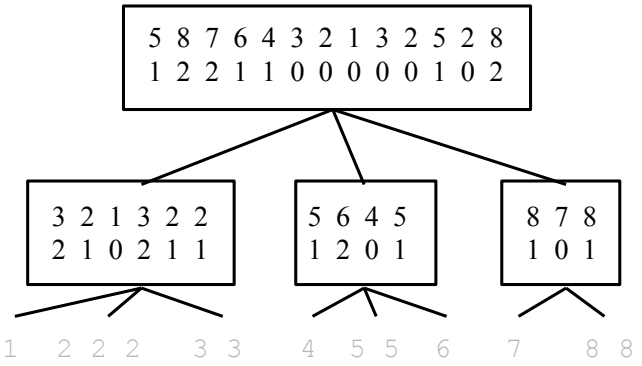


Figure 2.6: Multi-ary wavelet tree (`MWT`) with $r = 3$ corresponding to the running example.

bitmap of each of the ℓ_a ancestors of the leaf. The size of this tree, plus `rank`/`select` overheads, is thus upper bounded by $n(\mathcal{H}_0(S) + 1) + o(n(\mathcal{H}_0(S) + 1)) + O(\sigma \lg n)$ bits. Figure 2.7 (bottom) depicts a Huffman-shaped wavelet tree (`WTH`).

The wavelet tree operations are performed verbatim on Huffman-shaped wavelet trees. Moreover, they become faster on average: If $i \in [1, n]$ is chosen at random for $\texttt{access}(S, i)$, or a is chosen with probability n_a/n in operations $\texttt{rank}_a(S, i)$ and $\texttt{select}_a(S, j)$ (which is the typical case in most applications), then the average time is $O(\mathcal{H}_0(S) + 1)$. By rebalancing deep leaves, the space and average time are maintained and the worst-case time of the operations is limited to $O(\lg \sigma)$ [BN13].

Zero-order compression can also be achieved on the balanced wavelet tree, by using a compressed representation of the bitmaps [RRR07]. The time remains the same and the space decreases to $n\mathcal{H}_0(S) + o(n \lg \sigma)$ bits [GGV03]. Combining the compressed bitmap representation with Huffman shape, we obtain $n\mathcal{H}_0(S) + o(n(\mathcal{H}_0(S) + 1)) + O(\sigma \lg n)$ bits. This combination works well in practice [CN08], although the compressed bitmap representation is in practice slower than the plain one.

2.7.3 GMR Representation

Another solution that supports `rsa` queries on sequences is due to Golynski et al.[GMR06] (`GMR`). They present a data structure that takes $n \lg \sigma + o(\lg \sigma)n$ bits and answers `select` in $O(1)$ time, and `rank` and `access` in $O(\lg \lg \sigma)$ time. Although it does not compress the input, it is very fast both in theory and practice, being of interest in many applications. We do not need to explain their internal workings to present the results of this thesis.

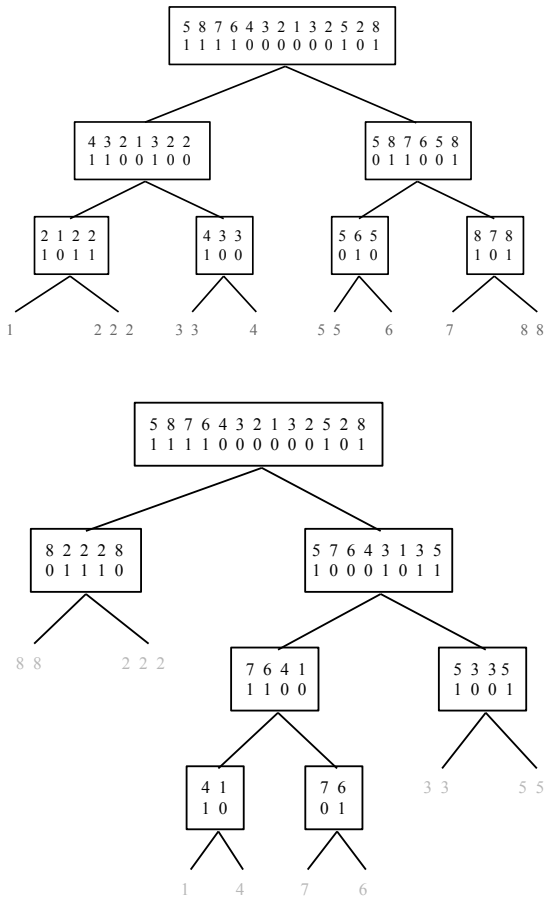


Figure 2.7: On the top, the same wavelet tree (`WT`) of Figure 2.5. On the bottom, its Huffman-shaped version (`WTH`).

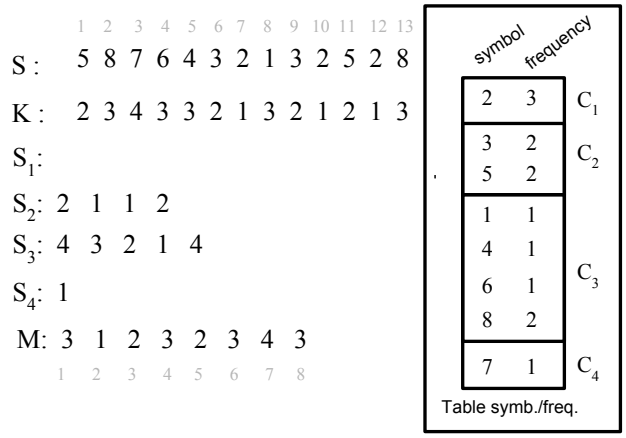


Figure 2.8: Alphabet Partitioning example.

2.7.4 Alphabet Partitioning

An alternative solution for `rsa` queries over large alphabets is *Alphabet Partitioning* (`AP`) [BCG+14], which obtains $nH_0(S) + o(n(H_0(S) + 1))$ bits and supports `rsa` operations in $O(\lg \lg \sigma)$ time. The main idea is to partition Σ into several subalphabets Σ_j, and S into the corresponding subsequences S_j, each defined over Σ_j (see Figure 2.8). The practical variant sorts the σ symbols by decreasing frequency and then splits that sequence into disjoint subsets, or subalphabets, of increasingly exponential size, so that Σ_j contains the 2^{j-1}th to the $(2^j - 1)$th most frequent symbols. The information on the partitioning is kept in a sequence M, where $M[i] = j$ iff $i \in \Sigma_j$. A new string $K[1, n]$ indicates the subalphabet each symbol of S belongs to: $K[i] = M[S[i]]$. Analogously to wavelet trees, the sequences S_j are defined as $S_j[i] = S[\texttt{select}_j(K, i)]$. Note that the number of subalphabets is at most $\lfloor \lg \sigma \rfloor + 1$, and this is the alphabet size of M and K. Therefore, a binary `WT` representation of M and K solves `rsa` operations in time $O(\lg \lg \sigma)$. Further, the symbols in each Σ_j are of roughly the same frequency, thus a fast compact (but not compressed) representation of S_j (`GMR`) (Section 2.7.3) yields $O(\lg \lg \sigma)$ time and does not ruin the statistical compression of S.

Algorithm 4 shows how the `rsa` operations on S translate into `rsa` operations on M, K, and on some subsequence S_j, thus obtaining $O(\lg \lg \sigma)$ times. In practice, the sequences S_j with the smallest alphabets are better integrated directly into the `WT` of K.

There are other representations that improve upon this solution in theory, but are unlikely to do better in practice. For example, it is possible to retain similar time complexities while reducing the space to $nH_k(S) + o(n \lg \sigma)$ bits, for any

Algorithm 4 Alphabet Partition algorithms for `access`, `rank`, and `select`

```
access(s, i)
  j ← K[i]
  v ← S_j[rank_j(K, i)]
  return select_j(M, v)

rank_a(S, i)
  j ← M[a]
  v ← rank_j(M, a)
  r ← rank_j(K, i)
  return rank_v(S_j, r)

select_a(S, i)
  j ← M[a]
  v ← rank_j(M, a)
  s ← select_v(S_j, i)
  return select_j(K, s)
```

$k = o(\log_\sigma n)$ [BHMR11, GOR10]. It is also possible, within zero-order entropy space, to solve `access` and `select` in $O(1)$ and any $\omega(1)$ time, or vice versa, and `rank` in time $O(\lg \frac{\lg \sigma}{w})$, on a RAM machine with word size w, which matches lower bounds [BN15].

2.8 Directly Addressable Codes

A *Directly Addressable Code* [BLN13] (DAC) is a variable-length encoding for integers of any magnitude, like VByte (see Section 2.5.2), but supporting direct access operations (`access`) efficiently.

Suppose we are given a sequence $X = x_1 \dots x_n$ of integers and a chunk length b. We divide each $x_i = X[i]$ into $\lceil \lfloor (\lg(x_i) \rfloor + 1)/b \rceil$ chunks. At the most significant position of each chunk we will append a 0 if that chunk is the most significant, or a 1 otherwise. Therefore, x_i becomes

$$b_{1,i} a_{1,i} b_{2,i} a_{2,i} \dots b_{j,i} a_{j,i}$$

where $b_{1,i}$ is the bit prepended to the chunk $a_{1,i} = x_i \langle i * b, i * (b-1) + 1 \rangle$, and $j = \lceil \lfloor (\lg(x_i) \rfloor + 1)/b \rceil$. Note that this is just a generalization of a VByte encoding if we set $b = 7$. However, the extension of a DAC encoding is completely different from that of VByte. Instead of concatenating the encoding of x_{i+1} after that of x_i, it builds a multi-layer data structure. At each layer $1 \leq l \leq \lceil \lfloor (\lg(x_i) \rfloor + 1)/b \rceil$, it concatenates all the lth chunks of all the numbers (that have one), doing the same with the bits prepended to each chunk. For instance, for level $l = 1$ we obtain a binary sequence B_1 and a sequence A_1 such that:

$$B_1 = b_{1,1} b_{1,2} \dots b_{1,n}$$

$$A_1 = a_{1,1} a_{1,2} \dots a_{1,n}$$

The next layer is built by concatenating the second chunk of each number (if there exists), repeating the process until $j = \lceil (\lfloor \lg(x_i) \rfloor + 1)/b \rceil$. Figure 2.9 shows an example of a DAC encoding for an integer sequence X. In order to efficiently retrieve any x_i, we must process each binary sequence B_i to support `rank` and `access` queries in $O(1)$ time.

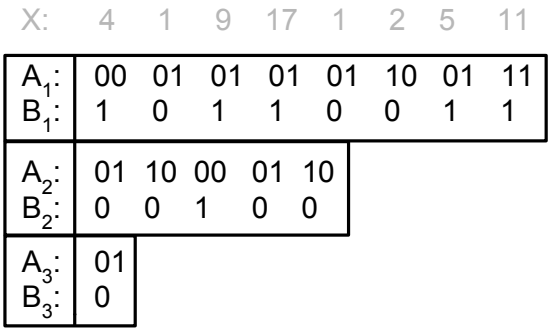


Figure 2.9: Example of a DAC for a sequence $X = 4, 1, 9, 17, 1, 2, 5, 11$ and $b = 2$.

Given a DAC encoding, access to $X[i]$ is carried out as follows. We start by setting $i_1 = i$, and reading $A_1[i_1] = a_{1,i_1}$. We set $res = A_1[i_1]$ and if $B_1[i_1] = 0$ we are done because this chunk is the last of x_i. If $B_1[i_1] = 1$, it means that x_i continues in the next layer. To compute the position of the next chunk in the next layer we set $i_2 = \texttt{rank}_1(B_1, i)$. In the second layer we concatenate $A_2[i_2]$ with the current result: $res = A_2[i_2]A_1[i_1]$ and then check $B_2[i_2]$, repeating the process until we get a $B_k[i_k] = 0$. As `rank` and `access` on each B_i are carried out in $O(1)$ time, and reading a chunk of length b takes $O(1)$ time, the total time to extract an element from X when represented with a DAC is worst-case $O(\lg(M)/b)$, being M the largest number in X.

A further optimization of DACs consists of permitting different chunk lengths for each layer. The authors presented a dynamic programming algorithm that computes the optimal chunk length, as well as the optimal number of layers, to obtain the most compact DAC for any given input X [BLN13].

2.9 Static Succinct Tree Representations

Trees are one of the most fundamental data structures in Computer Science. We may distinguish basically two types: *ordinal* and *cardinal* trees. An *ordinal* tree is a tree in which any node may have any number of children. For instance, an XML file is actually an *ordinal* tree.

On the other hand, in a *cardinal* tree both the number and the order of children is important. For instance, in a Binary Search Tree each node has at most two children that cannot be swapped. Even though both *ordinal* and *cardinal* trees have a wide number of applications, we will focus only on *ordinal* trees.

Given an *ordinal* tree $\mathcal{T}_n$ with n nodes, if we use machine pointers of length

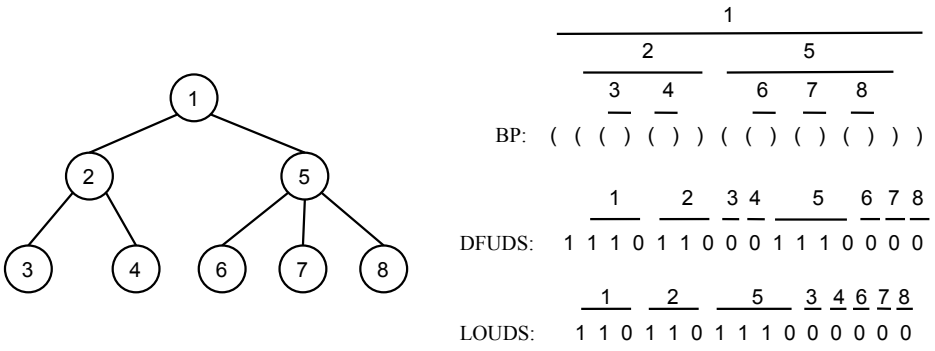


Figure 2.10: Ordinal succinct tree representations.

$\Theta(\lg n)$ bits, then we need $O(n \lg n)$ bits to store the tree topology. However, it is known that the number of different *ordinal* trees with n nodes is given by the $(n-1)th$ Catalan number, which is $C_{n-1} = \frac{1}{n}\binom{2n-1}{n-1}$. Thus, the minimum number of bits to represent any *ordinal* tree with n nodes is $\lfloor \lg(C_{n-1}) \rfloor + 1 = 2n - \Theta(\lg n)$ bits. That is, instead of $\lg n$ bits, just 2 per node suffice to store the tree topology.

The most well known techniques that achieve this bound are *BP* (Balanced Parentheses)[Jac89, MR01, NS14], *DFUDS* (Depth-First Unary Degree Sequence)[BDM$^+$05, JSS12], and *LOUDS* (Level Order Unary Degree Sequence)[Jac89, DRR06]. Figure 2.10 shows an example of the different representations.

BP [Jac89, MR01, NS14] represents the tree topology as a sequence of parentheses. It is built by traversing the tree in preorder, writing an opening parenthesis when we first arrive at a node, and a closing one when we leave its subtree. Thus a tree of n nodes is represented with $2n$ parentheses.

DFUDS [BDM$^+$05, JSS12] represents the tree topology as a sequence of bits. It is built by traversing the tree in depth-first order, annotating in unary the number of children of each node (for technical reasons, we add a 1 at the beginning of the sequence). Therefore, the number of 1s is equal to the number of nodes, which is also equal to the number of 0s. Thus, the total number of bits is $2n$.

LOUDS [Jac89, DRR06] represents the tree topology also as a sequence of bits. We traverse the tree in level-order annotating the number of children of each node we visit in unary (recall Section 2.5). Thus, the number of 1 in the resulting sequence is equal to $n-1$ since we have n nodes. The number of 0s is equal to the number of processed nodes, which is n. Thus, the final binary sequence contains a total of $2n-1$ bits.

The different formats allow us to solve different subsets of the desired tree operations (see Table 2.2), although the most functionally-richer solution is the

Name	Explanation
root()	The root of the tree.
fChild(v)	First child of node v.
lChild(v)	Last child of node v.
child(v, i)	ith child of v (if exists).
children(v)	Number of children of node v.
nSibling(v)	Next sibling of node v (if exists).
pSibling(v)	Previous sibling of node v (if exists).
parent(v)	Parent of node $v \neq$root().
isLeaf(v)	Whether v is a leaf.
depth(v)	Node depth of v.
height(v)	Height of a node v (distance from the root()).
subtree(v)	Number of nodes in the subtree of v.
isAncestor(u,v)	Whether u is an ancestor of v.
levelAncestor(v,i)	The ancestor at distance i from v (if exists).
lca(u,v)	Lowest Common Ancestor of u and v.
noderank(v)	A unique identifier in $[1, n]$ of node v.
nodeselect(id)	The node v with noderank(v)=id.

Table 2.2: List of operations in *ordinal* trees.

version of *BP* from Navarro and Sadakane [NS14], usually known as `FF` and described in Section 8.1. However, *LOUDS* is much easier to implement and suffices in many applications.

2.10 Text

Let $T[1, n]$ be a text (or the concatenation of the texts in a collection) over alphabet $\Sigma = [1, \sigma]$. The character at position i of T is denoted $T[i]$, whereas $T[i, j]$ denotes $T[i]T[i + 1] \dots T[j]$, a substring of T. $|T[i, j]| = j - i + 1$ is the length of the string $T[i, j]$. A *suffix* of T is a substring of the form $T[i, n]$ and a *prefix* of T is of the form $T[1, i]$. In many cases, and for convenience, we suppose $T[n] =$ \$, where \$ is a symbol lexicographically smaller than any other in T.

A *self-index* [FM05, GV06, NM07, Sad03] is a data structure that represents T and solves operations `count`(p), which returns the number of occurrences of a pattern p in T; `locate`(p), which reports the positions of the occurrences of p in T; and `extract`(i, j), which retrieves $T[i, j]$, without actually accessing T.

Chapter 3

Efficient Representation of Prefix Codes

Statistical compression is a well-established branch of Information Theory. Given a text T of length n, over an alphabet of σ symbols $\Sigma = \{a_1, \ldots, a_\sigma\}$ with relative frequencies $P = \langle p_1, \ldots, p_\sigma \rangle$ in T (where $\sum_{i=1}^{\sigma} p_i = 1$), the *empirical entropy* of the text (already defined in Section 2.3) is $\mathcal{H}_0(S) = \mathcal{H}_0(P) = \mathcal{H}(P) = \sum_{i=1}^{\sigma} p_i \lg(1/p_i)$. As it is known, Huffman encoding guarantees that the encoded text uses less than $n(\mathcal{H}(P)+1)$ bits. Arithmetic codes achieve less space, $n\mathcal{H}(P)+2$ bits, however they are not prefix-free, which complicates and slows down both encoding and decoding.

In terms of the *redundancy* of the code $\mathcal{L}(P) - \mathcal{H}(P)$, where $\mathcal{L}(P)$ is the *average code length* defined in Section 2.2, Huffman codes are optimal and the topic can be considered closed. How to store *the prefix code itself* or the *model*, however, is much less studied. It is not hard to store it using $\mathcal{O}(\sigma \lg \sigma)$ bits, and this is sufficient when σ is much smaller than n. There are several scenarios, however, where the storage of the code itself is problematic. One example is word-based compression, which is a standard to compress natural language text [BSTW86, Mof89]. Word-based Huffman compression not only performs very competitively, offering compression ratios around 25%, but also benefits direct access [WMB99], text searching [MNZB00], and indexing [BFLN12]. In this case the alphabet size σ is the number of distinct words in the text, which can reach many millions. Other scenarios where large alphabets arise are the compression of East Asian languages and general numeric sequences. Yet another case arises when the text is short, for example when it is cut into several pieces that are statistically compressed independently, for example for compression boosting [FGMS05, KP11] or for interactive communications or adaptive compression [BFNP07]. The more effectively the codes are stored, the finer-grained can the text be cut.

During encoding and decoding, the code must be maintained in main memory to

achieve reasonable efficiency, whereas the plain or the compressed text can be easily read or written in streaming mode. Therefore, the size of the code, and not that of the text, is what poses the main memory requirements for efficient compression and decompression. This is particularly stringent on mobile devices, for example, where the supply of main memory is comparatively short. With the modern trend of embedding small devices and sensors in all kinds of objects (e.g., the "Internet of Things"[1]), those low-memory scenarios may become common.

In this chapter we present various relevant results of theoretical and practical nature about how to store a code space-efficiently, while also considering the time efficiency of compression and decompression. The specific contributions of this chapter are the following.

1. In Section 3.2 we show that it is possible to store an optimal prefix code within $\mathcal{O}(\sigma \lg \ell_{\max})$ bits, where $\ell_{\max} = \mathcal{O}(\min(\sigma, \lg n))$ is the maximum length of a code (Theorem 1). Then we refine the space to $\mathcal{O}(\sigma \lg \lg(n/\sigma))$ bits (Corollary 1). Within this space, encoding and decoding are carried out in constant time on a RAM machine with word size $w = \Omega(\lg n)$. The result is obtained by using canonical Huffman codes [SK64], fast predecessor data structures [FW93, PT08] to find code lengths, and multiary wavelet trees [GGV03, FMMN07, BN15] to represent the mapping between codewords and symbols.

2. In Section 3.3 we show that, for any $0 < \epsilon < 1/2$, it takes $\mathcal{O}(\sigma \lg \lg(1/\epsilon))$ bits to store a prefix code with average codeword length at most $\mathcal{L}(P) + \epsilon$. Encoding and decoding can be carried out in constant time on a RAM machine with word size $w = \Omega(\lg n)$. Thus, if we can tolerate a small constant additive increase in the average codeword length, we can store a prefix code using only $\mathcal{O}(\sigma)$ bits. We obtain this result by building on the above scheme, where we use length-limited optimal prefix codes [ML01] with a carefully chosen $\ell_{\max}$ value.

3. In Section 3.4 we show that, for any constant $c > 1$, it takes $\mathcal{O}\big(\sigma^{1/c} \lg \sigma\big)$ bits to store a prefix code with average codeword length at most $c\,\mathcal{L}(P)$. Encoding and decoding can be carried out in constant time on a RAM machine with word size $w = \Omega(\lg \sigma)$. Thus, if we can tolerate a small constant multiplicative increase, we can store a prefix code in $o(\sigma)$ bits. To achieve this result, we only store the codes that are shorter than about $\ell_{\max}/c$, and use a simple code of length $\ell_{\max} + 1$ for the others. Then all but the shortest codewords need to be explicitly represented.

4. In Section 3.5 we engineer and implement all the schemes above and compare them with careful implementations of state-of-the-art optimal and suboptimal

[1] `http://en.wikipedia.org/wiki/Internet_of_Things`

codes. Our model representations are shown to use 6–8 times less space than classical ones, at the price of being several times slower for compression (2.5–8 times) and decompression (12–24 times). The additive approximations reduce these spaces up to a half and the times by 20%–30%, at the expense of a small increase (5%) in the redundancy. The multiplicative approximations can obtain models of the same size of the additive ones, yet increasing the redundancy to around 10%. In exchange, they are about as fast as the classical compression methods. If we allow them increase the redundancy to 15%–20%, the multiplicative approximations obtain model sizes that are orders of magnitude smaller than classical representations.

5. As a byproduct, Section 3.5 also compares varios heuristic, approximation, and exact algorithms to generate length-restricted prefix codes. The experiments show that the optimal algorithm is practical to implement and runs fast, while obtaining significantly better average code lengths than the heuristics and the approximations. A very simple-to-program approximation reaches the same optimal average code length in our experiments, yet it runs significantly slower.

3.1 Related Work

A simple pointer-based implementation of a Huffman tree takes $\mathcal{O}(\sigma \lg \sigma)$ bits, and it is not difficult to show this is an optimal upper bound for storing a prefix code with minimum average codeword length. For example, suppose we are given a permutation π over σ symbols. Let P be the probability distribution that assigns probability $p_{\pi(i)} = 1/2^i$ for $1 \leq i < \sigma$, and probability $p_{\pi(\sigma)} = 1/2^{\sigma-1}$. Since P is dyadic, every optimal prefix code assigns codewords of length $\ell_{\pi(i)} = i$, for $1 \leq i < \sigma$, and $\ell_{\pi(\sigma)} = \sigma - 1$. Therefore, given any optimal prefix code and a bit indicating whether $\pi(\sigma - 1) < \pi(\sigma)$, we can reconstruct π. Since there are $\sigma!$ choices for π, in the worst case it takes $\Omega(\lg \sigma!) = \Omega(\sigma \lg \sigma)$ bits to store an optimal prefix code.

Considering the argument above, it is natural to ask whether the same lower bound holds for probability distributions that are not so skewed, and the answer is no. A prefix code is *canonical* [SK64, MT97] if a shorter codeword is always lexicographically smaller than a longer codeword. Given any prefix code, we can always generate a canonical code with the same code lengths (recall Section 2.4.2.1). Moreover, we can reassign the codewords such that, if a symbol is lexicographically the jth with a codeword of length ℓ, then it is assigned the jth consecutive codeword of length ℓ. It is clear that it is sufficient to store the codeword length of each symbol to be able to reconstruct such a code, and thus the code can be represented in $\mathcal{O}(\sigma \lg \ell_{\max})$ bits.

There are more interesting upper bounds than $\ell_{\max} \leq \sigma$. Katona and Nemetz [KN76] (see also Buro [Bur93]) showed that, if a symbol has relative frequency p, then any Huffman code assigns it a codeword of length at most $\lfloor \log_\phi(1/p) \rfloor$, where

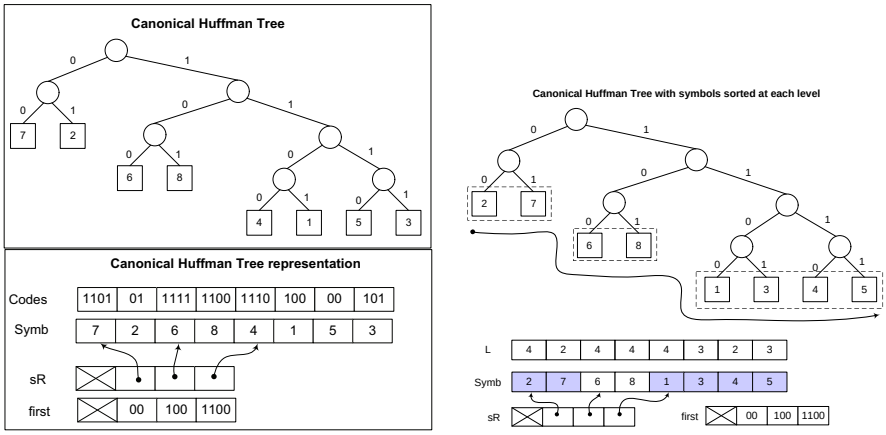
$\phi = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio, and thus $\ell_{\max}$ is at most $\lfloor \log_\phi(1/p_{\min}) \rfloor$, where $p_{\min}$ is the smallest relative frequency in P. Note also that, since $p_{\min} \geq 1/n$, it must hold $\ell_{\max} \leq \log_\phi n$, therefore the canonical code can be stored in $\mathcal{O}(\sigma \lg \lg n)$ bits.

Alternatively, one can enforce a value for $\ell_{\max}$ (which must be at least $\lceil \lg \sigma \rceil$) and pay a price in terms of average codeword length. The same bound above [KN76] hints at a way to achieve any desired $\ell_{\max}$ value: artificially increase the frequency of the least frequent symbols until the new $p_{\min}$ value is over $\phi^{-\ell_{\max}}$, and then an optimal prefix code built on the new frequencies will hold the given maximum code length. Another simple technique (see, e.g., [BN09], where it was used for Hu-Tucker codes) is to start with an optimal prefix code, and then spot all the highest nodes in the code tree with depth $\ell_{\max} - d$ and more than 2^d leaves, for any d. Then the subtrees of the parents of those nodes are made perfectly balanced. A more sophisticated technique, by Milidiú and Laber [ML01], yields a performance guarantee. It first builds a Huffman tree T_1, then removes all the subtrees rooted at depth greater than $\ell_{\max}$, builds a complete binary tree T_2 of height h whose leaves are those removed from T_1, finds the node $v \in T_1$ at depth $\ell_{\max} - h - 1$ whose subtree T_3's leaves correspond to the symbols with minimum total probability, and finally replaces v by a new node whose subtrees are T_2 and T_3. They show that the resulting average code length is at most $\mathcal{L}(P) + 1/\phi^{\ell_{\max} - \lceil \lg(\sigma + \lceil \lg \sigma \rceil - \ell_{\max}) \rceil - 1}$.

All these approximations require $\mathcal{O}(\sigma)$ time plus the time to build the Huffman tree. A technique to obtain the optimal length-restricted prefix code, by Larmore and Hirshberg [LH90], performs in $\mathcal{O}(\sigma\, \ell_{\max})$ time by reducing the construction to a binary version of the coin-collector's problem.

The above is an example of how an additive increase in the average codeword length may yield less space to represent the code itself. Another well-known additive approximation follows from Gilbert and Moore's proof [GM59] that we can build an alphabetic prefix code with average codeword length less than $\mathcal{H}(P) + 2$, and indeed no more than $\mathcal{L}(P) + 1$ [Nak91, She92]. In an alphabetic prefix code, the lexicographic order of the codewords is the same as that of the source symbols, so we need to store only the code tree and not the assignment of codewords to symbols. Any code tree, of $\sigma - 1$ internal nodes, can be encoded in $4\sigma + o(\sigma)$ bits so that it can be navigated in constant time per operation [DRS12], and thus encoding and decoding of any symbol takes time proportional to its codeword length.

Multiplicative approximations have the potential of yielding codes that can be represented within $o(\sigma)$ bits. Adler and Maggs [AM01] showed it generally takes more than $(9/40)\sigma^{1/(20c)} \lg \sigma$ bits to store a prefix code with average codeword length at most $cH(P)$. Gagie [Gag06a, Gag06b, Gag08] showed that, for any constant $c \geq 1$, it takes $\mathcal{O}\big(\sigma^{1/c} \lg \sigma\big)$ bits to store a prefix code with average codeword length at most $cH(P) + 2$. He also showed his upper bound is nearly optimal because, for any positive constant ϵ, we cannot always store a prefix code with average codeword length at most $cH(P) + o(\lg \sigma)$ in $\mathcal{O}\big(\sigma^{1/c - \epsilon}\big)$ bits. Note that our result does not



(a) (b)

Figure 3.1: An arbitrary canonical prefix code (a) and the result of sorting the source symbols at each level (b).

have the additive term "+2" in addition to the multiplicative term, which is very relevant on low-entropy texts.

3.2 Representing Optimal Codes

In Section 2.4.2 we have shown how to build a Huffman tree and how to extract the canonical Huffman codes from that tree (Figure 3.1(a) illustrates a canonical Huffman code). Suppose now we have an array Codes in which we had stored at position i the code c_i of source symbol a_i, using $\ell_{\max} = \mathcal{O}(\lg n)$ bits for each. For encoding in constant time, we can simply use an array like Codes. For decoding, the source symbols are written in an array Symb, in left-to-right order of the leaves. This array requires $\sigma \lg \sigma$ bits. The access to this array is done via two smaller arrays, which have one entry per level: $\mathsf{sR}[\ell]$ points to the first position of level ℓ in Symb, whereas $\mathsf{first}[\ell]$ stores the first code in level ℓ. The space for these two arrays is $\mathcal{O}(\ell_{\max}^2)$ bits.

Then, if we have to decode the first symbol encoded in a bitstream, we first have to determine its length ℓ. In our example, if the bitstream starts with 0, then $\ell = 2$; if it starts with 10, then $\ell = 3$, and otherwise $\ell = 4$. Once the level ℓ is found, we read the next ℓ bits of the stream in c_i, and decode the symbol as $a_i = \mathsf{Symb}[\mathsf{sR}[\ell] + c_i - \mathsf{first}[\ell]]$.

The problem of finding the appropriate entry in first can be recast into a predecessor search problem [GN09, KN09]. We extend all the values $\mathsf{first}[\ell]$ by

appending $\ell_{\max} - \ell$ bits at the end. In our example, the values become $\mathsf{0000} = 0$, $\mathsf{1000} = 8$, and $\mathsf{1100} = 12$. Now, we find the length ℓ of the next symbol by reading the first $\ell_{\max}$ bits from the stream, interpreting it as a binary number, and finding its predecessor value in the set. Since we have only $\ell_{\max} = \mathcal{O}(\lg n)$ numbers in the set, and each has $\ell_{\max} = \mathcal{O}(\lg n)$ bits, the predecessor search can be carried out in constant time using fusion trees [FW93] (see also Patrascu and Thorup [PT08]), within $\mathcal{O}(\ell_{\max}^2)$ bits of space.

Although the resulting structure allows constant-time encoding and decoding, its space usage is still $\mathcal{O}(\sigma\, \ell_{\max})$ bits. In order to reduce it to $\mathcal{O}(\sigma \lg \ell_{\max})$, we will use a *multiary wavelet tree* data structure (see also Section 2.7.1). In particular, we use the version that does not need universal tables [BN15, Thm. 7]. This structure represents a sequence $L[1, \sigma]$ over alphabet $[1, \ell_{\max}]$ using $\sigma \lg \ell_{\max} + o(\sigma \lg \ell_{\max})$ bits, and carries out the operations in time $\mathcal{O}(\lg \ell_{\max} / \lg w)$. In our case, where $\ell_{\max} = \mathcal{O}(w)$, the space is $\sigma \lg \ell_{\max} + o(\sigma)$ bits and the time is $\mathcal{O}(1)$. The operations supported by wavelet trees are also described in Section 2.7.

Assume that the symbols of the canonical Huffman tree are in increasing order within each depth, as in Figure 3.1(b).[2] Now, the key property is that $\mathsf{Codes}[i] = \mathsf{first}[\ell] + \mathtt{rank}_\ell(L, i) - 1$, where $\ell = L[i]$, which finds the code $c_i = \mathsf{Codes}[i]$ of a_i in constant time. The inverse property is useful for decoding code c_i of length ℓ: the symbol is $a_i = \mathsf{Symb}[\mathsf{sR}[\ell] + c_i - \mathsf{first}[\ell]] = \mathtt{select}_\ell(L, c_i - \mathsf{first}[\ell] + 1)$. Therefore, arrays Codes, Symb, and sR are not required; we can encode and decode in constant time using just the wavelet tree of L and first, plus its predecessor structure. This completes the result.

Theorem 1. *Let P be the frequency distribution over σ symbols for a text of length n, so that an optimal prefix code has maximum codeword length $\ell_{\max}$. Then, under the RAM model with computer word size $w = \Omega(\ell_{\max})$, we can store an optimal prefix code using $\sigma \lg \ell_{\max} + o(\sigma) + \mathcal{O}(\ell_{\max}^2)$ bits, note that $\ell_{\max} \le \log_\phi n$. Within this space, encoding and decoding any symbol takes $\mathcal{O}(1)$ time.*

Therefore, under mild assumptions, we can store an optimal code in $\mathcal{O}(\sigma \lg \lg n)$ bits, with constant-time encoding and decoding operations. In the next section we refine this result further. On the other hand, note that Theorem 1 is also valid for nonoptimal prefix codes, as long as they are canonical and their $\ell_{\max}$ is $\mathcal{O}(w)$.

We must warn the practice-oriented reader that Theorem 1 (as well as those to come) must be understood as a theoretical result. As we will explain in Section 3.5, other structures with worse theoretical guarantees perform better in practice than those chosen to obtain the best theoretical results. Our engineered implementation of Theorem 1 reaches $\mathcal{O}(\lg \lg n)$, and even $\mathcal{O}(\lg n)$, decoding time. It does, indeed, use much less space than previous model representations, but it is also much slower.

[2] In fact, most previous descriptions of canonical Huffman codes assume this increasing order, but we want to emphasize that this is essential for our construction.

3.3 Additive Approximation

In this section we exchange a small additive penalty over the optimal prefix code for an even more space-efficient representation of the code, while retaining constant-time encoding and decoding.

It follows from Milidiú and Laber's bound [ML01] that, for any ϵ with $0 < \epsilon < 1/2$, there is always a prefix code with maximum codeword length $\ell_{\max} = \lceil \lg \sigma \rceil + \lceil \log_\phi(1/\epsilon) \rceil + 1$ and average codeword length within an additive

$$\begin{aligned} \frac{1}{\phi^{\ell_{\max} - \lceil \lg(\sigma + \lceil \lg \sigma \rceil - \ell_{\max}) \rceil - 1}} &\leq \\ \frac{1}{\phi^{\ell_{\max} - \lceil \lg \sigma \rceil - 1}} &\leq \\ \frac{1}{\phi^{\log_\phi(1/\epsilon)}} &= \epsilon \end{aligned}$$

of the minimum $\mathcal{L}(P)$. The techniques described in Section 3.2 give a way to store such a code in $\sigma \lg \ell_{\max} + \mathcal{O}\big(\sigma + \ell_{\max}^2\big)$ bits, with constant-time encoding and decoding. In order to reduce the space, we note that our wavelet tree representation [BN15, Thm. 7] in fact uses $\sigma \mathcal{H}_0(L) + o(\sigma)$ bits when $\ell_{\max} = \mathcal{O}(w)$. Here $\mathcal{H}_0(L)$ denotes the empirical zero-order entropy of L. Then we obtain the following result.

Theorem 2. *Let $\mathcal{L}(P)$ be the optimal average codeword length for a distribution P over σ symbols. Then, for any $0 < \epsilon < 1/2$, under the RAM model with computer word size $w = \Omega(\lg \sigma)$, we can store a prefix code over P with average codeword length at most $\mathcal{L}(P) + \epsilon$, using $\sigma \lg \lg(1/\epsilon) + \mathcal{O}(\sigma)$ bits, such that encoding and decoding any symbol takes $\mathcal{O}(1)$ time.*

Proof. Our structure uses $\sigma \mathcal{H}_0(L) + o(\sigma) + \mathcal{O}\big(\ell_{\max}^2\big)$ bits, which is $\sigma \mathcal{H}_0(L) + o(\sigma)$ because $\ell_{\max} = \mathcal{O}(\lg \sigma)$. To complete the proof it is sufficient to show that $\mathcal{H}_0(S) \leq \lg \lg(1/\epsilon) + \mathcal{O}(1)$.

To see this, consider L as two interleaved subsequences, L_1 and L_2, of length σ_1 and σ_2, with L_1 containing those lengths $\leq \lceil \lg \sigma \rceil$ and L_2 containing those greater. Thus $\sigma \mathcal{H}_0(L) \leq \sigma_1 \mathcal{H}_0(L_1) + \sigma_2 \mathcal{H}_0(L_2) + \sigma$ (from an obvious encoding of L using L_1, L_2, and a bitmap).

Let us call $\mathrm{occ}(\ell, L_1)$ the number of occurrences of symbol ℓ in L_1. Since there are at most 2^ℓ codewords of length ℓ, assume we complete L_1 with spurious symbols so that it has exactly 2^ℓ occurrences of symbol ℓ. This completion cannot decrease $\sigma_1 \mathcal{H}_0(L_1) = \sum_{\ell=1}^{\lceil \lg \sigma \rceil} \mathrm{occ}(\ell, L_1) \lg \frac{\sigma_1}{\mathrm{occ}(\ell, L_1)}$, as increasing some $\mathrm{occ}(\ell, L_1)$ to $\mathrm{occ}(\ell, L_1) + 1$ produces a difference of $f(\sigma_1) - f(\mathrm{occ}(\ell, L_1)) \geq 0$, where $f(x) = (x+1)\lg(x+1) - x \lg x$ is increasing. Hence we can assume L_1 contains exactly 2^ℓ occurrences of symbol $1 \leq \ell \leq \lceil \lg \sigma \rceil$; straightforward calculation then shows $\sigma_1 \mathcal{H}_0(L_1) = \mathcal{O}(\sigma_1)$.

On the other hand, L_2 contains at most $\ell_{\max} - \lceil \lg \sigma \rceil$ distinct values, so $\mathcal{H}_0(L_2) \leq \lg(\ell_{\max} - \lceil \lg \sigma \rceil)$, unless $\ell_{\max} = \lceil \lg \sigma \rceil$, in which case L_2 is empty and $\sigma_2 \mathcal{H}_0(L_2) = 0$. Thus $\sigma_2 \mathcal{H}_0(L_2) \leq \sigma_2 \lg(\lceil \log_\phi(1/\epsilon) \rceil + 1) = \sigma_2 \lg\lg(1/\epsilon) + \mathcal{O}(\sigma_2)$. Combining both bounds, we get $\mathcal{H}_0(L) \leq \lg\lg(1/\epsilon) + \mathcal{O}(1)$ and the theorem holds. □

In other words, under mild assumptions, we can store a code using $\mathcal{O}(\sigma \lg\lg(1/\epsilon))$ bits at the price of increasing the average codeword length by ϵ, and in addition have constant-time encoding and decoding. For constant ϵ, this means that the code uses just $\mathcal{O}(\sigma)$ bits at the price of an arbitrarily small constant additive penalty over the shortest possible prefix code. Figure 3.2 shows an example. Note that the same reasoning of this proof, applied over the encoding of Theorem 1, yields a refined upper bound.

Corollary 1. *Let P be the frequency distribution of σ symbols for a text of length n. Then, under the RAM model with computer word size $w = \Omega(\lg n)$, we can store an optimal prefix code for P using $\sigma \lg\lg(n/\sigma) + \mathcal{O}\big(\sigma + \lg^2 n\big)$ bits, while encoding and decoding any symbol in $\mathcal{O}(1)$ time.*

Proof. Proceed as in the proof of Theorem 2, using that $\ell_{\max} \leq \log_\phi n$ and putting inside L_1 the lengths up to $\lceil \log_\phi \sigma \rceil$. Then $\sigma_1 \mathcal{H}(L_1) = \mathcal{O}(\sigma_1)$ and $\sigma_2 \mathcal{H}(L_2) \leq \lg\lg(n/\sigma) + \mathcal{O}(\sigma_2)$. □

3.4 Multiplicative Approximation

In this section we obtain a multiplicative rather than an additive approximation to the optimal prefix code, in order to achieve a sublinear-sized representation of the code. We will divide the alphabet into frequent and infrequent symbols, and store information about only the frequent ones.

Given a constant $c > 1$, we use Milidiú and Laber's algorithm [ML01] to build a prefix code with maximum codeword length $\ell_{\max} = \lceil \lg \sigma \rceil + \lceil 1/(c-1) \rceil + 1$ (our final codes will have length up to $\ell_{\max} + 1$). We call a symbol's codeword *short* if it has length at most $\ell_{\max}/c + 2$, and *long* otherwise. Notice there are $S \leq 2^{\ell_{\max}/c+2} = \mathcal{O}\big(\sigma^{1/c}\big)$ symbols with short codewords. Also, although applying Milidiú and Laber's algorithm may cause some exceptions, symbols with short codewords are usually more frequent than symbols with long ones. We will hereafter call *frequent/infrequent* symbols those encoded with short/long codewords.

Note that, if we build a canonical code, all the short codewords will precede the long ones. We first describe how to handle the frequent symbols. A perfect hash data structure [FKS84] hash will map the frequent symbols in $[1, \sigma]$ to the interval $[1, S]$ in constant time. The reverse mapping is done via a plain array $\mathsf{ihash}[1, S]$ that stores the original symbol that corresponds to each mapped symbol. We use this mapping also to reorder the frequent symbols so that the corresponding prefix

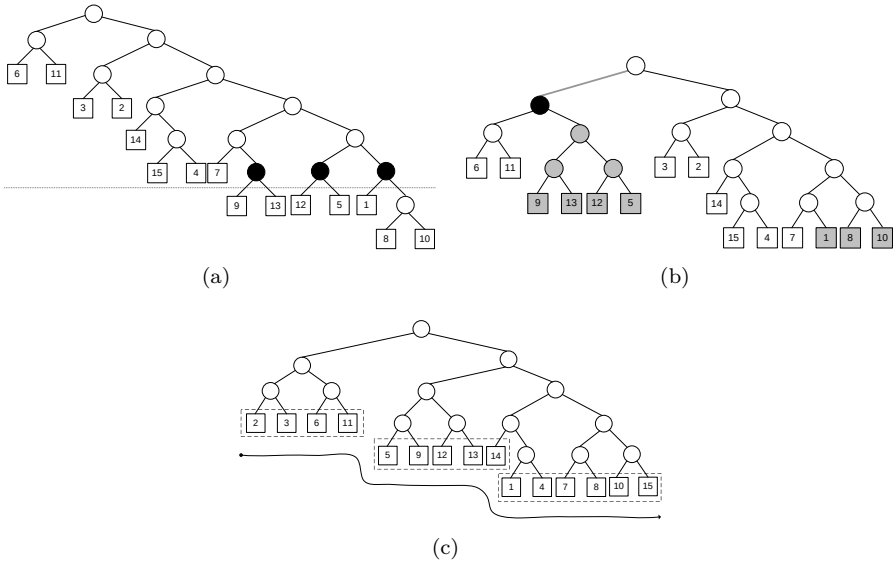


Figure 3.2: An example of Milidiú and Laber's algorithm [ML01]. In (a), a canonical Huffman tree. We set $l_{max} = 5$ and remove all the symbols below that level (marked with the dotted line), which yields three empty nodes (marked as black circles in the top tree). In (b), those black circles are replaced by the deepest symbols below level l_{max}: 1, 8, and 10. The other symbols below $\ell_{\max}$, 9, 13, 12 and 5, form a balanced binary tree that is hung from a new node created as the left child of the root (in black in the middle tree). The former left child of the root becomes the left child of this new node. Finally, in (c), we transform the middle tree into its cannonical form, but sorting those symbols belonging to the same level in increasing order.

in array Symb (recall Section 3.2) reads $1, 2, \ldots, S$. Thanks to this, we can encode and decode any frequent symbol using just first, sR, predecessor structures on both of them, and the tables hash and ihash. To encode a frequent symbol a_i, we find it in hash, obtain the mapped symbol $a' \in [1, S]$, find the predecessor $\mathsf{sR}[\ell]$ of a' and then the code is the ℓ-bit integer $c_i = \mathsf{first}[\ell] + a' - \mathsf{sR}[\ell]$. To decode a short code c_i, we first find its corresponding length ℓ using the predecessor structure on first, then obtain its mapped code $a' = \mathsf{sR}[\ell] + c_i - \mathsf{first}[\ell]$, and finally the original symbol is $_i = \mathsf{ihash}[a']$. Structures hash and ihash require $\mathcal{O}\big(\sigma^{1/c} \lg \sigma\big)$ bits, whereas sR and first, together with their predecessor structures, require less, $\mathcal{O}\big(\lg^2 \sigma\big)$ bits.

The long codewords will be replaced by *new* codewords, all of length $\ell_{\max} + 1$. Let c_{long} be the first long codeword and let ℓ be its length. Then we form the

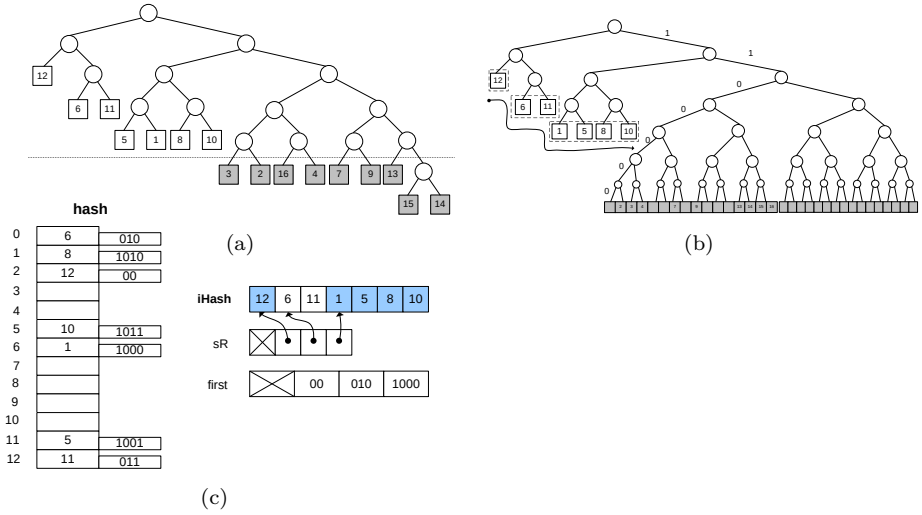


Figure 3.3: An example of the multiplicative approximation, with $\sigma = 16$ and $c = 3$. The tree shown in (a) is the result of applying the algorithm of Milidiú and Laber to a given set of codes. Now, we set $\ell_{\max} = 6$ according to our formula, and declare short those codewords of lengths up to $\lfloor \ell_{\max}/c \rfloor + 2 = 4$. Short codewords (above the dashed line on top) are stored unaltered but with all symbols at each level sorted in increasing order (b). Long codewords (below the dashed line) are extended up to length $\ell_{\max} + 1 = 7$ and reassigned a code according to their values in the contiguous slots of length 7 (those in gray in the middle). Thus, given a long codeword x, its code is directly obtained as $c'_{\text{long}} + x - 1$, where $c'_{\text{long}} = 1100000_2$ is the first code of length $\ell_{max} + 1$. In (c), a representation of the hash and inverse hash to code/decode short codewords. We set the hash size to $m = 13$ and $h(x) = (5x + 7) \bmod m$. We store the code associated with each cell.

new codeword c'_{long} by appending $\ell_{\max} + 1 - \ell$ zeros at the end of c_{long}. The new codewords will be the $(\ell_{\max}+1)$-bit integers $c'_{\text{long}}, c'_{\text{long}} + 1, \ldots, c'_{\text{long}} + \sigma - 1$. An infrequent symbol a_i will be mapped to code $c'_{\text{long}} + i - 1$ (frequent symbols a_i will leave unused symbols $c'_{\text{long}} + i - 1$). Figure 3.3 shows an example.

Since $c > 1$, we have $\sigma^{1/c} < \sigma/2$ for sufficiently large σ, so we can assume without loss of generality that there are fewer than $\sigma/2$ short codewords,[3] and thus there are at least $\sigma/2$ long codewords. Since every long codeword is replaced by at least two new codewords, the total number of new codewords is at least σ. Thus there are sufficient slots to assign codewords c'_{long} to $c'_{\text{long}} + \sigma - 1$.

To encode an infrequent symbol a_i, we first fail to find it in table hash. Then, we assign it the $(\ell_{\max}+1)$-bits long codeword $c'_{\text{long}} + i - 1$. To decode a long codeword, we first read $\ell_{\max} + 1$ bits into c_i. If $c_i \geq c'_{\text{long}}$, then the codeword is long, and corresponds to the source symbol $a_{c_i - c'_{\text{long}}+1}$. Note that we use no space to store the infrequent symbols. This leads to proving our result.

Theorem 3. *Let $\mathcal{L}(P)$ be the optimal average codeword length for a distribution P over σ symbols. Then, for any constant $c > 1$, under the RAM model with computer word size $w = \Omega(\lg \sigma)$, we can store a prefix code over P with average codeword length at most $c\,\mathcal{L}(P)$, using $\mathcal{O}\big(\sigma^{1/c} \lg \sigma\big)$ bits, such that encoding and decoding any symbol takes $\mathcal{O}(1)$ time.*

Proof. Only the claimed average codeword length remains to be proved. By analysis of the algorithm by Milidiú and Laber [ML01] we can see that the codeword length of a symbol in their length-restricted code exceeds the codeword length of the same symbol in an optimal code by at most 1, and only when the codeword length in the optimal code is at least $\ell_{\max} - \lceil \lg \sigma \rceil - 1 = \lceil 1/(c-1) \rceil$. Hence, the codeword length of a frequent symbol exceeds the codeword length of the same symbol in an optimal code by a factor of at most $\frac{\lceil 1/(c-1) \rceil + 1}{\lceil 1/(c-1) \rceil} \leq c$. Every infrequent symbol is encoded with a codeword of length $\ell_{\max} + 1$. Since the codeword length of an infrequent symbol in the length-restricted code is more than $\ell_{\max}/c + 2$, its length in an optimal code is more than $\ell_{\max}/c + 1$. Hence, the codeword length of an infrequent symbol in our code is at most $\frac{\ell_{\max}+1}{\ell_{\max}/c+1} < c$ times greater than the codeword length of the same symbol in an optimal code. Hence, the average codeword length for our code is less than c times the optimal one. □

Again, under mild assumptions, this means that we can store a code with average length within c times the optimum, in $\mathcal{O}\big(\sigma^{1/c} \lg \sigma\big)$ bits and allowing constant-time encoding and decoding.

[3] If this is not the case, then $\sigma = \mathcal{O}(1)$, so we can use any optimal encoding: there will be no redundancy over $\mathcal{L}(P)$ and the asymptotic space formula for storing the code will still be valid.

3.5 Experimental Results

We engineer and implement the optimal and approximate code representations described above, obtaining complexities that are close to the theoretical ones. We compare these with the best known alternatives to represent prefix codes we are aware of. Our comparisons will measure the size of the code representation, the encoding and decoding time and, in the case of the approximations, the redundancy on top of $\mathcal{H}(P)$.

3.5.1 Implementations

Our constant-time results build on two data structures. One is the multiary wavelet tree (`MWT`) (see Section 2.7.1). A practical study [Bow10] shows that multiary wavelet trees can be faster than binary ones, but require significantly more space (even with the better variants they design). To prioritize space, we will use binary wavelet trees, which perform the operations in time $\mathcal{O}(\lg \ell_{\max}) = \mathcal{O}(\lg \lg n)$.

The second constant-time data structure is the fusion tree [FW93], of which there are no practical implementations as far as we know. Even implementable loglogarithmic predecessor search data structures, like van Emde Boas trees [vEBKZ77], are worse than binary search for small universes like our range $[1, \ell_{\max}] = [1, \mathcal{O}(\lg n)]$. With a simple binary search on first we obtain a total encoding and decoding time of $\mathcal{O}(\lg \lg n)$, which is sufficiently good for practical purposes. Even more, preliminary experiments showed that sequential search on first is about as good as binary search in our test collections (this is also the case with classical representations [LM06]). Although sequential search costs $\mathcal{O}(\lg n)$ time, the higher success of instruction prefetching makes it much faster than binary search. Thus, our experimental results use sequential search.

To achieve space close to $\sigma\mathcal{H}_0(L)$ in the wavelet tree, we use a Huffman-shaped wavelet tree (`WTH`) (recall Section 2.7.2). The bitmaps of the wavelet tree are represented in plain form and using a space overhead of 37.5% to support `rank`/`select` operations [Nav09]. The total space of the wavelet tree is thus close to $1.375 \cdot n\mathcal{H}_0(L)$ bits in practice. Besides, we enhance these bitmaps with a small additional index to speed up `select` operations ([NP12] or see Section 2.6), which increases the constant 1.375 to at least 1.4, or more if we want more speed. A previous implementation [NO13] recasts this wavelet tree into a compressed permutation representation [BN13] of vector Symb, which leads to a similar implementation.

For the additive approximation of Section 3.3, we use the same implementation as for the exact version, after modifying the code tree as described in that section. The lower number of levels will automatically make sequence L more compressible and the wavelet tree faster.

For the multiplicative approximation of Section 3.4, we implement table hash with double hashing. The hash function is of the form $h(x, i) = (h_1(x) + (i-1) \cdot h_2(x)) \bmod m$ for the ith trial, where $h_1(x) = x \bmod m$, $h_2(x) = 1 + (x \bmod (m-1))$,

where m is a prime number. Predecessor searches over sR and first are done via binary search since, as discussed above, theoretically better predecessor data structures are not advantageous on this small domain.

3.5.1.1 Classical Huffman codes

As a baseline to compare with our encoding, we use the representation of Figure 3.1(a), using $n\,\ell_{\max}$ bits for Codes, $n \lg n$ bits for Symb, $\ell_{\max}^2$ bits for first, and $\ell_{\max} \lg n$ bits for sR. For compression, the obvious constant-time solution using Codes is the fastest one. We also implemented the fastest decompression strategies we are aware of, which are more sophisticated. The naive approach, dubbed TABLE in our experiments, consists of iteratively probing the next ℓ bits from the compressed sequence, where ℓ is the next available tree depth. If the relative numeric code resulting from reading ℓ bits exceeds the number of nodes at this level, we probe the next level, and so on until finding the right length [SK64].

Much research has focused on impoving upon this naive approach [MT97, MLMD03, CKP85, Sie88, Has95, LM06]. For instance, one could use an additional table that takes a prefix of b bits of the compressed sequence and tells which is the minimum code length compatible with that prefix. This speeds up decompression by reducing the number of iterations needed to find a valid code. This technique was proposed by Moffat and Turpin [MT97] and we call it TABLE$_S$ in our experiments. Alternatively, one could use a table that stores, for all the b-bit prefixes, the symbols that can be directly decoded from them (if any) and how many bits those symbols use. Note this technique can be combined with TABLE$_S$: if no symbol can be decoded, we use TABLE$_S$. In our experiments, we call TABLE$_E$ the combination of these two techniques.

Note that, when measuring compression/decompression times, we will only consider the space needed for compression/decompression (whereas our structure is a single one for both operations).

3.5.1.2 Hu-Tucker codes

As a representative of a suboptimal code that requires little storage space [BNO12], we also implement alphabetic codes, using the Hu-Tucker algorithm [HT71, Knu73]. This algorithm takes $\mathcal{O}(\sigma \lg \sigma)$ time and yields the optimal alphabetic code, which guarantees an average code length below $\mathcal{H}(P) + 2$. As the code is alphabetic, no permutation of symbols needs to be stored; the ith leaf of the code tree corresponds to the ith source symbol. On the other hand, the tree shape is arbitrary. We implement the code tree using succinct tree representations (recall Section 2.9), more precisely the Balanced Parentheses implementation of Sadakane and Navarro known as `FF` (see Section 8.1 for more details about `FF`), which efficiently supports the required navigation operations. This representation requires 2.37 bits per tree node, that is, 4.74σ bits for our tree (which has σ leaves and $\sigma - 1$ internal nodes).

Collection	Length (n)	Alphabet (σ)	Entropy ($\mathcal{H}(P)$)	Depth ($\ell_{\max}$)	Level entr. ($\mathcal{H}_0(L)$)
`EsWiki`	200,000,000	1,634,145	11.12	28	2.24
`EsInv`	300,000,000	1,005,702	5.88	28	2.60
`Indo`	120,000,000	3,715,187	16.29	27	2.51

Table 3.1: Main statistics of the texts used.

`FF` represents general trees, so we convert the binary code tree into a general tree using the well-known mapping [MR01]: we identify the left child of the code tree with the first child in the general tree, and the right child of the code tree with the next sibling in the general tree. The general tree has an extra root node whose children are the nodes in the rightmost path of the code tree.

With this representation, compression of symbol c is carried out by starting from the root and descending towards the cth leaf. We use the number of leaves on the left subtree to decide whether to go left or right. The left/right decisions made in the path correspond to the code. In the general tree, we compute the number of nodes k in the subtree of the first child, and then the number of leaves in the code tree is $k/2$. For decompression, we start from the root and descend left or right depending on the bits of the code. Each time we go right, we accumulate the number of leaves on the left, so that when we arrive at a leaf the decoded symbol is the final accumulated value plus 1.

3.5.2 Experimental Setup

We used an isolated AMD Phenom(tm) II X4 955 running at 800MHz with 8GB of RAM memory and a ST3250318AS SATA hard disk. The operating system is GNU/Linux, Ubuntu 10.04, with kernel 3.2.0-31-generic. All our implementations use a single thread and are coded in `C++`. The compiler is `gcc` version 4.6.3, with `-O9` optimization. Time results refer to CPU user time. The stream to be compressed and decompressed is read from and written to disk, using the buffering mechanism of the operating system.

We use three datasets[4] in our experiments. `EsWiki` is a sequence of word identifiers obtained by stemming the Spanish Wikipedia with the Snowball algorithm. Compressing natural language using word-based models is a strong trend in text databases [Mof89]. `EsInv` is the concatenation of differentially encoded inverted lists of a random sample of the Spanish Wikipedia. These have large alphabet sizes but also many repetitions, so they are highly compressible. Finally, `Indo` is the concatenation of the adjacency lists of Web graph `Indochina-2004` available at

[4]Made available in `http://lbd.udc.es/research/ECRPC`

Collection	Naive (σw)	Engineered ($\sigma\,\ell_{\max}$)	Canonical ($\sigma \lg \sigma$)	Ours ($\sigma \mathcal{H}_0(L)$)	Compressed [TM00]
`EsWiki`	6.23 MB	5.45 MB	4.02 MB	0.44 MB	0.45 MB
`EsInv`	3.83 MB	3.35 MB	2.39 MB	0.31 MB	0.33 MB
`Indo`	14.17 MB	11.96 MB	9.67 MB	1.11 MB	1.18 MB

Table 3.2: Rough minimum size of various model representations.

`http://law.di.unimi.it/datasets.php`. Compressing adjacency lists to zero-order entropy is a simple and useful tool for graphs with power-law degree distributions, although it is usually combined with other techniques [CN10a]. We use a prefix of each of the sequences to speed up experiments.

Table 3.1 gives various statistics on the collections. Apart from n and σ, we give the empirical entropy of the sequence ($\mathcal{H}(P)$, in bits per symbol or bps), the maximum length of a Huffman code ($\ell_{\max}$), and the zero-order entropy of the sequence of levels ($\mathcal{H}_0(L)$, in bps). It can be seen that $\mathcal{H}_0(L)$ is significantly smaller than $\lg \ell_{\max}$, thus our compressed representation of L can indeed be up to an order of magnitude smaller than the worst-case upper bound of $\sigma \lg \ell_{\max}$ bits.

Before we compare the exact sizes of different representations, which depend on the extra data structures used to speed up encoding and decoding, Table 3.2 gives the size of the basic data that must be stored in each case. The first column shows σw, the size of a naive model representation using computer words of $w = 32$ bits. The second shows $\sigma\,\ell_{\max}$, which corresponds to a more engineered representation where we use only the number of bits required to describe a codeword. In these two, more structures are needed for decoding but we ignore them. The third column gives $\sigma \lg \sigma$, which is the main space cost of a canonical Huffman tree representation: basically the permutation of symbols (different ones for encoding and decoding). The fourth column shows $\sigma \mathcal{H}_0(L)$, which is a lower bound on the size of our model representation (the exact value will depend on the desired encoding/decoding speed). These raw numbers explain why our technique will be much more effective to represent the model than the typical data structures, and that we can expect up to 7–9-fold space reductions (these will decrease to 6–8-fold on the actual structures). Indeed, this entropy space is close to that of a sophisticated model representation [TM00] that can be used only for transmitting the model in compressed form; this is shown in the last column.

3.5.3 Representing Optimal Codes

Figure 3.4 compares compression and decompression times, as a function of the space used by the code representations, of our new data structure (COMPR) versus

the table based representations described in Section 3.5.1 (TABLE, TABLE$_S$, and TABLE$_E$). We used sampling periods of $\{16, 32, 64, 128\}$ for the auxiliary data structures added to the wavelet tree bitmaps to speed up `select` operations [NP12], and parameter $b = 14$ for table based approaches (this gave us the best time performance).

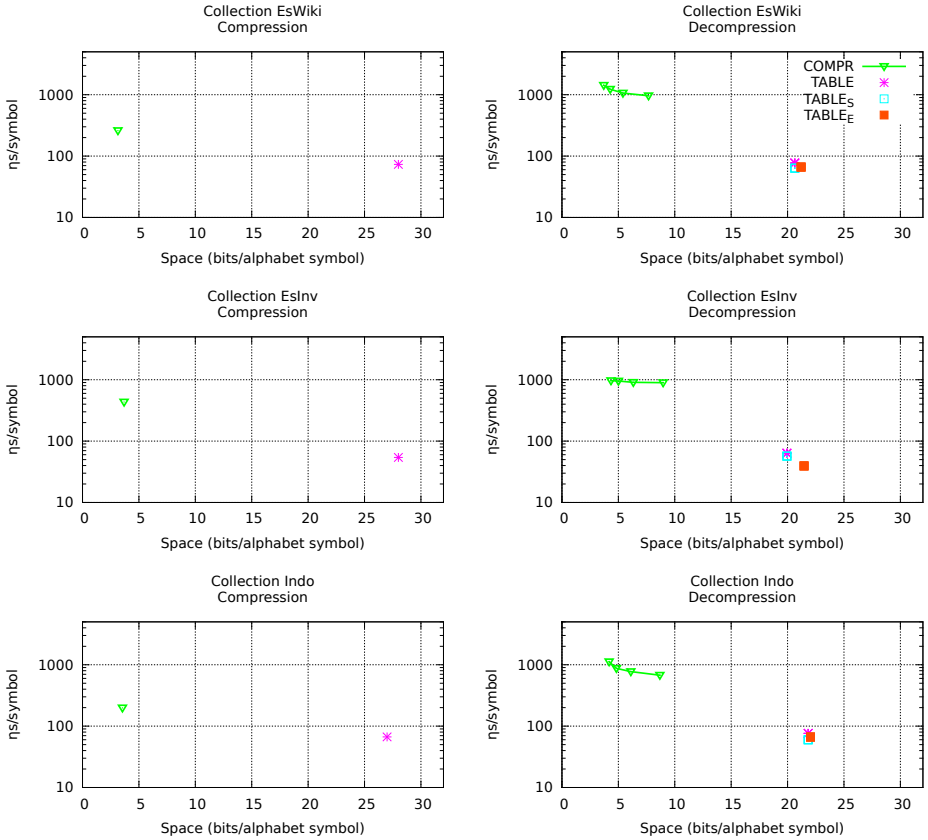


Figure 3.4: Code representation size versus compression (left) and decompression (right) time for table based representations (TABLE, TABLE$_S$, and TABLE$_E$) and ours (COMPR). Time (in logscale) is measured in nanoseconds per symbol.

It can be seen that our compressed representations takes just around 12% of the space of the table implementation for compression (an 8-fold reduction), while being 2.5–8 times slower. Note that compression is performed by carrying out `rank`

operations on the wavelet tree bitmaps. Therefore, we do not consider the space overhead incurred to speed up `select` operations, and we only plot a single point for technique COMPR at compression charts. Also, we only show the simple (and most compact) TABLE variant, as the improvements of the others apply only to decompression.

For decompression, our solution (COMPR) takes 17% to 45% of the space of the TABLE$_*$ variants (thus reaching almost a 6-fold space reduction), but it is also 12–24 times slower. This is because our solution uses operation `select` for decompression, and this is slower than `rank` even with the structures for speeding it up.

Overall, our compact representation is able to compress at a rate around 2.5–5 MB/sec and decompress at 1 MB/sec, while using much less space than a classical Huffman implementation (which compresses/decompresses at around 14–25 MB/sec).

Finally, note that we only need a single data structure to both compress and decompress, while the naive approach uses different tables for each operation. In the cases where both functionalities are simultaneously necessary (as in compressed sequence representations [Nav14]), our structure uses as little as 7% of the space needed by a classical representation.

3.5.4 Length-Limited Codes

In the theoretical description, we refer to an optimal technique for limiting the length of the code trees to a given value $\ell_{\max} \geq \lceil \lg \sigma \rceil$ [LH90], as well as several heuristics and approximations:

- `Milidiú`: the approximate technique proposed by Milidiú and Laber [ML01] that nevertheless guarantees the upper bound we have used in the paper. It takes $\mathcal{O}(\sigma)$ time.
- `Increase`: inspired in the bounds of Katona and Nemetz [KN76], we start with $f = 2$ and set to f the frequency of each symbol whose frequency is $< f$. Then we build the Huffman tree, and if its height is $\leq \ell_{\max}$, we are done. Otherwise, we increase f by 1 and repeat the process. Since the Huffman construction algorithm is linear-time once the symbols are sorted by frequency and the process does not need to reorder them, this method takes $\mathcal{O}\big(\sigma \lg(\sigma\phi^{-\ell_{\max}})\big) = \mathcal{O}(\sigma \lg \sigma)$ time if we use exponential search to find the correct f value. A close predecessor of this method appears in Chapter 9 of *Managing Gigabytes* [WMB99]. They use a multiplicative instead of an additive approximation, so as to find an appropriate f faster. Thus they may find a value of f that is larger than the optimal.
- `Increase-A`: analogous to `Increase`, but instead adds f to the frequency of each symbol.
- `Balance`: the technique (e.g., see [BN09]), that balances the parents of the maximal subtrees that, even if balanced, exceed the maximum allowed height.

It also takes $\mathcal{O}(\sigma)$ time. In the case of a canonical Huffman tree, this is even simpler, since only one node along the rightmost path of the tree needs to be balanced.

- `Optimal`: the package-merge algorithm of Larmore and Hirshberg [LH90]. Its time complexity is $\mathcal{O}(\sigma\,\ell_{\max})$.

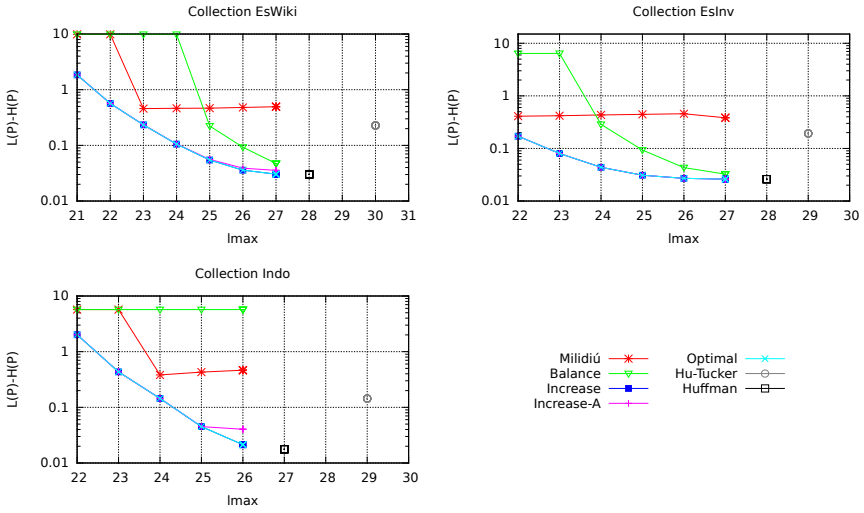


Figure 3.5: Comparison of the length-restricted approaches measured as their additive redundancy (in logscale) over the zero-order empirical entropy, $\mathcal{H}(P)$, for each value of $\ell_{\max}$. We also include Hu-Tucker and Huffman as reference points.

Figure 3.5 compares the techniques for all the meaningful $\ell_{\max}$ values, showing the additive redundancy they produce over $\mathcal{H}(P)$. It can be seen that the average code lengths obtained by `Milidiú`, although they have theoretical guarantees, are not so good in practice. They are comparable with those of `Balance`, a simpler and still linear-time heuristic, which however does not provide any guarantee and sometimes can only return a completely balanced tree. On the other hand, technique `Increase` performs better than or equal to `Increase-A`, and actually matches the average code length of `Optimal` systematically in the three collections.

Techniques `Milidiú`, `Balance`, and `Optimal` are all equally fast in practice, taking about 2 seconds to find their length-restricted code in our collections. The time for `Increase` and `Increase-A` depends on the value of $\ell_{\max}$. For large values of $\ell_{\max}$, they also take around 2 seconds, but this raises up to 20 seconds when

$\ell_{\max}$ is closer to $\lceil \lg \sigma \rceil$ (and thus the value f to add is larger, up to 100–300 in our sequences).

In practice, technique `Increase` can be recommended for its extreme simplicity to implement and very good approximation results. If the construction time is an issue, then `Optimal` should be used. It performs fast in practice and it is not so hard to implement[5]. For the following experiments, we will use the results of `Optimal/Increase`.

As a final note, observe that by restricting the code length to, say, $\ell_{\max} = 22$ on `EsWiki` and `EsInv` and $\ell_{\max} = 23$ on `Indo`, the additive redundancy obtained is below $\epsilon = 0.6$, and the redundancy is below 5% of $\mathcal{H}(P)$.

3.5.5 Approximations

Now we evaluate the additive and multiplicative approximations, in terms of average code length $\mathcal{L}$, compression and decompression performance. We compare them with two optimal model representations, OPT-T and OPT-C, which correspond to TABLE and COMPR of Section 3.5.3. The additive approximations (Section 3.3) included, ADD+T and ADD+C, are obtained by restricting the maximum code lengths to $\ell_{\max}$ and storing the resulting codes using TABLE or COMPR, respectively. We show one point per $\ell_{\max} = 22 \ldots 27$ on `EsWiki` and `EsInv`, and $\ell_{\max} = 22 \ldots 26$ on `Indo`. For the multiplicative approximation (Section 3.4), we test the variants MULT-$\ell_{\max}$, which limit $\ell_{\max}$ to 25 and 26, and use c values 1.5, 1.75, 2, and 3. For all the solutions that use a wavelet tree, we have fixed a `select` sampling rate to 32.

Figure 3.6 shows the results in terms of bps for storing the model versus the resulting redundancy of the code, measured as $\mathcal{L}(P)/\mathcal{H}(P)$.

The additive approximations have a mild impact when implemented in classical form. However, the compact representation, ADD+C, reaches half the space of our exact compact representation, OPT-C. This is obtained at the price of a modest redundancy, below 5% in all cases, if one uses reasonable values for $\ell_{\max}$.

With the larger c values, the multiplicative approach is extremely efficient for storing the model, reaching reductions up to 2 and 3 orders of magnitude with respect to the classic representations. However, this comes at the price of a redundancy that can reach 50%. The redundancy may go beyond $\lceil \lg \sigma \rceil / \mathcal{H}(P)$, at which point it is better to use a plain code of $\lceil \lg \sigma \rceil$ bits. Instead, with value $c = 1.75$, the model size is still 20 times smaller than a classical representation, and 2–3 times smaller than the most compact representation of additive approximations, with a redundancy only slightly over 10%.

Figure 3.7 compares these representations in terms of compression and decompression performance. The numbers near each point show the redundancy (as a percentage over the entropy) of the model representing that point. We use

[5]There are even some public implementations, for example `https://gist.github.com/imaya/3985581`

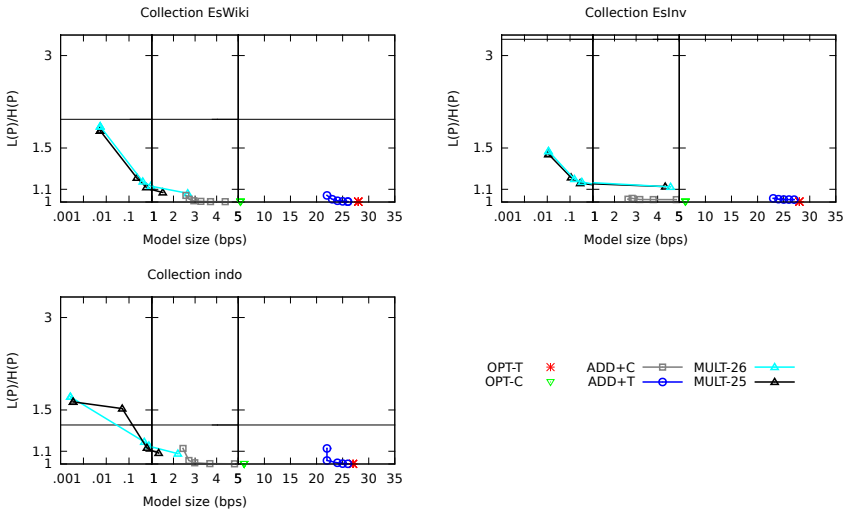


Figure 3.6: Relation between the size of the model and the average code length. The x-axes are in logscale for values smaller than 1 and in two linear scales for 1–5 and 5–35. The horizontal line shows the limit $\lceil \lg n \rceil / \mathcal{H}(P)$, where no compression is obtained compared with a fixed-length code.

ADD+C with values $\ell_{\max} = 22$ on `EsWiki` and `EsInv` and $\ell_{\max} = 23$ on `Indo`. For ADD+T, the decompression times are the same for all the tested $\ell_{\max}$ values. In this figure we set the `select` samplings of the wavelet trees to $(32, 64, 128)$. We also include in the comparison the variant MULT-26 with $c = 1.75$ and 1.5.

It can be seen that the multiplicative approach is very fast, comparable to the table-based approaches ADD+T and OPT-T: 10%–50% slower at compression and at most 20% slower at decompression. Within this speed, if we use $c = 1.75$, the representation is 6–11 times smaller than the classical one for compression and 5–9 times for decompression, at the price of about 10% of redundancy. If we choose $c = 1.5$, the redundancy increases to about 20% but the model becomes an order of magnitude smaller.

The compressed additive approach (ADD+C) achieves a smaller model than the multiplicative one with $c = 1.75$ (it is 14 times smaller than the classical representation for compression and 11 times for decompression). This is achieved with significantly less redundancy than the multiplicative model, just 3%–5%. However, although ADD+C is about 20%–30% faster than the exact code OPT-C, it is still significantly slower than the table-based representations (2–5.5 times slower for compression and 9–17 for decompression).

Finally, we can see that our compact implementation of Hu-Tucker codes

achieves competitive space, but it is an order of magnitude slower than our additive approximations, which can always use simultaneously less space and time. With respect to the redundancy, Figure 3.5 shows that Hu-Tucker codes are equivalent to our additive approximations with $\ell_{\max} = 23$ on `EsWiki`, $\ell_{\max} = 22$ on `EsInv`, and $\ell_{\max} = 24$ on `Indo`. This shows that the use of alphabetic codes as a suboptimal code to reduce the model representation size is inferior, in all aspects, to our additive approximations. Figure 3.7 shows that Hu-Tucker is also inferior, in the three aspects, to our compact optimal codes, OPT-C. We remark that alphabetic codes are interesting by themselves for other reasons, in contexts where preserving the order of the source symbols is important.

3.6 Discussion

We have explored the problem of providing compact representations of Huffman models. The model size is relevant in several applications, particularly because it must reside in main memory for efficient compression and decompression.

We have proposed new representations achieving constant compression and decompression time per symbol while using $\mathcal{O}(\sigma \lg \lg(n/\sigma))$ bits per symbol, where σ is the alphabet size and n the sequence length. This is in contrast to the (at least) $\mathcal{O}(\sigma \lg \sigma)$ bits used by previous representations. In our practical implementation, the time complexities are $\mathcal{O}(\lg \lg n)$ and even $\mathcal{O}(\lg n)$, but we do achieve 8-fold space reductions for compression and up to 6-fold for decompression. This comes, however, at the price of increased compression and decompression time (2.5–8 times slower at compression and 12–24 at decompression), compared to current representations. In low-memory scenarios, the space reduction can make the difference between fitting the model in main memory or not, and thus the increased times are the price to pay.

We also showed that, by tolerating a small additive overhead of ϵ on the average code length, the model can be stored in $\mathcal{O}(\sigma \lg \lg(1/\epsilon))$ bits, while maintaining constant compression and decompression time. In practice, these additive approximations can halve our compressed model size (becoming 11–14 times smaller than a classical representation), while incurring a very small increase (5%) in the average code length. They are also faster, but still 2–5.5 times slower for compression and 5–9 for decompression.

Finally, we showed that a multiplicative penalty in the average code length allows storing the model in $o(\sigma)$ bits. In practice, the reduction in model size is sharp, while the compression and decompression times are only 10%–50% and 0%–20% slower, respectively, than classical implementations. Redundancies are higher, however. With 10% of redundancy, the model size is close to that of the additive approach, and with 20% the size decreases by another order of magnitude.

Some challenges for future work are:

- Adapt these representations to dynamic scenarios, where the model undergoes changes as compression/decompression progresses. While our compact

representations can be adapted to support updates, the main problem is how to efficiently maintain a dynamic canonical Huffman code. We are not aware of such a technique.

- Find more efficient representations of alphabetic codes. Our baseline achieves reasonably good space, but the navigation on the compact tree representations slows it down considerably. It is possible that faster representations supporting left/right child and subtree size can be found.
- Find constant-time encoding and decoding methods that are fast and compact in practice. Multiary wavelet trees [Bow10] are faster than binary wavelet trees, but generally use much more space. Giving them the shape of a (multiary) Huffman tree and using plain representations for the sequences in the nodes could reduce the space gap with our binary Huffman-shaped wavelet trees used to represent L. As for the fusion trees, looking for a practical implementation of trees with arity w^ϵ, which outperforms a plain binary search, is interesting not only for this problem, but in general for predecessor searches on small universes.

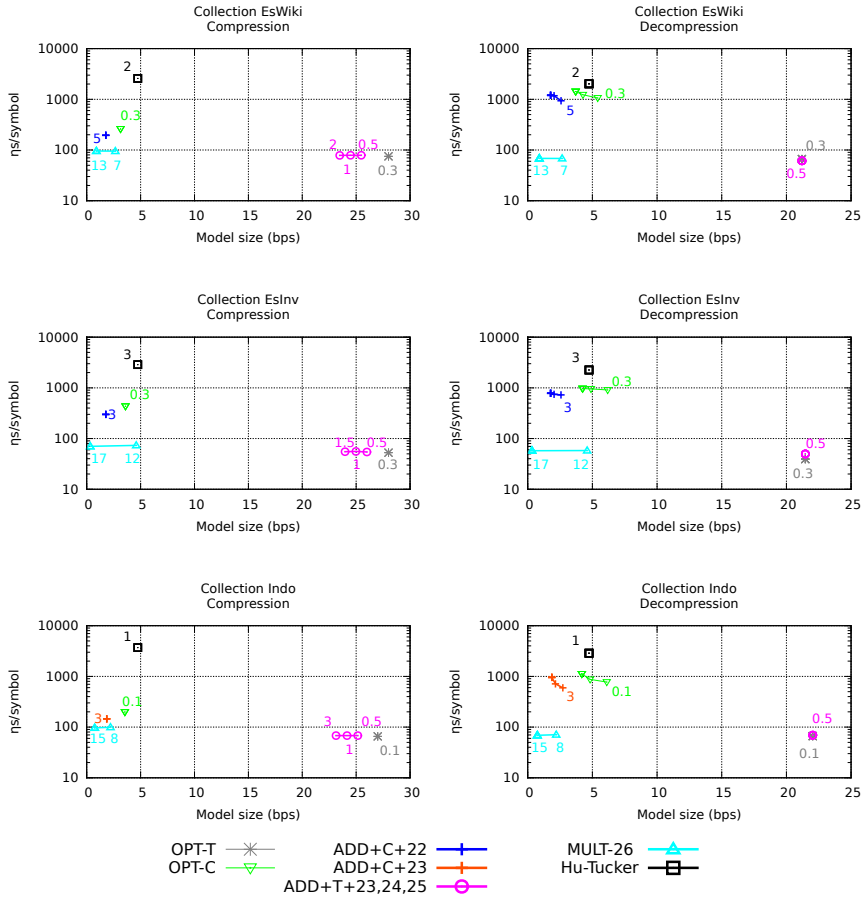


Figure 3.7: Space/time performance of the approximate and exact approaches on compression (left) and decompression (right). Times are in nanoseconds per symbol and in logscale. The numbers around the points are their redundancy as a percentage of the entropy.

Chapter 4

The Compressed Wavelet Matrix

In many applications related to text indexing and succinct data structures, it is necessary to represent a sequence $S[1, n]$ over an integer alphabet $[1, \sigma]$ so as to support `rank`, `select`, and `access` queries (recall Section 2.7).

Some examples where this problem arises are indexes for supporting indexed pattern matching on strings [GGV03, GV06, FM05, FMMN07, NM07], indexes for solving computational biology problems on sequences [SOG10, BGOS11], simulation of inverted indexes over natural language text collections [BFLN12, AGO10], representation of labeled trees and XML structures [BDM$^+$05, BAHM12, FLMM09, ACM$^+$15, BHMR11], representation of binary relations and graphs [BGMR07, CN10a, BCN10, BHMR11], solving document retrieval problems [VM07, GNP12], and many more.

An elegant data structure to solve this problem is the *wavelet tree* (see Section 2.7.1). In its most basic form, this is a balanced tree of $O(\sigma)$ nodes storing bitmaps. It requires $n \lg \sigma + o(n \lg \sigma) + O(\sigma \lg n)$ bits to represent S and solves the three queries in time $O(\lg \sigma)$. The wavelet tree supports not only the three queries we have mentioned, but more general range search operations that find applications in representing geometric grids [Cha88, BHMM09, BLNS10, BCN10, NNR13] and text indexes based on them [Nav04, FM05, MN07b, CHSV08, CN10b, KN13, NN12], complex queries on numeric sequences [GPT09, KN13, GKNP13, FOP14], and many others. Various recent surveys [NM07, FGM09, GVX11, Mak12, Nav12] are dedicated, partially or totally, to the number of applications of this versatile data structure.

In various applications, the alphabet size σ is significant compared to the length n of the sequence. Some examples are sequences of words (seen as integer tokens) when simulating inverted indexes, sequences of XML tags, and sequences of document

numbers in document retrieval. When using wavelet trees to represent grids, the sequence length n becomes the width of the grid and the alphabet size becomes the height of the grid, and both are equal in most cases. Finally, the problem arises when a sequence is indexed as many short sequences, each of which is relatively short compared to the alphabet [KP11].

A large value of σ affects the space usage of wavelet trees. A *pointerless wavelet tree* [MN07b] concatenates all the bitmaps levelwise and removes the $O(\sigma \lg n)$ bits from the space. It retains the time complexity of pointer-based wavelet trees, albeit it is slower in practice. This representation can be made to use $n\mathcal{H}_0(S) + o(n \lg \sigma)$ bits, where $\mathcal{H}_0(S) \leq \lg \sigma$ is the per-symbol zero-order entropy of S, by using compressed bitmaps [RRR07, GGV03]. This makes the wavelet tree traversal even slower in practice, however.

The *wavelet matrix* [CN12] is an alternative representation of the balanced pointerless wavelet tree that reorders the nodes in each level, in a way that retains all the wavelet tree functionality while the traversals needed to carry out the operations are simplified and sped up. The wavelet matrix then retains all the capabilities of wavelet trees, is resistant to large alphabets, and its speed gets close to that of pointer-based wavelet trees. It can also obtain zero-order compression by compressing the bitmaps (which slows it down).

A pointer-based wavelet tree, instead, can achieve zero-order compression by replacing the balanced tree by the Huffman tree [Huf52] (see Section 2.7.2). Then, even without compressing the bitmaps, the storage space becomes $n(\mathcal{H}_0(S) + 1) + o(n(\mathcal{H}_0(S) + 1)) + O(\sigma \lg n)$ bits. Adding bitmap compression removes the n bits of the Huffman redundancy. In addition, this technique is faster than the basic one, as the average access time is $O(\mathcal{H}_0(S))$. By using canonical Huffman codes (see Section 2.4.2.1), a pointerless Huffman-shaped wavelet tree is also possible [ZPYL08]. This removes the $O(\sigma \lg n)$ bits of the pointers, whereas those of the Huffman model can be reduced to $\sigma \lg \sigma + O(\sigma)$, and even $O(\sigma \lg \lg n)$, in the case of a canonical code (see Chapter 3). Therefore the total space can be written as $n\mathcal{H}_0(S) + o(n(\mathcal{H}_0(S) + 1)) + O(\sigma \lg \lg n)$ bits.

Other than wavelet trees, Golynski et al. (see Section 2.7.3) proposed a sequence representation for large alphabets, which uses $n \lg \sigma + o(n \lg \sigma)$ bits (no compression) and offers much faster time complexities to support the three operations, $O(\lg \lg \sigma)$. Later, Barbay et al. (see Section 2.7.4) built on this idea to obtain zero-order compression, $n\mathcal{H}_0(S) + o(n(\mathcal{H}_0(S) + 1))$ bits, while retaining the times. This so-called "alphabet-partitioned" representation does not, however, offer the richer functionality of wavelet trees. Moreover, as shown in their experiments [BCG$^+$14], its sublinear space terms are higher in practice than those of a zero-order compressed wavelet tree (yet their better complexity does show up in practice). There are recent theoretical developments slightly improving those complexities [BN15], but their sublinear space terms would be even higher in practice.

In this chapter we present the *compressed wavelet matrix*, which is a combination

of wavelet matrices and Huffman-shaped wavelet trees, so as to obtain Huffman-shaped wavelet matrices. These yield simultaneously zero-order compression and fast operations. It turns out, however, that the canonical Huffman codes cannot be directly combined with the node numbering induced by the wavelet matrix, so we derive an alternative code assignment scheme that is also optimal and compatible with the wavelet matrix.

We implement these variants and test them over various real-life sequences, showing that a few versions of the wavelet matrix dominate all the wavelet tree variants across the space/time tradeoff map, on diverse sequences over large alphabets.

This chapter is organized as follows: Section 4.1 describes the previous work; Section 4.2 presents the compressed wavelet matrix; Section 4.3 provides a complete experimental evaluation; and Section 4.4 gives our conclusions.

4.1 Related Work

Given a sequence $S[1, n]$ over $\Sigma = [1, \sigma]$, in this section we introduce several data structures aimed at solving `rank`, `select`, and `access` queries for sequences with large alphabets.

4.1.1 Pointerless Wavelet Trees

Having in mind the wavelet tree data structure (`WT`) presented in Section 2.7.1, it is important to notice that if all the internal nodes at each level of the wavelet tree are all to the left, it is possible to concatenate all the bitmaps at each level and still retain the same functionality [MN07b]. The result is a data structure named pointerless wavelet tree (`WTNP`) in which, instead of a bitmap per node v, there will be a single bitmap per level ℓ, $\tilde{B}_\ell[1, n]$. Figure 4.1 (bottom) illustrates this arrangement. The main problem is how to keep track of the range $\tilde{B}_\ell[s_v, e_v]$ corresponding to a node v of depth ℓ.

4.1.1.1 The Strict Variant

The strict variant [MN07b] stores no data apart from the $\lceil \lg \sigma \rceil$ pointers to the level bitmaps. Keeping track of the node ranges is not hard if we start at the root (as in `access` and `rank`). Initially, we know that $[s_\nu, e_\nu] = [1, n]$, that is, the whole bitmap $\tilde{B}_0$ is equal to the bitmap of the root, B_ν. Now, imagine that we have navigated towards a node v at depth ℓ, and know $[s_v, e_v]$. The two children of v share the same interval $[s_v, e_v]$ at $\tilde{B}_{\ell+1}$. The split point is $m = \texttt{rank}_0(\tilde{B}_\ell, e_v) - \texttt{rank}_0(\tilde{B}_\ell, s_v - 1)$, the number of 0s in $\tilde{B}_\ell[s_v, e_v]$. Then, if we descend to the left child v_l, we will have $[s_{v_l}, e_{v_l}] = [s_v, s_v + m - 1]$. If we descend to the right child v_r, we will have $[s_{v_r}, e_{v_r}] = [s_v + m, e_v]$.

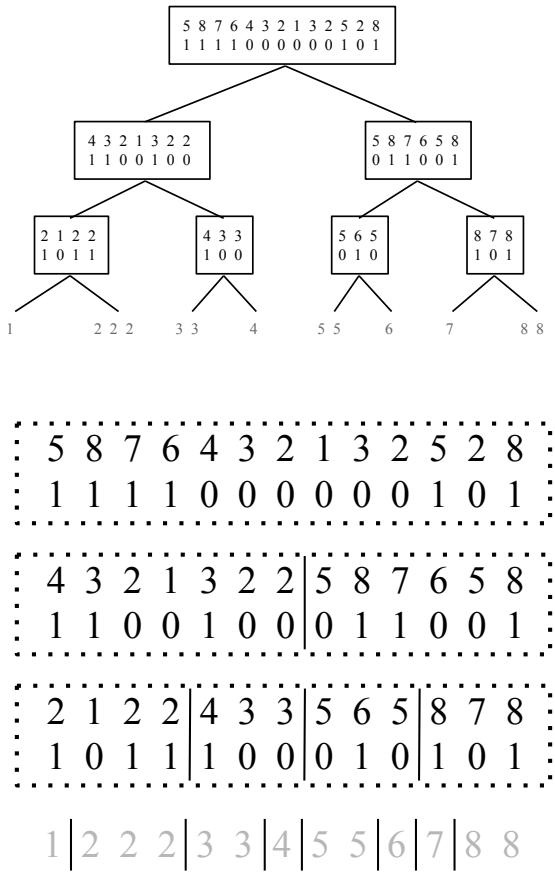


Figure 4.1: On the top, the standard wavelet tree (`WT`) over a sequence. The subsequences S_v are not stored. The bitmaps B_v, are stored, as well as the tree topology. On the bottom, its pointerless version (`WTNP`). The divisions into nodes are not stored but computed on the fly.

Algorithm 5 Pointerless wavelet tree algorithms (strict variant): On the wavelet tree of sequence S, $\mathbf{acc}(0,i,0,n)$ returns $S[i]$; $\mathbf{rnk}(0,a,i,0,n)$ returns $\texttt{rank}_a(S,i)$; and $\mathbf{sel}(0,a,j,0,n)$ returns $\texttt{select}_a(S,j)$. For simplicity we have omitted the computation of $[\alpha_v,\omega_v]$.

```
acc(ℓ, i, p, e)
  if ω_v − α_v = 0 then
    return α_v
  end if
  l ← rank_0(B̃_ℓ, p)
  r ← rank_0(B̃_ℓ, e)
  if B̃_ℓ[i] = 0 then
    z ← rank_0(B̃_ℓ, p+i)
    return
    acc(ℓ+1,
      z−l, p, p+r−l)
  else
    z ← rank_1(B̃_ℓ, p+i)
    return
    acc(ℓ+1,
      z−(p−l), p+r−l, e)
  end if
```

```
rnk(ℓ, a, i, p, e)
  if ω_v − α_v = 0 then
    return i
  end if
  l ← rank_0(B̃_ℓ, p)
  r ← rank_0(B̃_ℓ, e)
  if a ≤ 2^⌈lg(ω_v−α_v)⌉−1 then
    z ← rank_0(B̃_ℓ, p+i)
    return
    rnk(ℓ+1, a,
      z−l, p, p+r−l)
  else
    z ← rank_1(B̃_ℓ, p+i)
    return
    rnk(ℓ+1, a,
      z−(p−l), p+r−l, e)
  end if
```

```
sel(ℓ, a, j, p, e)
  if ω_v − α_v = 0 then
    return j
  end if
  l ← rank_0(B̃_ℓ, p)
  r ← rank_0(B̃_ℓ, e)
  if a ≤ 2^⌈lg(ω_v−α_v)⌉−1 then
    j←sel(ℓ+1, a, j, p, p+r−l)
    return
      select_0(B̃_ℓ, l+j)−p
  else
    j←sel(ℓ+1, a, j, p+r−l, e)
    return
      select_1(B̃_ℓ(p−l)+j)−p
  end if
```

Things are a little bit harder for `select`, because we must proceed upwards. In the strict variant, the way to carry out $\texttt{select}_a(S,j)$ is to first descend to the leaf corresponding to symbol a, and then track the leaf position j up to the root as we return from the recursion.

Algorithm 5 gives the pseudocode (we use $p = s-1$ instead of $s = s_v$). Note that, compared to the standard version, the strict variant requires two extra binary `rank` operations per original binary `rank`, on the top-down traversals (i.e., for queries `access` and `rank`). For query `select`, the strict variant requires two extra binary `rank` operations per original binary `select`. Thus the times may up to triple for these traversals.[1]

4.1.1.2 The Extended Variant

The *extended* variant [CN08], instead, stores an array $C[1,\sigma]$ of pointers to the σ starting positions of the symbols in the (virtual) array of the leaves, or said another way, $C[a]$ is the number of occurrences of symbols smaller than a in S. Note this array requires $O(\sigma \lg n)$ bits (or at best $O(\sigma \lg(n/\sigma)) + o(n)$ if represented as the

[1] In practice the effect is not so large because of cache effects when s_v is close to e_v. In addition, binary `select` is more expensive than `rank` in practice, thus the impact on query `select` is lower.

RRR compressed bitmap from Section 2.6), but the constant is much lower than on a pointer-based tree (which stores the left child, the right child, the parent, the value n_v, the pointer to bitmap B_v, pointers to the leaves, etc.). We also need to indicate the level in which each leaf is found.

With the help of array C, the number of operations becomes closer to the standard version, since C lets us compute the ranges: The range of any node v is simply $[s_v, e_v] = [C[\alpha_v] + 1, C[\omega_v]]$. In the algorithms for queries `access` and `rank`, where we descend from the root, the values α_v and ω_v are easily maintained. Thus we do not need to compute r in Algorithm 5, as it is used only to compute $e = e_v = C[\omega_v]$. Thus we require only one extra binary `rank` operation per level.

This is slightly more complicated when solving query $\texttt{select}_a(S, j)$. We start at offset j in the interval $[C[\alpha_u] + 1, C[\omega_u]]$ for $(\alpha_u, \omega_u) = (a, a + 1)$ and track this position upwards: If the leaf u is a left child of its parent v (i.e., if α_u is even), then the parent's range (in the deepest bitmap $\tilde{B}_\ell$) is $(\alpha_v, \omega_v) = (\alpha_u, \omega_u + 1)$. Instead, if the leaf is a right child of its parent, then the parent's range is $(\alpha_v, \omega_v) = (\alpha_u - 1, \omega_u)$. We use binary `select` on the range $[C[\alpha_v] + 1, C[\omega_v]]$ to map the position j to the parent's range. Now we proceed similarly at the parent w of v. If $(\alpha_v - 1) \bmod 2 = 0$, then v is the left child of w, otherwise it is the right child. In the first case, the range of w in bitmap $\tilde{B}_{\ell-1}$ is $(\alpha_w, \omega_w) = (\alpha_v, \omega_v + 2)$, otherwise it is $(\alpha_w, \omega_w) = (\alpha_v - 2, \omega_v)$. We continue until the root, where j is the answer. In this case we need only one extra binary `rank` operation per level. Algorithm 6 details the algorithms.

4.1.2 The Wavelet Matrix

The idea of the wavelet matrix is to break the assumption that the children of a node v, at interval $\tilde{B}_\ell[s_v, e_v]$, must be aligned to it and occupy the interval $\tilde{B}_{\ell+1}[s_v, e_v]$ (as wavelet trees do). Freeing the structure from this assumption allows us to design a much simpler mapping mechanism from one level to the next: *all* the zeros of the level go left, and *all* the ones go right. For each level, we will store a single integer z_ℓ that tells the number of 0s in level ℓ. This requires just $O(\lg n \lg \sigma)$ bits, which is insignificant, and allows us to implement the pointerless mechanisms in a simpler and faster way.

More precisely, if $\tilde{B}_\ell[i] = 0$, then the corresponding position at level $\ell + 1$ will be $\texttt{rank}_0(\tilde{B}_\ell, i)$. If $\tilde{B}_\ell[i] = 1$, the position at level $\ell + 1$ will be $z_\ell + \texttt{rank}_1(\tilde{B}_\ell, i)$. Note that we can map the position without knowledge of the boundaries of the node the position belongs. Still, every node v at level ℓ occupies a contiguous range in $\tilde{B}_\ell$, as proved next.

Proposition 1. *All the bits in any bitmap $\tilde{B}'_\ell$ of the pointerless wavelet tree that correspond to a wavelet tree node v are also contiguous in the bitmap $\tilde{B}_\ell$ of the the wavelet matrix.*

Proof. This is obviously true for the root $v = \nu$, as it corresponds to the whole $\tilde{B}'_0 = \tilde{B}_0$. Now, assuming it is true for a node v, with interval $\tilde{B}_\ell[s_v, e_v]$, all the

Algorithm 6 Pointerless wavelet tree algorithms (extended variant): On the wavelet tree of sequence S, $\mathbf{acc}(0,i)$ returns $S[i]$; $\mathbf{rnk}(0,a,i)$ returns $\mathtt{rank}_a(S,i)$; and $\mathbf{sel}(a,j)$ returns $\mathtt{select}_a(S,j)$. For simplicity we have omitted the computation of $[\alpha_v,\omega_v)$, except on $\mathbf{sel}(a,j)$, where for simplicity we assume $C[a]$ refers to level $\ell = \lceil \lg \sigma \rceil$.

```
acc(ℓ, i)
  if ω_v − α_v = 0 then
    return α_v
  end if
  l ← rank_0(B̃_ℓ, C[α_v])
  z ← rank_0(B̃_ℓ, C[α_v]+i)
  if B̃_ℓ[i] = 0 then
    return
      acc(ℓ+1, z−l)
  else
    return
      acc(ℓ+1, i−(z−l))
  end if
```

```
rnk(ℓ, a, i)
  if ω_v − α_v = 0 then
    return i
  end if
  l ← rank_0(B̃_ℓ, C[α_v])
  z ← rank_0(B̃_ℓ, C[α_v] + i)
  if a ≤ 2^⌈lg(ω_v−α_v)⌉−1 then
    return
      rnk(ℓ+1, a, z−l)
  else
    return
      rnk(ℓ+1, a, i−(z−l))
  end if
```

```
sel(a, j)
  ℓ ← ⌈lg σ⌉
  d ← ⌈lg σ⌉ − ℓ + 1
  while ℓ ≥ 0 do
    if (a−1) mod 2^d = 0 then
      l ← rank_0(B̃_ℓ, C[α_v])
      j ← select_0(B̃_ℓ, l+j)
    else
      α_v ← α_v − 2^(d−1)
      l ← rank_1(B̃_ℓ, C[α_v])
      j ← select_1(B̃_ℓ, l+j)
    end if
    j ← j − C[α_v]
    ℓ ← ℓ − 1, d ← d + 1
  end while
  return j
```

positions with $\tilde{B}_\ell[i] = 0$ for $s_v \le i \le e_v$ will be mapped to consecutive positions $\tilde{B}_{\ell+1}[\mathtt{rank}_0(\tilde{B}_\ell, i)]$, and similarly with positions $\tilde{B}_\ell[i] = 1$. □

Figure 4.2 illustrates the wavelet matrix, where it can be seen that the blocks of the wavelet tree are maintained, albeit in different order. We now describe how to carry out the operations under the strict and the extended variants.

4.1.2.1 The Strict Variant

To carry out $\mathtt{access}(S,i)$, we first set i_0 to i. Then, if $\tilde{B}_0[i_0] = 0$, we set i_1 to $\mathtt{rank}_0(\tilde{B}_0, i_0)$. Else we set i_1 to $z_0 + \mathtt{rank}_1(\tilde{B}_0, i_0)$. Now we descend to level 1, and continue until reaching a leaf. The sequence of bits $\tilde{B}_\ell[i_\ell]$ read along the way form the value $S[i]$ (or, said another way, we maintain the interval $[\alpha_v, \omega_v)$ and upon reaching the leaf it holds $S[i] = \alpha_v$). Note that we have carried out only one binary $\mathtt{rank}$ operation per level, just as the standard wavelet tree.

Consider now the computation of $\mathtt{rank}_a(S,i)$. This time we need to keep track of the position i, and also of the position preceding the range, initially $p_0 = 0$. At each node v of depth ℓ, if $a \le 2^{\lceil \lg(\omega_v - \alpha_v)\rceil - 1}$, then we go "left" by mapping $p_{\ell+1}$ to $\mathtt{rank}_0(\tilde{B}_\ell, p_\ell)$ and $i_{\ell+1}$ to $\mathtt{rank}_0(\tilde{B}_\ell, i_\ell)$. Otherwise, we go "right" by mapping $p_{\ell+1}$ to $z_\ell + \mathtt{rank}_1(\tilde{B}_\ell, p_\ell)$ and $i_{\ell+1}$ to $z_\ell + \mathtt{rank}_1(\tilde{B}_\ell, i_\ell)$. When we arrive at the

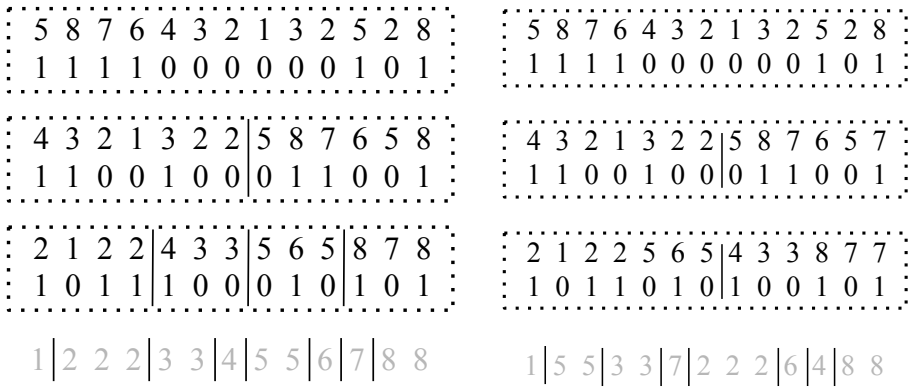


Figure 4.2: On the left, the pointerless wavelet tree (`WTNP`) of Figure 2.7.1. On the right, the wavelet matrix (`WM`) over the same sequence. One vertical line per level represents the position stored in the z_ℓ values.

leaf level, the answer is $i_\ell - p_\ell$. Note that we have needed one extra binary `rank` operation per original `rank` operation of the standard wavelet tree, instead of the two extra operations required by the (strict) pointerless variant.

Finally, consider operation $\texttt{select}_a(S, j)$. We first descend towards the leaf of a just as done for $\texttt{rank}_a(S, i)$, keeping track only of p_ℓ. When we arrive at the last level, p_ℓ precedes the range corresponding to the leaf of a, and thus we wish to track upwards position $j_\ell = p_\ell + j$. The upward tracking of a position $\tilde{B}_\ell[j_\ell]$ is simple: If we went left from level $\ell - 1$, then this position was mapped from a 0 in $\tilde{B}_{\ell-1}$, and therefore it came from $j_{\ell-1} = \tilde{B}_{\ell-1}[\texttt{select}_0(\tilde{B}_\ell, j_\ell)]$. Otherwise, position j_ℓ was mapped from a 1, and thus it came from $j_{\ell-1} = \tilde{B}_{\ell-1}[\texttt{select}_1(\tilde{B}_\ell, j_\ell - z_\ell)]$. When we arrive at the root bitmap, j_0 is the answer. Note that we have needed one extra binary `rank` per original binary `select` required by the standard wavelet tree. We remind that in practice `rank` is much less demanding, so this overhead is low. Algorithm 7 gives the pseudocode.

4.1.2.2 The Extended Variant

We can speed up `rank` and `select` operations if the array C that points to the starting positions of each symbol in its bitmap is available. First, we note that for $\texttt{rank}_a(S, i)$ we do not need anymore to keep track of p_ℓ, since all we need at the end is to return $i_\ell - C[a]$. Thus the cost becomes similar to that of the standard wavelet tree, which was not achieved with the extended variant of the pointerless wavelet tree.

Algorithm 7 Wavelet matrix algorithms (strict variant): On the wavelet matrix of sequence S, $\mathbf{acc}(0,i)$ returns $S[i]$; $\mathbf{rnk}(0,a,i,0)$ returns $\mathtt{rank}_a(S,i)$; and $\mathbf{sel}(0,a,j,0)$ returns $\mathtt{select}_a(S,j)$. For simplicity we have omitted the computation of $[\alpha_v,\omega_v)$.

```
acc(ℓ, i)
  if ω_v − α_v = 0 then
    return α_v
  end if
  if B̃_ℓ[i] = 0 then
    i ← rank_0(B̃_ℓ, i)
  else
    i ← z_ℓ + rank_1(B̃_ℓ, i)
  end if
  return acc(ℓ+1, i)
```

```
rnk(ℓ, a, i, p)
  if ω_v − α_v = 0 then
    return i − p
  end if
  if a ≤ 2^(⌈lg(ω_v − α_v)⌉ − 1) then
    p ← rank_0(B̃_ℓ, p)
    i ← rank_0(B̃_ℓ, i)
  else
    p ← z_ℓ + rank_1(B̃_ℓ, p)
    i ← z_ℓ + rank_1(B̃_ℓ, i)
  end if
  return rnk(ℓ+1, a, i, p)
```

```
sel(ℓ, a, j, p)
  if ω_v − α_v = 0 then
    return p + j
  end if
  if a ≤ 2^(⌈lg(ω_v − α_v)⌉ − 1) then
    p ← rank_0(B̃_ℓ, p)
    j ← sel(ℓ+1, a, j, p)
    return select_0(B̃_ℓ, j)
  else
    p ← z_ℓ + rank_1(B̃_ℓ, p)
    j ← sel(ℓ+1, a, j, p)
    return select_1(B̃_ℓ, j − z_ℓ)
  end if
```

For $\mathtt{select}_a(S,j)$ we can avoid the first downward traversal, as in the pointerless wavelet tree, and use the same technique to determine whether we came from the left or from the right in the parent bitmap. Once again, the cost becomes the same as in a standard wavelet tree, with no extra `rank` operations required. Algorithm 8 gives the detailed algorithm.

4.1.2.3 Construction

Construction of the wavelet matrix is simple. At the first level we keep in bitmap $\tilde{B}_0$ the highest bits of the symbols in S, and then stably sort S by those highest bits. Now we keep in bitmap $\tilde{B}_1$ the next-to-highest bits, and stably sort S by those next-to-highest bits. We continue until considering the lowest bit. This takes $O(n \lg \sigma)$ time.

Indeed, we can build the wavelet matrix almost in place, by removing the highest bits after using them and packing the symbols of S. This frees n bits, where we can store the bitmap $\tilde{B}_0$ we have just generated, and keep doing the same for the next levels. We generate the $o(n \lg \sigma)$-space indexes at the end. Thus the construction space is $n\lceil \lg \sigma \rceil + \max(n, o(n \lg \sigma))$ bits. Other more sophisticated techniques [CNS11, Tis11] may use even less space.

4.1.3 Pointerless Huffman Shaped Wavelet Trees

Here we revisit the technique [ZPYL08] to use canonical Huffman codes [SK64] to represent Huffman-shaped wavelet trees without pointers, this way removing the

Algorithm 8 Wavelet matrix algorithms (extended variant): On the wavelet matrix of sequence S, $\mathbf{acc}(0, i)$ returns $S[i]$; $\mathbf{rnk}(0, a, i)$ returns $\texttt{rank}_a(S, i)$; and $\mathbf{sel}(a, j)$ returns $\texttt{select}_a(S, j)$. For simplicity we have omitted the computation of $[\alpha_v, \omega_v)$, and in $\mathbf{sel}(a, j)$ we assume $C[a]$ refers to level $\ell = \lceil \lg \sigma \rceil$.

```
acc(ℓ, i)
  if ω_v − α_v = 0 then
    return α_v
  end if
  if B̃_ℓ[i] = 0 then
    i ← rank_0(B̃_ℓ, i)
  else
    i ← z_ℓ + rank_1(B̃_ℓ, i)
  end if
  return acc(ℓ+1, i)
```

```
rnk(ℓ, a, i)
  if ω_v − α_v = 0 then
    return i − C[a]
  end if
  if a ≤ 2^⌈lg(ω_v − α_v)⌉−1
  then
    i ← rank_0(B̃_ℓ, i)
  else
    i ← z_ℓ + rank_1(B̃_ℓ, i)
  end if
  return rnk(ℓ+1, a, i)
```

```
sel(a, j)
  ℓ ← ⌈lg σ⌉
  d ← ⌈lg σ⌉ − ℓ + 1
  j ← C[a] + j
  while ℓ ≥ 0 do
    if (a−1) mod 2^d = 0 then
      j ← select_0(B̃_ℓ, j)
    else
      j ← select_1(B̃_ℓ, j−z_ℓ)
    end if
    ℓ ← ℓ − 1, d ← d + 1
  end while
  return j
```

main component of the $O(\sigma \lg n)$ extra bits and retaining the advantages of reduced space and $O(\mathcal{H}_0(S) + 1)$ average traversal time.

The problem that arises when storing a standard Huffman-shaped wavelet tree in levelwise form is that a leaf that appears in the middle of a level leaves a "hole" that ruins the calculations done at the nodes to the right of it to find their position in the next level. Canonical Huffman codes (explained at Section 2.4.2) choose one of the many optimal Huffman trees that, among other interesting benefits [SK64, Sal07], yields a set of codes in which longer codes appear to the left of shorter codes.[2] Note that by interpreting the bit 0 as the right child and the bit 1 as the left child in a *canonical* Huffman encoding, we have that all the leaves at any level are the rightmost nodes. As a consequence, all the leaves of a level appear grouped to the right, and therefore do not alter the navigation calculations for the other nodes. The levelwise deployment of the tree can be seen as a sequence of "contiguous" bitmaps of varying length. Figure 4.3 illustrates the standard (`WTH`) and the levelwise (`WTHNP`) deployment of a canonical Huffman code.

The navigation procedures of Algorithm 5 can then be used verbatim, except for a few alphabet mappings that must be carried out: For $\texttt{access}(S, i)$, we need to maintain the Huffman tree so that, given the 0/1 labels of the traversed path, we determine the alphabet symbol corresponding to that leaf of the Huffman tree. For $\texttt{rank}_a(S, i)$, we need to convert the symbol a to its variable-length code, in order to follow the corresponding path in the wavelet tree. Finally, for $\texttt{select}_a(S, i)$, we need the same as for `rank` for the strict variant, or a pointer to the corresponding

[2] It is usually to the right, but this way is more convenient for us.

leaf area in some bitmap $\tilde{B}_\ell$, for the extended variant.

The mappings are also used to determine when to stop a top-down traversal. The mapping information amounts to $O(\sigma \lg n)$ bits as well, but it is much less in practice than what is stored for pointer-based wavelet trees, as explained. Moreover, in the case of canonical codes, $\sigma \lg \sigma + O(\sigma)$ bits are sufficient to represent the mappings. Note this space can be reduced to $O(\sigma \lg \lg n)$ bits using the technique proposed in Chapter 3.

The maximum number of levels in a Huffman tree is $O(\lg n)$, and it can be made $O(\lg \sigma)$ [BN13] by rebalancing deep leaves without affecting the asymptotic performance. Thus the pointers to the levels add up to a negligible $O(\lg^2 n)$ bits. The rest of the space is as for standard Huffman-shaped wavelet trees: $n(\mathcal{H}_0(S) + 1) + o(n(\mathcal{H}_0(S) + 1))$ bits. Moreover, by using the `RRR` compressed bitmaps of Section 2.6, the space is reduced to $n\mathcal{H}_0(S) + o(n(\mathcal{H}_0(S) + 1))$ bits, albeit in practice the navigation is slowed down.

4.2 The Compressed Wavelet Matrix

Just as on the pointerless wavelet tree, we can achieve zero-order entropy with the wavelet matrix by replacing the plain representations of bitmaps $\tilde{B}_\ell$ by the `RRR` compressed ones (see Section 2.6), the space becoming $n\mathcal{H}_0(S) + o(n \lg \sigma)$ bits. Compared to obtaining zero-order entropy using Huffman-shaped trees, this solution has several disadvantages, as explained: (1) the compressed bitmaps are slower to operate than in a plain representation; (2) the number of operations on a Huffman-shaped tree is lower on average than on a balanced tree; (3) the Huffman-shaped wavelet tree is more compact, as it reduces the redundancy from $o(n \lg \sigma)$ to $o(n(\mathcal{H}_0(S) + 1))$ (albeit a small $O(\sigma \lg n)$-bit space term is added to hold the Huffman model); (4) the bitmap compression can be additionally combined with the Huffman shape, obtaining further compression (yet higher time).

The idea is the same as in Section 2.7.2: Arrange the codes so that all the leaves are grouped to the right of the bitmaps $\tilde{B}_\ell$. However, because of the reordering of nodes produced by the wavelet matrix, the use of canonical Huffman codes does not guarantee that the leaves of the same level are contiguous. In the wavelet matrix, the position of a code c in $\tilde{B}_{\ell+1}$ depends only on the position of c in $\tilde{B}_\ell$ and on the bit of c in that level, $c[\ell]$. Figure 4.4 illustrates an example of a canonical set of codes where the first 16 shortest codewords take values from 00000 to 01111 and the remaining 32 from 100000 to 111110. The figure shows the relative positions of the codes at successive levels of the wavelet matrix for a sequence $\ldots c_8, c_{12}, c_{32}, c_{48} \ldots$, where $c_8 = 01000$, $c_{12} = 01100$, $c_{32} = 100000$, and $c_{48} = 110000$. As we can see, codes c_8 and c_{12} finish at level 5 but they are not contiguous since there is a c_{48} between them.

We require a different mechanism to design an optimal prefix-free code that guarantees that, under the shuffling rules of the wavelet matrix, all the leaves at any

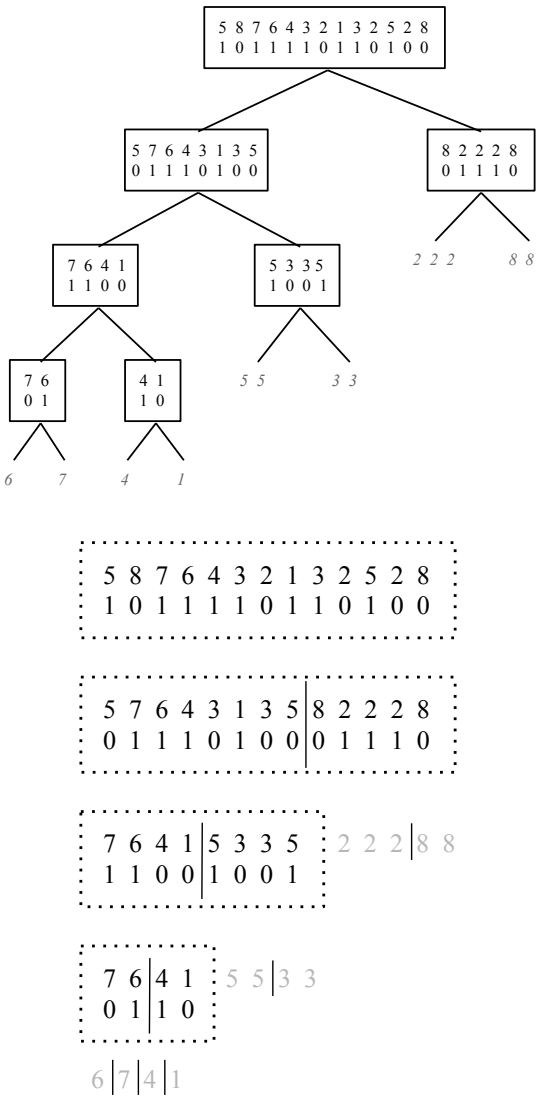


Figure 4.3: On the top, the pointer-based canonical Huffman wavelet tree (`WTH`) for our running example. On the bottom, its pointerless representation (`WTHNP`). Note that from now on we interpret the bit 0 as going right and the bit 1 as going left.

$$
\begin{array}{ll}
\ell = 1 & \ldots, c_8, c_{12}, c_{32}, c_{48} \ldots \\
\ell = 2 & \ldots, c_8, c_{12}, \ldots | \ldots, c_{32}, c_{48} \ldots \\
\ell = 3 & \ldots, c_{32} \ldots | \ldots, c_8, c_{12}, \ldots, c_{48} \ldots \\
\ell = 4 & \ldots, c_{32}, \ldots, c_8, \ldots, c_{48} | \ldots, c_{12}, \ldots \\
\ell = 5 & \ldots, c_{32}, \ldots, c_8, \ldots, c_{48}, \ldots, c_{12} \ldots | \ldots \\
\ell = 6 & \ldots, c_{32}, c_{48}, \ldots | \ldots
\end{array}
$$

Figure 4.4: Example of a sequence of canonical codes along wavelet matrix levels, showing that the leaves do not span a contiguous area. The the vertical bar "|" marks the points z_ℓ.

level form a contiguous area to the right of the bitmap.

We start by studying how the wavelet matrix sorts the codes at each level. Consider a pair of codes $c_1[1, \ell_1]$ and $c_2[1, \ell_2]$. Depending on their bits at a given level ℓ of the wavelet matrix, two cases are possible: (*a*) $c_1[\ell] = c_2[\ell]$ and then the relative positions of c_1 and c_2 stay the same at level $\ell + 1$, or (*b*) $c_1[\ell] \neq c_2[\ell]$ and then their relative positions in level $\ell + 1$ depend on the relation between $c_1[\ell]$ and $c_2[\ell]$. This yields the following proposition:

Proposition 2. *In a wavelet matrix, given any pair of codes c_1 and c_2, c_1 appears before(after) c_2 in $\tilde{B}_\ell$ if, for some $0 \leq i < \ell$, it holds $c_1[\ell - i, \ell - 1] = c_2[\ell - i, \ell - 1]$ and $c_1[\ell - i - 1] = 0(1) \neq c_2[\ell - i - 1]$.*

Proof. If $c_1[\ell - i, \ell - 1] = c_2[\ell - i, \ell - 1]$, then c_1 and c_2 transitively keep their relative positions from level $\ell - i$ to level ℓ. Instead, $c_1[\ell - i - 1] \neq c_2[\ell - i - 1]$ makes their ordering in level $\ell - i$ dependent only on how $c_1[\ell - i - 1]$ and $c_2[\ell - i - 1]$ compare to each other. □

As a second step, assume we want to design a set of fixed-length codes $\{c_a, a \in [0, \sigma)\}$ such that $c_a < c_b$ iff the area of c_a is before that of c_b in $\tilde{B}_{\lceil \lg \sigma \rceil}$. That is, we want the codes to be listed in order in the last level. Let $inv : \{0,1\}^{\mathbb{N}^+} \times \mathbb{N}^+ \to \{0,1\}^{\mathbb{N}^+}$ be defined as $inv(c[1, \ell], \ell) = c^{-1}[1, \ell]$, where $c^{-1}[i] = c[\ell - i + 1]$ for all $1 \leq i \leq \ell$. That is, $inv(c, \ell)$ takes number c as a codeword of ℓ bits and returns the code obtained by reading c backwards. Then, the following proposition holds:

Proposition 3. *Given any two values $i, j \in [0, \sigma)$ where $i < j$, code $inv(i, \lceil \lg \sigma \rceil)$ is located to the left of code $inv(j, \lceil \lg \sigma \rceil)$ in the bitmap $\tilde{B}_{\lceil \lg \sigma \rceil}$ of a wavelet matrix that uses such codes.*

Proof. Let $\tau_i = inv(i, \lceil \lg \sigma \rceil)$ and $\tau_j = inv(j, \lceil \lg \sigma \rceil)$. If τ_i and τ_j do not share any common suffix, then their relative positions in $\tilde{B}_{\lceil \lg \sigma \rceil}$ depend only on their last bit

and the relation is given by that bit. Otherwise, τ_i and τ_j share a common suffix of length $\lceil \lg \sigma \rceil - \delta + 1 \in [1, \lceil \lg \sigma \rceil]$, that is, $\tau_i[\delta, \lceil \lg \sigma \rceil] = \tau_j[\delta, \lceil \lg \sigma \rceil]$. Then, according to Proposition 2, τ_i is before τ_j iff $\tau_i[\delta] < \tau_j[\delta]$. In both cases the relation is given by the last distinct bit of the codes, or the first if they are read backwards. Since the codes are of the same length, comparing by the first distinct bit is equivalent to comparing numerically. That is, τ_i is before τ_j iff $inv(\tau_i, \lceil \lg \sigma \rceil) < inv(\tau_j, \lceil \lg \sigma \rceil)$. In turn, since $inv(inv(c, \ell), \ell) = c$, this is equivalent to $i < j$. □

The proposition gives a way to force a desired order in a set of fixed-length codes: Given symbols $a \in [0, \sigma)$, we can assign them codes $c_a = inv(a, \lceil \lg \sigma \rceil)$ to ensure that the areas become ordered in $\tilde{B}_{\lceil \lg \sigma \rceil}$. As a side note, we observe that we could have retained the symbol order natively in the wavelet matrix if we had chosen to decompose the symbols from their least to their most significant bit, and not the other way (in this case the wavelet matrix is actually radix-sorting the values). This brings problems in the extended variants, however, because the resulting range of codes has unused entries if σ is not a power of 2. For example, consider alphabet $0, 1, 2, 3, 4 = 000, \ldots, 100$; after reversing the bits we obtain numbers $0, 1, 2, 4, 6$, so we need to allocate 7 cells for C instead of 5. The size of C can double in the worst case. We cannot either directly use the idea of reversing the canonical Huffman codes, because the codes could not be prefix-free anymore. A more sophisticated scheme, based on Proposition 3, is required.

Assume we have obtained the desired code lengths ℓ_a, as well as the array *nCodes* from the canonical Huffman construction. We generate the final Huffman tree in levelwise order. The simplest description is as follows. We start with a set of valid codes $\mathcal{C} = \{0, 1\}$ and level $\ell = 1$. At each level ℓ, we remove from $\mathcal{C}$ the $nCodes[\ell]$ codes c with minimum $inv(c, \ell)$ value. The removed nodes are assigned to the $nCodes[\ell]$ symbols that require codes of length ℓ. Now we replace each code c remaining in $\mathcal{C}$, by two new codes, $c : 0$ and $c : 1$, and continue with level $\ell + 1$. It is clear that this procedure generates a prefix-free set of codes that, when reversed, satisfy that the codes finishing at a level are smaller than those that continue.

It is not hard to see that the total cost of this algorithm is linear. There are two kind of codes inserted in $\mathcal{C}$: those that will be chosen for a code and those that will not. There are exactly σ nodes of the first class, whereas for each node of the second class we insert other two codes in $\mathcal{C}$. Therefore the total number of codes ever inserted in $\mathcal{C}$ adds up to $O(\sigma)$. The codes to use at each level ℓ can be obtained by linear-time selection over the set of codes just extended (sorting codes by $inv(c, \ell)$), thus adding up to $O(\sigma)$ time as well.

Figure 4.5 gives an example of the construction.

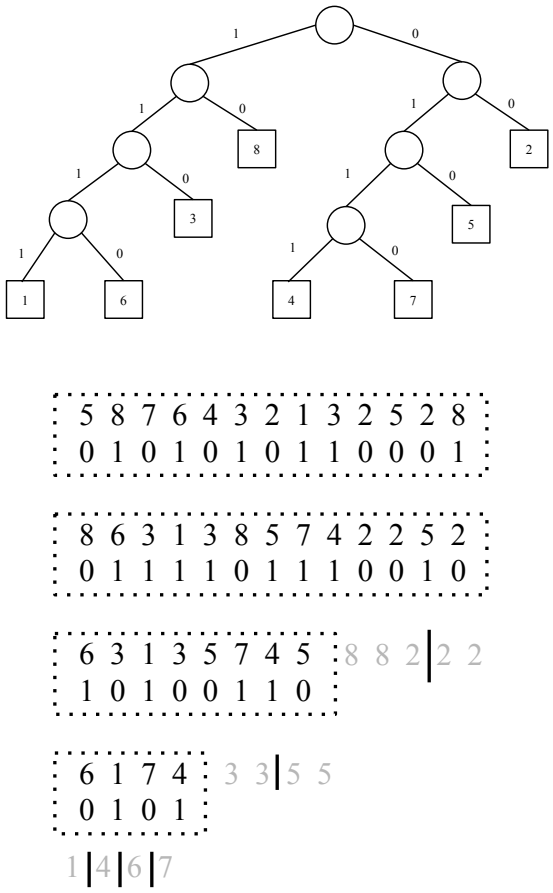


Figure 4.5: On the top, the Huffman tree resulting from our code reassignment algorithm on the running example. On the bottom, the resulting Huffman-shaped wavelet matrix.

4.3 Experimental Results

Our implementations build over the wavelet tree implementations of Libcds [Cla] a library implementing several space-efficient data structures. For each wavelet tree/matrix variant we present two versions, `CM` [Cla96, Mun96] and `RRR` [RRR07] (recall Section 2.6). The variants compared are the following:

- `WT`: standard pointer-based wavelet tree (Section 2.7.1);
- `WTNP`: the (extended) pointerless wavelet tree (Section 4.1.1);
- `WM`: the (extended) wavelet matrix (Section 4.1.2);
- `WTH`: the Huffman-shaped standard pointer-based wavelet tree (Section 2.7.2);
- `WTHNP`: the Huffman-shaped extended pointerless wavelet tree (Section 4.1.3);
- `WMH`: the Huffman-shaped or compressed (extended) wavelet matrix (Section 4.2);
- `AP`: the alphabet-partitioned data structure of Section 2.7.4, which is the best state-of-the-art alternative to wavelet trees.

These names are composed with the bitmap implementations by appending the bitmap representation name. For example, we call `WT.RRR` the standard pointer-based wavelet tree with all bitmaps represented with Raman's et al. (see Section 2.6) compressed bitmaps. `AP` uses always `CM` bitmaps, which is the best choice for this structure.

Note that all the pointerless structures use the array C. The extended versions generally achieve space very close to the strict ones and perform much faster.

4.3.1 Datasets

In order to evaluate the performance of `access`, `rank` and `select`, we use four different datasets:[3]

- `ESWiki`: Sequence of word identifiers generated by stemming the Spanish Wikipedia[4] with the Snowball algorithm. The sequence has length $n =$ 200,000,000, alphabet size $\sigma =$ 1,634,145, and zero-order entropy $\mathcal{H}_0 = 11.12$. This sequence can be used to simulate a positional inverted index [CN08, AGO10, GNP12, BFLN12].

[3] Left at `http://lbd.udc.es/research/ECWTLA`
[4] `http://es.wikipedia.org` dated 03/02/2010.

- `BWT`: The Burrows-Wheeler transform (BWT) [BW94] of `ESWiki`. The length and size of the alphabet, as well as the zero-order entropy, match those of `ESWiki`. However, `BWT` has a much lower high-order entropy [Man01]. Many full-text compressed self-indexes [FM05, FMMN07, NM07] use the BWT of the text they represent.

- `Indochina`: The concatenation of all adjacency lists of Web graph `Indochina-2004`, available at the WebGraph project.[5] The length of the sequence is $n =$ 100,000,000, the alphabet size $\sigma =$ 2,705,024, and the entropy is $\mathcal{H}_0 = 15.69$. This representation has been used to support forward and backward traversals on the graph [CN08, CN10a].

- `INV`: Concatenation of inverted lists for a random sample of 2,961,510 documents from the English Wikipedia.[6] This sequence has length $n =$ 338,027,430 and its alphabet size is $\sigma =$ 2,961,510. From this sequence we extract the first $n =$ 180,000,000 elements with an alphabet of size $\sigma =$ 1,590,398 and an entropy of $\mathcal{H}_0 = 19.01$. This sequence has been used to simulate document inverted indexes [NP10, GNP12].

4.3.2 Measurements

To measure performance we generated 100,000 inputs for each query and averaged their execution time, running each query 10 times. The $\texttt{access}(S, i)$ queries were generated by choosing positions i uniformly at random in $[1, n]$. Queries $\texttt{rank}_a(S, i)$ were generated by choosing i uniformly at random, and then setting $a = S[i]$. Each $\texttt{select}_a(S, j)$ query was generated by first choosing a position i at random in $[1, n]$, then setting $a = S[i]$, and finally choosing j at random in $[1, \texttt{rank}_a(S, n)]$. The resulting distribution is the most common in applications, and it obtains the $O(\mathcal{H}_0(S) + 1)$ average time performance in the Huffman-shaped variants.

4.3.3 Results on Sequences

Figures 4.6 to 4.8 show the time and space for the different data structures and configurations for `access`, `rank` and `select` queries. The black vertical bar on the plots shows the value of $\mathcal{H}_0$. The bitmaps are parametrized by setting their sampling values to 32, 64, and 128. In the case of `AP`, these bitmap samplings are combined with permutation samplings 4, 16, and 64, respectively, and all are run with $\ell_{min} = 10$, as in previous work [BCG+14].

[5] `http://law.dsi.unimi.it`
[6] `http://en.wikipedia.org`

4.3.3.1 Space

We start by discussing the space usage, which we measure in bits per symbol (bps). First we note that the WM variants use always the same space as the corresponding WTNP variants (while being faster, as we discuss soon). The space of WTNP.CM and WM.CM is obviously close to $\lceil \lg \sigma \rceil$ bps. The extra space incurred by WT.CM is the overhead of the wavelet tree pointers, and is roughly proportional to σ/n (times some implementation-dependent constant). This amounts to nearly 4 bps in ESWiki and BWT, but 3.5 times more (14 bps) in Indochina, as expected from its larger alphabet size, and again 4 bps in INV. On the other hand, the space of WTHNP.CM and WMH.CM is always close to $\mathcal{H}_0$ bits per symbol, plus a small extra to store the Huffman model. The space overhead of WTH.CM on top of those corresponds, again, to the wavelet tree pointers.

The sampling parameter affects more sharply the RRR variants, as they store more data per sample. The difference between WTNP.RRR or WM.RRR and WT.RRR is also proportional to σ/n, but this time the constant is higher because the RRR implementation needs more constants to be stored per bitmap (i.e., per wavelet tree node). Thus the penalty is 6 bps on ESWiki and BWT, 21 bps (3.5 times more) on Indochina, and 7 bps on INV. The same differences can be observed between WTHNP.RRR or WMH.RRR and WTH.RRR . We return later to the fact that WMH.RRR takes more space than WTHNP.RRR on Indo and INV.

Finally, how WTNP.RRR/WM.RRR and WTHNP.RRR/WMH.RRR compare to WTHNP.CM/WM.CM depends strongly on the type of sequence. In general, RRR compression achieves the zero-order entropy as an upper bound, but it can reach much less when the sequence has local regularities. On the other hand, RRR representation poses an additive overhead of 27% of $\lg \sigma$, which corresponds to the $o(n \lg \sigma)$ overhead in this implementation [CN08]. When combining Huffman and bitmap compression, this 27% overhead acts over $\mathcal{H}_0$ and not over $\lg \sigma$, which brings it down, but on the other hand we must add the overhead of storing the Huffman model. On ESWiki, which has no special properties, the 27% overhead is around 5.7 bps, showing that RRR compression reaches around 8.3 bps, well below $\mathcal{H}_0$. When combining with Huffman compression, this overhead becomes 14%, that is, nearly 3 bps. Added to the 8.3 bps and to the 1 bps of the Huffman model overhead, we still get slightly more space than plain Huffman compression, which is the best choice and reaches only 10% overhead over the zero-order entropy.

The picture changes when we consider BWT. The Burrows-Wheeler transform of ESWiki boosts its higher-order compressibility [Man01], which is captured by RRR compression [MN07a], making RRR compression reach the same space of Huffman compression, despite its 27% space overhead. When combining both compressions, the result breaks the zero-order entropy barrier by more than 10% and becomes the best choice in terms of space.

RRR gives another surprising result on Indochina and INV, where bitmap compression alone is more space-effective than in combination with Huffman

compression, and breaks the zero-order entropy by a large margin. This cannot be explained by high-order compressibility, as in this case the combination with Huffman would not harm. This behavior corresponds to the special nature of these sequences: the adjacency lists of the graph and the inverted lists are sorted in increasing order. Long increasing sequences induce long runs of 0s and 1s in the bitmaps of the wavelet trees and matrices. Those are retained in deeper levels when our partitioning by the most significant bit is used.[7] The Huffman algorithm, instead, combines the nodes in unpredictable ways and destroys those long runs. Still, our Huffman algorithm maintains the order between those symbols whose codewords have the same length, and thus the impact of this reordering is not as high as it could be. Instead, the Huffman wavelet matrix completely reshuffles the symbols. As a result, for example, the space of `WMH.RRR` exceeds that of `WTHNP.RRR` by around 5 bps on `Indo` and 6–7 bps on `INV`.

4.3.3.2 Time

The time results are rather consistent across collections. Let us first consider operation `access`. If we start considering the variants that do not use Huffman compression, we have that `WT.RRR` is about 10%–25% slower than `WT.CM` , which is explained by a more complex implementation [CN08]. Instead, the pointerless variant, `WTNP.CM` , is 20%–25% slower (recall that, in their extended variant, these require twice the number of `rank` operations, but locality of reference makes them faster than twice the cost of one `rank` operation). However, `WTNP.RRR` is about 40% slower than `WT.RRR` as the `rank` operation is slower and its higher number impacts more on the total time (but still locality of reference makes the percentage much less than 100%). The wavelet matrix, instead, carries out the same number of `rank` operations than the pointer-based wavelet tree, so this time penalty disappears. Actually, `WM.CM` is 8%–14% *faster* than `WT.CM` , and `WM.RRR` is up to 4% faster than `WT.RRR` This may be due to less memory usage, which increases locality of reference. Finally, the use of Huffman compression improves times by about $\mathcal{H}_0 / \lg \sigma$, as expected: times are reduced to about 50%–60% on `ESWiki` and `BWT`, to about 65%–85% on `Indochina`, and there is almost no reduction on `INV`.

The situation is basically the same for operation `rank`, as expected from the algorithms. The times are usually slightly lower because it is not necessary to access the bitmaps as we descend. The use of the wavelet matrix still gives essentially the same time (and even slightly faster) than a pointer-based wavelet tree, and the use of Huffman-shaped trees reduces the times by the same factors as for `access`, as expected.

The times of operation `select` show less difference between the standard and the pointerless variants, because performing one extra `rank` operation is less relevant compared to the original (slower) `select` operation on the bitmaps. One can see

[7] This is another advantage over using the least significant bit, which would break the runs faster.

that WTNP.CM is 30%–40% slower than WT.CM and that WTNP.RRR is 35%–50% slower than WT.RRR The difference between plain and compressed bitmaps does not vary much, on the other hand: WT.RRR is 25%–30% slower than WT.CM What is more surprising is that the wavelet matrix is clearly slower than the pointer-based wavelet trees: WM.CM is 10%–15% slower than WT.CM and WM.RRR is 20%–30% slower than WT.RRR The reason is that the implementations of `select` [GGMN05, CN08] proceed by binary search on the sampled values, thus their cost has in practice a component that is logarithmic on the bitmap length. The bitmaps on the wavelet tree nodes are shorter than n, whereas in the wavelet matrix (and the pointerless wavelet tree) they are always of length n. Indeed, the wavelet matrix is faster than the pointerless wavelet tree: WM.CM is 20%–25% faster than WTNP.CM and WM.RRR is 12%–15% faster than WTNP.RRR Once again, the use of Huffman reduces all the times by about the same space fraction obtained by zero-order compression.

4.3.3.3 Bottom Line

On ESWiki, where zero-order compression is the dominant space factor, our Huffman-shaped wavelet matrix, WMH.CM , obtains the best space (only 10% off the zero-order entropy) and the best time, by a good margin.

On BWT, where higher-order compression is exploited by RRR, the space-time tradeoff map is dominated by the combination of WMH.RRR (minimum space) and WMH.CM (minimum time), the two variants of our Huffman-shaped wavelet matrix. The former breaks the zero-order entropy barrier by about 10%.

On Indochina and INV, where RRR achieves space gains that are only degraded by Huffman compression, the dominant techniques are variants of the wavelet matrix: WM.RRR (least space) and WMH.CM (least time). The former takes about 75% of the zero-order entropy.

Summarizing, the wavelet matrix variants obtain the same space of the pointerless wavelet trees, but they operate in about 65% of their time, reaching basically the same performance of the pointer-based variants but much less space. As a result, they are always the dominant technique. Which variant is the best, WMH.CM , WMH.RRR or WM.RRR , depends on the nature of the collection.

The comparison with AP is interesting. In collections similar to ESWiki, Barbay et al. [BCG$^+$14] show that AP generally achieves the best space and time among the alternatives WTNP.RRR , WTNP.CM , WT.CM , and WT.RRR , thus becoming an excellent choice in that group. The pointerless Huffman-shaped alternatives, however, clearly outperform AP in space: pointerless Huffman compression, and in particular Huffman wavelet matrices, improve upon the old wavelet tree alternatives in both space and time, using much less space than AP. Still, AP is a faster representation, only slightly faster in operations `access` and `rank`, and definitely faster in operation `select`. The other collections also demonstrate that wavelet trees and matrices can exploit other compressibility features of the sequences apart from $\mathcal{H}_0$, whereas AP is blind

to those (this is also apparent in their experiments [BCG+14], even using the basic wavelet tree variants).

Note that, for simplicty and uniformity, we have built our prototypes and experiments on the implementations of Libcds. Other ones could be tried. For example, there exists an alternative `RRR` implementation briefly described in Section 2.6 [NP12] that, at the price of some (further) increase in time, reduces the 27% space overhead to 10% on a 64-bit architecture. This would multiply the space of all the `RRR` alternatives by up to 0.86, making them even more space-attractive (and making `WMH.RRR` the most space-efficient choice on `ESWiki`). On the other collections, our conclusions above would not change. Moreover, on those collections the space fraction is likely to be higher than 0.86, as these benefit from locality in compression. The alternative implementation [NP12] captures zero-order entropy of 63-bit chunks, and thus it is less local than the implementation we used, which takes 15-bit chunks. As another example, one could add more efficient implementations of bitwise `select` [OS07, Vig08, NP12], which would make the impact of the better wavelet matrix organization more clear.

4.4 Discussion

The pointerless wavelet tree [MN06, MN07b], designed to avoid the $O(\sigma \lg n)$ space overhead of standard wavelet trees [GGV03], was unnecessarily slow in practice. We have redesigned this data structure so that its time overhead over standard wavelet trees is significantly lower. The result, dubbed *wavelet matrix*, enjoys all the good properties of pointerless wavelet trees but performs significantly faster in practice. It requires $n \lg \sigma + o(n \lg \sigma)$ bits of space, and can be built in $O(n \lg \sigma)$ time and almost in-place. We have also adapted pointerless Huffman-shaped wavelet trees to become Huffman-shaped wavelet matrices. This required a nontrivial redesign of the variable-length code assignment mechanism. Our experimental results show that the compressed wavelet matrix dominates the space/time tradeoff map for all the real-life sequences we considered, also outperforming in most cases other structures designed for large alphabets [BCG+14].

Dynamic wavelet trees [MN08, HM10, NS14] can immediately be translated into wavelet matrices. It would be interesting to consider newer, theoretically more efficient dynamic versions [NN14], and obtain practically efficient implementations over wavelet matrices.

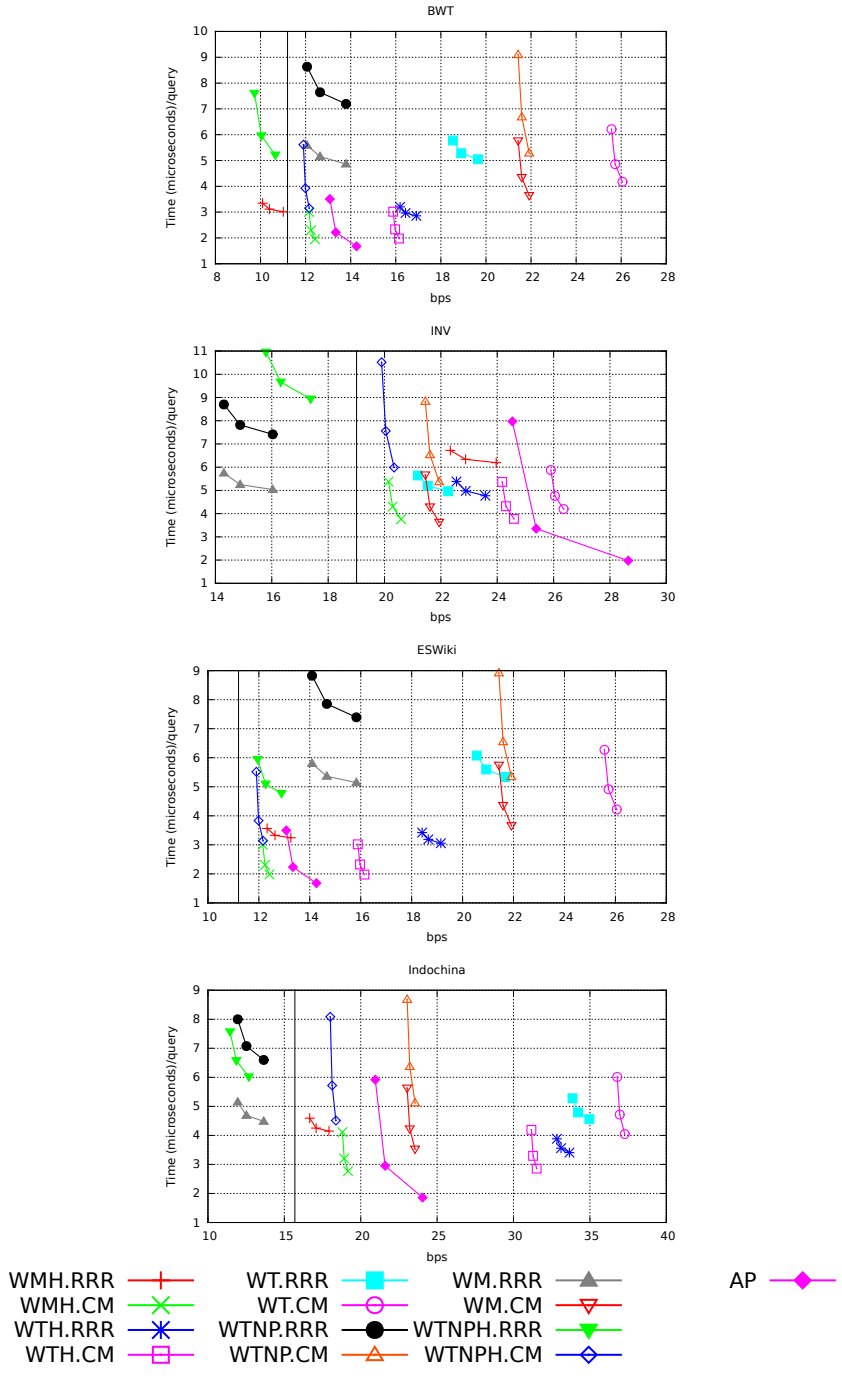


Figure 4.6: Running time per `access` query over the four datasets.

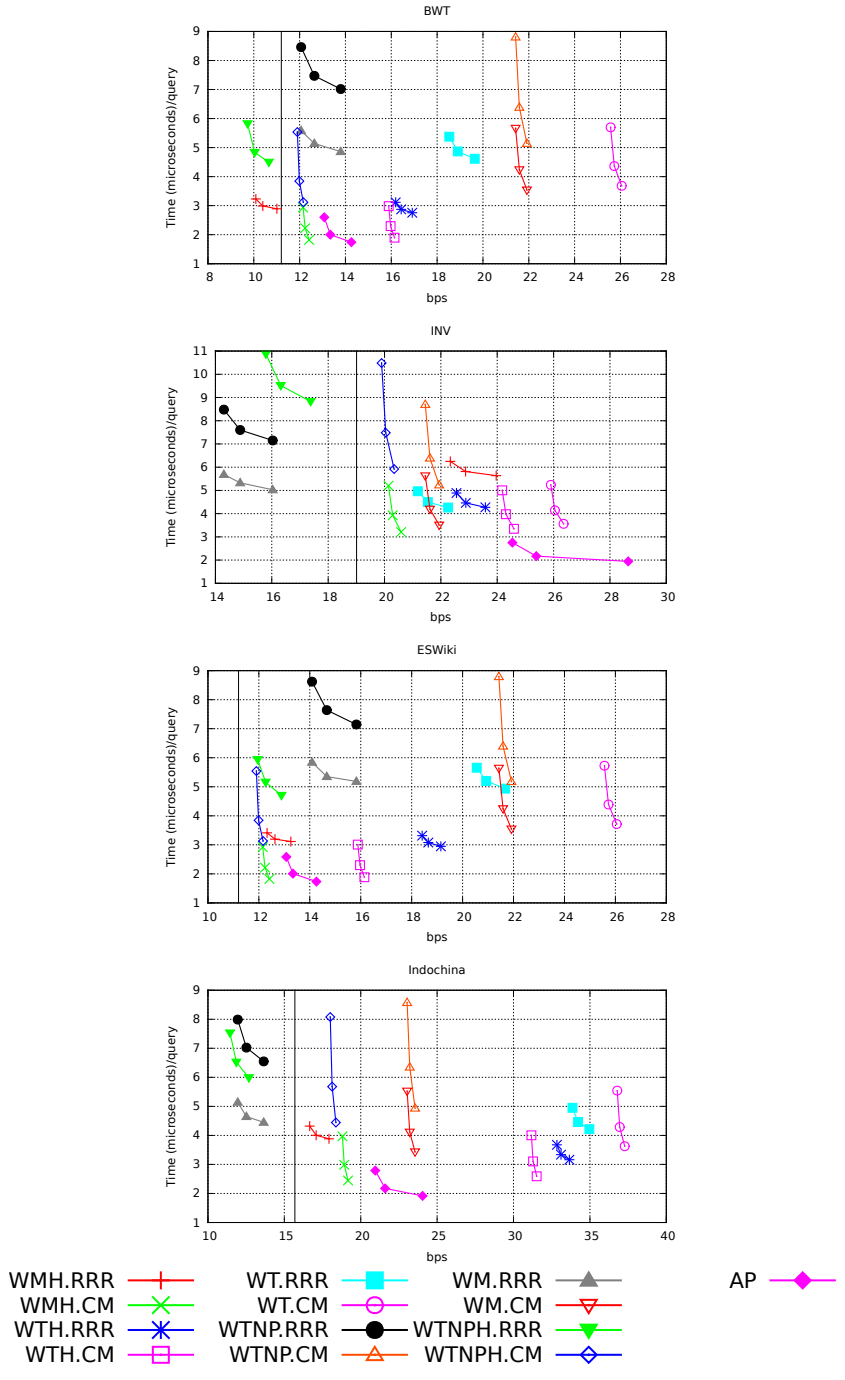


Figure 4.7: Running time per `rank` query over the four datasets.

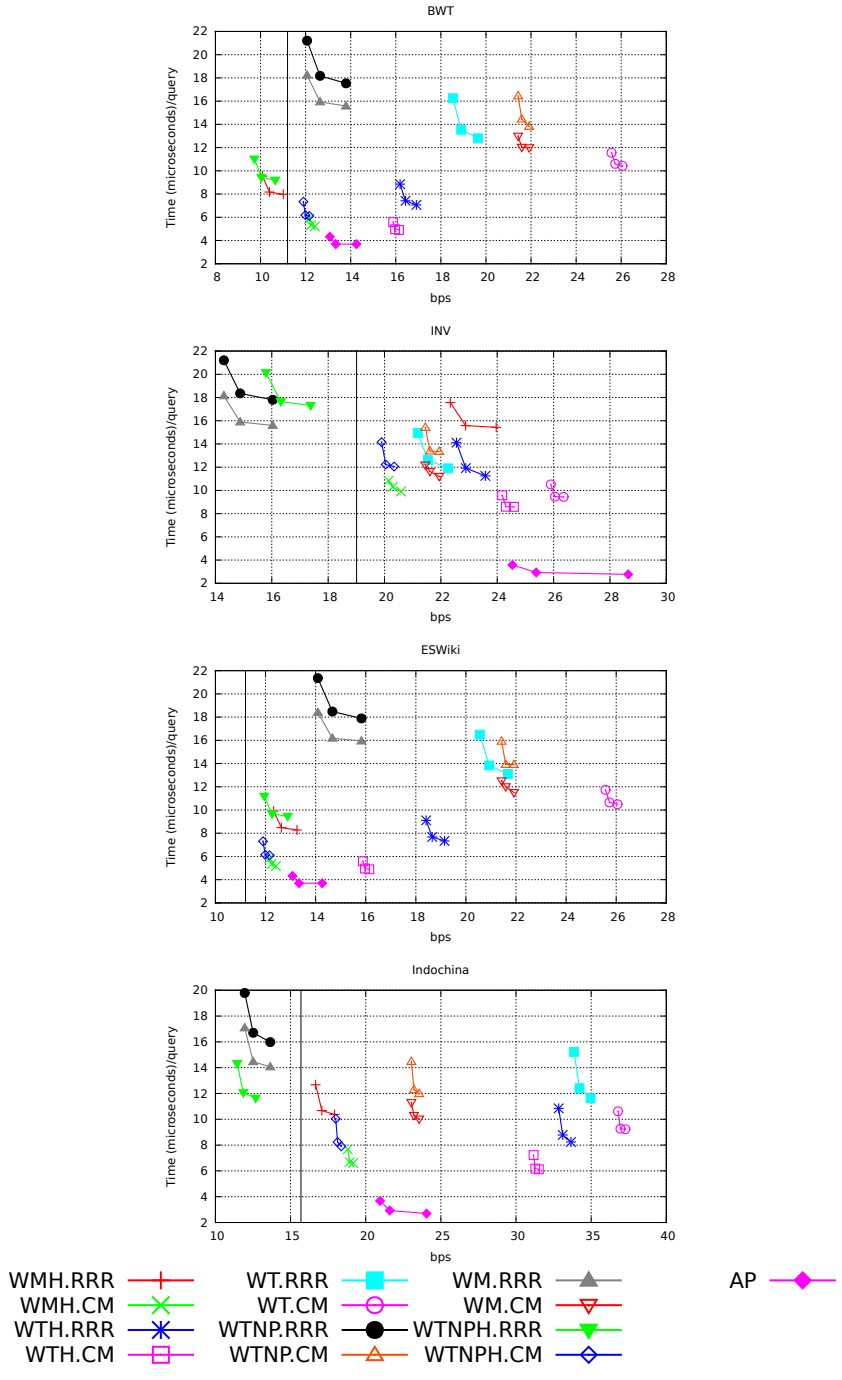


Figure 4.8: Running time per `select` query over the four datasets.

Part II

Repetition-based Data Structures

Chapter 5

Previous Concepts on Repetitive Scenarios

5.1 Why Repetition-based Data Structures?

Given a sequence $S[1, n]$, we define the concept of *repetition* as a segment $S[i, j]$ that appears more than once in S. Although the *empirical entropy* models are adequate for a wide number of applications, as shown in the first part of this thesis, it has proven to be ineffective in highly repetitive scenarios, that is, when dealing with sequences with many repetitions. This is because if one concatenates a sequence S with itself, resulting in SS, the statistical properties of SS do not vary with regard to that of S. In particular, $\mathcal{H}_0(S) = \mathcal{H}_0(SS)$ since the probability of occurrence of each symbol are exactly the same. In case of *high order entropy* $\mathcal{H}_k$ ($k = o(\log_\sigma n)$), the situation does not improve significantly since, in most cases, repetitions are much father apart thank $k = o(\log_\sigma n)$.

Repetitive sequences arise in many fields, from software or document repositories, to DNA datasets. For instance, DNA analysis is a trend nowadays, and being able to space-efficiently represent these kind of sequences at the same time of carrying out complex queries on them is fundamental. However, most DNA collections have empirical entropy close to 2 bps, as they typically have only for bases (A, G, C, T), which make them practically non-statistically compressible[1]. Fortunately, DNA collections are formed by the sequenced genomes of many individuals of the same species, which make these sequences *highly repetitive* since it is known that for two human genomes share 99.5% to 99.9% of their sequences [JW04, TK04].[2] Statistical compression does not take proper advantage of repetitiveness [KN13], but other

[1] See, for example, `http://pizzachili.dcc.uchile.cl/texts.html`.

[2] This number may be even higher on individuals of the same geographic area, for example. There is always controversy about this number and on how it is measured, however.

techniques like grammar or Lempel-Ziv compression do [Nav12].

This part of the thesis presents several data structures that are able to exploit this new kind of repetitiveness, while solving the same operations than their statistical or compact counterparts but being orders of magnitude more space-efficiently. Lower bounds, however, show that they are also bound to be slower than the representations based on statistical compression.

5.2 Kolmogorov Complexity

Suppose we are given a random variable X with mass probability function $P\{X = x\} = p(x)$. Then, and according to Shannon (see Section 2.2), we need $\lceil \lg \frac{1}{p(x)} \rceil$ bits to describe the event $X = x$. This can also be seen as the *descriptive* complexity of X according to Shannon or as a statistical description of X.

An alternative way of describing an object X (or a random variable X) was proposed by the Russian mathematician Kolmogorov [Kol65, Kol68]. He went further by defining the *algorithmic descriptive complexity* of an object X (also known as Kolmogorov complexity $K(X)$) as the length of the smallest computer program that generates X. This definition is computer independent, which means that no matter which language we use to create that description, by assuming an additive penalty, we can rewrite the program with a different language. The main drawback of this definition of complexity is that it is not computable!

Trying to find effectively computable descriptions that go beyond Shannon's definition and get close to that of Kolmogorov, several techniques were proposed. Typical examples are all kinds of *LZ*-parsings and grammar-compression techniques, which are explained next.

5.3 Lempel-Ziv Parsings

Lempel-Ziv parsings, *LZ*-parsings, or *LZ*-factorizations are different names for the same concept. An *LZ*-parsings is a dictionary-based technique in which an input sequence or string is parsed into a sequence of *phrases*, each of them belonging to a *dictionary*. Depending on how the parsing is carried out, different variations arise, although all of them rely on the same principle: Find a previous occurrence of an string in the sequence (called the *source*) and replace the new copy of the source by a backpointer to it.

An important property of any *LZ* parsing is the *transitive depth*, which is a measure of the nesting in the parsing. Concretely, the *transitive depth* is an upper bound on how many times we have to follow backpointers in the dictionary to obtain a symbol.

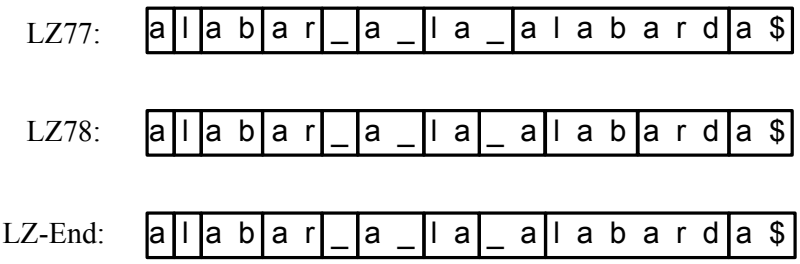


Figure 5.1: Example of an *LZ77*, *LZ78*, and *LZ-End* parsing for an input sequence *T=alabar_a_la_alabarda$*.

Among all possible variants, *LZ77* [ZL77], *LZ78*[3][ZL78], and *LZ-End* [KN13] are probably the most interesting.

5.3.1 LZ77

The *LZ77* [ZL77] parsing of a sequence $S[1,n]$ over $\Sigma = [1,\sigma]$ is a sequence of phrases $Z[1,z]$ such that $S = Z[1]Z[2]\dots Z[z]$. The parsing is defined by induction as follows. Let $Z[1] = S[1]$. Suppose we have already processed $S[1,i-1]$ obtaining $Z[1,p-1]$. Then, we have to find the longest prefix $S[i,i'-1]$ of $S[i,n]$ that occurs in $S[1,i-1]$. We then set $Z[p] = T[i,i']$, $p = p+1$, and $i = i'+1$. Figure 5.1 shows an example of an *LZ77* parsing.

For each *phrase* in the dictionary we need to store a backpointer to its *source*, which requires $\lg n$ bits, plus the trailing symbol of the current *phrase*, which needs $\lg \sigma$. Thus, the size of the representation of an *LZ77* dictionary is $|LZ77(S)| = z(\lg n + \lg \sigma)$ bits. This solution obtains the best space performance on repetitive collections, and at the same time it is very efficient at decompression. However, it is slow for supporting more complex operations. For instance, `extract`(S,i,j) takes $O((j-i+1)h)$ time, being h the *transitive depth* of the parsing. No solutions for `rank` and `select` are known on top of an *LZ77*-compressed sequence.

5.3.2 LZ78

The *LZ78* [ZL78] parsing of a sequence $S[1,n]$ is a sequence of phrases $Z[1,z]$ such that $S = Z[1]Z[2]\dots Z[z]$. The parsing is defined by induction as follows. Let $Z[1] = S[1]$. Suppose we have already processed $S[1,i-1]$ obtaining $Z[1,p-1]$. Then, we have to find the longest prefix $S[i,i'-1]$ of $S[i,n]$ that is a phrase $Z[q]$,

[3]Although *LZ78* belongs to the Lempel-Ziv family, it is also considered a *grammar* compression technique.

$q < p$. We then set $Z[p] = T[i, i']$, $p = p + 1$, and $i = i' + 1$. Figure 5.1 shows an example of an *LZ78* parsing.

This kind of parsing introduces a serious restriction with regard to *LZ77*: It can only add new *phrases* to the dictionary if the *phrase* already exists in the dictionary. This implies we cannot point back to any position in the input sequence but to just to a restricted set of them. Therefore, the number of *phrases* of an $z_{LZ78} \geq z_{LZ77}$, an typically $z_{LZ78} >> z_{LZ77}$. We can store pointers back with just $O(z \lg z)$ bits, becoming the total space of an *LZ78* parsing $O(z(\lg z + \lg \sigma))$ bits.

5.3.3 LZ-End

The *LZ-End* [KN13] parsing of a sequence $S[1, n]$ is a sequence of phrases $Z[1, z]$ such that $S = Z[1]Z[2] \dots Z[z]$. The parsing is defined by induction as follows. Let $Z[1] = S[1]$. Suppose we have already processed $S[1, i - 1]$ obtaining $Z[1, p - 1]$. Then, we have to find the longest prefix $S[i, i' - 1]$ of $S[i, n]$ that is also a suffix of $Z[1] \dots Z[q]$, for some $q < p$. We then set $Z[p] = T[i, i']$, $p = p + 1$, and $i = i' + 1$. Figure 5.1 shows an example of an *LZ-End* parsing.

The difference between *LZ77* and *LZ-End* is that the second forces the sources to end where a previous source does. This is again a restriction that makes $z_{LZ\text{-}End} \geq z_{LZ77}$, but faster in practice for $\texttt{extract}(S, i, j)$. The space to store an *LZ-End* dictionary is $z(\lg z + \lg \sigma)$, although the authors proposed a more sophisticated representation that takes $z(\lg \sigma + \lg n) + o(n)$ bits [KN13], and supports more efficient extraction. Concretely, $\texttt{extract}(S, i, j)$ is carried out in optimal $O(j - i + 1)$ time if j matches a source ending, and $O((j - i + 1) + h)$ otherwise, being h the *transitive depth* of the parsing. Again, no solution is known that supports `rank` and `select` on top of an *LZ-End*-compressed sequence.

5.4 Grammar Compression

Grammar-compressing a sequence S means to find a context-free grammar that generates S and only S. Finding the smallest grammar $\mathcal{G}(S)^*$ that generates a given sequence S is an NP-complete problem [CLL+05]. Even if it were easy to obtain, the smallest grammar $\mathcal{G}(S)^*$ is never smaller than an *LZ77* factorization of the same input [Ryt03]. This is because, in an *LZ77* parsing, any substring preceding the current position may act as a source. However, in grammar compression, we can only create new rules with rules that already belong to the grammar (as in an *LZ78*). This is a limitation with regard to *LZ77* that results in worse space performance.

Even though grammar compression techniques are inferior to *LZ77* encodings, they are preferable in many scenarios since they are more tractable and permit to solve more complex operations than *LZ77* encodings. There exist in the literature many algorithms to convert an *LZ77* factorization of a sequence into a context-free grammar. As an example, Rytter [Ryt03] proposed an algorithm to obtain a

grammar $\mathcal{G}(S)$ whose space is an $O(\lg n)$-approximation of an *LZ77* parsing of the same sequence, that is, $|\mathcal{G}(S)| = O(|LZ77(S)| \lg n)$.

Beyond *LZ77*-reductions, there exist also several proper grammar compression algorithms like *LZ78* [ZL78], Sequitur [NMWM94], or *RePair* [LM00] which, despite of obtaining grammars larger than $\mathcal{G}^*$, they perform very well in practice. In particular, *RePair* [LM00] is able to compute a context-free grammar for a sequence $S[1,n]$ in $O(n)$ time, obtaining competitive space performance both in classical and repetitive scenarios.

Basically, given a sequence $S[1,n]$ over $\Sigma = [1,\sigma]$, *RePair* [LM00] finds the most frequent pair of symbols ab in S, adds a rule $X \to ab$ to a dictionary R, and replaces each occurrence of ab in S by X (X is known as a nonterminal symbol of the grammar). This process is repeated (X can be involved in future pairs) until the most frequent pair appears only once. The result is a tuple (R, C), where the dictionary R contains $r = |R|$ rules and C, of length $c = |C|$, is the final reduction of S after all the replacements are carried out. Figure 5.2 shows an example of applying *RePair* on a binary input S. Note that C is drawn from an alphabet of size $\sigma + r$ (σ terminal plus r nonterminal symbols), not only σ. Thus, the total output size of (R, C) is $(2r + c)\lg(r + \sigma)$ bits.

Given a grammar (R, C) and a rule $X \to YZ$ with $X \in R$ and $Y, Z \in \Sigma \cup R$, the *expansion tree* of X, $T(x)$, is an ordinal tree with root at X and two children: $T(Y)$ and $T(Z)$. If X is a terminal symbol, then $T(X)$ is defined as a single node containing the terminal X. Figure 5.3 shows an example of an expansion tree of a grammar rule. The *grammar tree* of a grammar (R, C) is an ordinal tree with a virtual node and $|C|$ children, each containing the expansion tree of $C[i]$, $i \in [1, |C|]$. A previous work of Sakamoto [Sak05] gave efficient algorithms to generate balanced grammars, that is, whose grammar tree is of height $O(\lg n)$.

A *RePair* implementation that typically gives good practical performance can be found at `http://www.dcc.uchile.cl/ gnavarro/software/re pair.tgz`. It also includes a variant that in most cases generates a balanced grammar.

5.5 Rank, Select, and Access on Repetitive Scenarios

Although there exist plenty of data structures to solve `rsa` queries while obtaining zero-order compression, the number of solutions to solve these kind of queries on highly repetitive scenarios is much more limited. In fact, only Bille et al. [BLR+11] showed how to represent S using $O(g \lg n)$ bits so that `access` is solved in $O(\lg n)$ time (g is the size of the grammar). This time is essentially optimal [VY13]: any structure using $g^{O(1)} \lg n$ bits requires $\Omega(\lg^{1-\epsilon} n / \lg g)$ time for `access`, for any $\epsilon > 0$. If S is not very compressible and $g = \Omega(n^\alpha)$ for some constant α, then the time is $\Omega(\lg n / \lg \lg n)$ for any structure using $O(n \operatorname{polylog} n)$ bits.

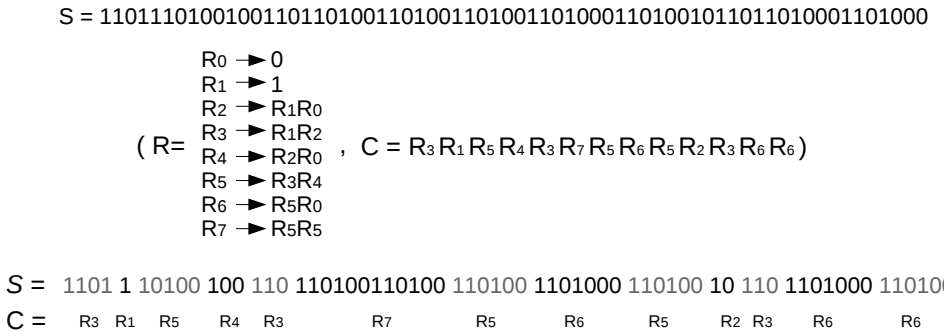


Figure 5.2: The data structures (R, C) are the result of executing the *RePair* algorithm on the input sequence S and $\sigma = 2$.

Although this solves the problem for `access`, we are not aware of any solution that solves `rsa` queries beyond that of Navarro et al. [NPV14], which uses $O(g\sigma \lg n)$ bits and solves queries in $O(\lg \sigma \lg n)$ time. This solution is of practical nature and is basically the state of the art of practical grammar compression techniques for `rsa` queries. The authors showed [NPV14] how to support `rsa` queries on bitmaps, and then how to combine this solution with others already known to support `rsa` queries on larger alphabets. How it works is explained next.

5.5.1 Rank, Select, and Access on Repetitive Bitmaps

Given a repetitive binary sequence $B[1, n]$, Navarro et al. [NPV14] *RePair*-compress B with a balanced grammar and enhance the output (R, C) with extra information to solve `rsa` queries. For each rule $X \in R$, let $exp(X)$ be the string of terminals X expands to. Then, they store two numbers per nonterminal X:

- $\ell(X) = |exp(X)|$.
- $z(X) = \texttt{rank}_0(exp(X), \ell(X))$ (the number of 0s in $exp(X)$).

Note that these values can be recursively computed as $\ell(X) = \ell(Y) + \ell(Z)$, with $\ell(0) = \ell(1) = 1$; $z(X) = z(Y) + z(Z)$, with $z(0) = 1, z(1) = 0$; and $exp(X) = exp(Y)exp(Z)$.

To save space, they store $\ell(\cdot)$ and $z(\cdot)$ only for a subset of nonterminals, and compute the others recursively by partially expanding the nonterminal. Given a parameter δ, they guarantee that, to compute any $\ell(X)$ or $z(X)$, we have to expand at most 2δ rules. The sampled rules are marked in a bitmap $B_d[1, r]$ and the sampled values are stored in two vectors, S_ℓ and S_z, of length $\texttt{rank}_1(B_d, r)$. To obtain $\ell(X)$ we check whether $B_d[X] = 1$. If so, then $\ell(X) = S_\ell[\texttt{rank}_1(B_d, X)]$. Otherwise $\ell(X)$ is obtained recursively as $\ell(Y) + \ell(Z)$. The process for $z(X)$ is analogous.

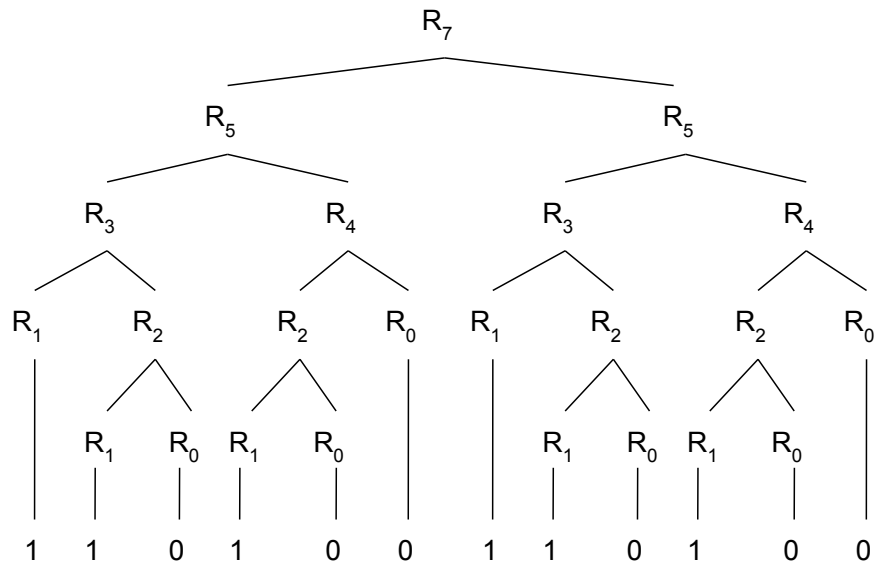


Figure 5.3: Expansion tree of the grammar rule R_7 of Figure 5.2.

Finally, every sth position of B is sampled, for a parameter s. An array $S_n[0, n/s]$ stores a tuple (p, o, rnk) at $S_n[i]$, where the expansion of $C[p]$ contains $B[i \cdot s]$, that is, $p = \max\{j, L(j) \leq i \cdot s\}$, where $L(j) = 1 + \sum_{k=1}^{j-1} \ell(C[k])$; $o = i \cdot s - L(p)$ is the offset within that symbol; and $rnk = \texttt{rank}_0(B, L(p) - 1)$.

Let $S[0] = (0, 0, 0)$. To solve $\texttt{rank}_0(B, i)$, let $S_n[\lfloor i/s \rfloor] = (p, o, rnk)$ and set $l = s \cdot \lfloor i/s \rfloor - o$. Then we move forward from $C[p]$, updating $l = l + \ell(C[p])$, $rnk = rnk + z(C[p])$, and $p = p+1$, as long as $l+\ell(C[p]) \leq i$. When $l \leq i < l+\ell(C[p])$, we have reached the rule $C[p] = X \to YZ$ whose expansion contains $B[i]$. Then, we recursively traverse X as follows. If $l+\ell(Y) > i$, we recursively traverse Y. Otherwise we update $l = l + \ell(Y)$ and $rnk = rnk + z(Y)$, and recursively traverse Z. This is repeated until $l = i$ and we reach a terminal symbol in the grammar. Finally, we return rnk. Obviously, we can also compute $\texttt{rank}_1(B, i) = i - \texttt{rank}_0(B, i)$. Solving $\texttt{access}(B, i)$ is completely equivalent, but instead of returning rnk we return the terminal symbol we reach when $l = i$.

To solve $\texttt{select}_0(B, j)$, we binary search S_n to find $S_n[i] = (p, o, rnk)$ and $S_n[i + 1] = (p', o', rnk')$ such that $rnk < j \leq rnk'$. Then we proceed as for $\texttt{rank}_0$, but iterating as long as $z + z(C[p]) \leq j$, and then traversing by going left (to Y) when $z + z(Y) > j$, and going right (to Z) otherwise. The process for $\texttt{select}_1(B, j)$ is analogous (note X contains $\ell(X) - z(X)$ 1s).

On a balanced grammar, a rule is traversed in $O(\lg n)$ time. The time to iterate over C between samples is $O(s)$. Therefore, the total time for `rsa` is $O(s + \lg n)$ and the total space is $O(r \lg n + (n/s) \lg n) + c \lg(\sigma + r)$ bits. The time is multiplied by δ if we use sampling to avoid storing all the information for all the rules.

The authors name this solution `RPB`, from *RePair* compressed Bitmaps.

5.5.2 Rank, Select, and Access on Repetitive Sequences

The structure `WTRP` [NPV14] (Wavelet Tree with bitmaps compressed with *RePair*) is the only existing solution to support `rsa` on grammar-compressed sequences. The structure is a pointerless wavelet tree (recall Section 4.1.1) where each bitmap B_l is compressed with `RPB` (Section 5.5.1). The rationale is that the repetitiveness of S is reflected in the bitmaps of the `WT`.

However, since the wavelet tree `WT` (Section 2.7.1) construction splits the alphabet at each level, those repetitions are cut into shorter ones at each new level, and become blurred after some depth. Therefore, the bitmaps of the first few `WT` levels are likely to be compressible with *RePair*, while the remaining ones are not. The authors [NPV14] use at each level l the technique to represent B_l that yields the least space, `RPB` (see Section 5.5.1), `RRR`, or `CM` (see Section 2.6). In case of a highly compressible sequence, the space can be drastically reduced, but the search performance degrades by one or more orders of magnitude compared to using `CM` or `RRR`: If all the levels use `RPB`, the `rsa` time becomes $O(\lg \sigma \lg n)$.

On the other hand, as repetitiveness is destroyed at deeper levels, the total space is far from that of a plain *RePair* compression of S. A worst-case analysis, albeit pessimistic, can be made as follows: Each node stores a subsequence of S, whose alphabet is mapped onto a binary one (or of size r in an r-ary wavelet tree). We could then take the same grammar that compresses S for each node, remove all the terminal symbols not represented in that node, and map the others onto $\{0, 1\}$ or $[1, r]$. This is not the best grammar for that node, but it is correct and at most of the same size g of the original one. Therefore, each node can be grammar-compressed to at most $O(g \lg n)$ bits, and summed over all the wavelet tree nodes, this yields $O(g\sigma \lg n)$. Therefore, the size grows at most linearly with σ.

5.6 Repetitive Datasets

In this section we describe several repetitive datasets commonly used in the experiments of this part of the thesis. Table 5.1 shows statistics of interest about them and their compressibility: length (n), alphabet size (σ), zero-order entropy ($\mathcal{H}_0$) (see Section 2.3), bits per symbol (bps) obtained by *RePair* (RP, assuming $(2r+c)\lceil \lg(\sigma+r) \rceil$ bits) (see Section 5.4), bps obtained by p7zip (LZ, `www.7-zip.org`), a Lempel-Ziv compressor (see Section 5.3.1), and r/n is the number of runs of the *Burrows-Wheeler Transform* [BW94] of the sequence divided by its length.

We use various DNA collections from the Repetitive Corpus of *Pizza&Chili*[4]. On one hand, to study precisely the effect of repetitiveness in the performance of our proposals, we generate four synthetic collections of about 100MB: `DNA 1%`, `DNA 0.1%`, `DNA 0.01%`, and `DNA 0.001%`. Each `DNA` p`%` text is generated starting from 1MB of real DNA text, which is copied 100 times, and each copied base is changed to some other value with probability $p/100$. This simulates a genome database with different variability between the genomes. As real genomes, we used collections `para`, `influenza`, and `escherichia`, also obtained from *Pizza&Chili*. From the statistics of Table 5.1, we see that `para` and `influenza` are actually very repetitive, while `escherichia` is not that much. Collection `einstein` corresponds to Wikipedia versions of articles about Albert Einstein in German (also available at *Pizza&Chili*) and is the most repetitive dataset we have. Text `einstein.words` is the same collection but regarded as a sequence of words, instead of characters. Sequence `fiwiki` is a prefix of a Wikipedia repository in Finnish[5] tokenized as a sequence of words instead of characters. Sequence `fiwikitags` corresponds to the XML tags extracted from a prefix from the same Finnish Wikipedia repository. Finally, `indochina` is a subgraph of the Web graph *Indochina2004* available at the WebGraph project[6] containing 2,531,039 nodes and 97,468,933 edges. Each node has an adjacency list of nodes, which is stored as a sequence of integers. Each list is separated from the next with a special separator symbol. Finally, we use several bitmaps (those collections with suffix *.st*) which correspond to the Balanced Parentheses representation (see Section 2.9) of the suffix tree topology (see Chapter 13) of the referred collection.

[4] `http://pizzachili.dcc.uchile.cl/repcorpus`
[5] `http://www.cs.helsinki.fi/group/suds/rlcsa`
[6] `http://law.dsi.unimi.it`

dataset	n	σ	H_0	RP	LZ	r/n
influenza.st	603	2	1.00	0.226	0.379	0.031
escherichia.st	434	2	1.00	0.711	0.595	0.188
para.st	1,692	2	1.00	0.372	0.400	0.0965
einstein.st	367	2	1.00	0.014	0.031	0.001
DNA.1.st	364	2	1.00	0.660	0.607	0.101
DNA.01.st	390	2	1.00	0.200	0.236	0.030
DNA.001.st	394	2	1.00	0.082	0.090	0.016
DNA.0001.st	395	2	1.00	0.046	0.062	0.012
DNA.1	99	5	2.00	0.819	0.172	0.094
DNA.01	99	5	2.00	0.178	0.042	0.016
DNA.001	99	5	2.00	0.075	0.024	0.007
DNA.0001	99	5	2.00	0.063	0.021	0.006
para	429	5	2.12	0.376	0.191	0.036
influenza	154	15	1.97	0.280	0.132	0.019
escherichia	112	15	2.00	1.048	0.524	0.133
fiwikitags	48	24	3.37	0.110	0.219	0.031
einstein	92	117	5.04	0.019	0.009	0.001
software	210	134	4.69	0.139	0.214	0.009
einstein.words	17	8,046	9.92	0.076	0.003	0.001
fiwiki	86	102,423	11.06	0.235	0.034	0.008
indochina	100	2,576,118	15.39	1.906	0.159	0.076

Table 5.1: Statistics of the repetitive datasets. Length n is measured in millions (and rounded).

Chapter 6

Grammar Compressed Sequences

In this chapter we propose two new solutions for `rsa` queries over grammar compressed sequences, and compare them with various alternatives on a number of real-life repetitive sequences. Our contributions are as follows:

1. Our first structure, tailored to sequences over small alphabets, extends and improves the current representation of bitmaps [NPV14]. On a balanced grammar of size g, it obtains $O(\lg n)$ time for all the `rsa` operations with $O(g\sigma \lg n)$ bits of space, using in practice similar space while being much faster than previous work [NPV14]. We dub this solution `GCC` (Grammar Compression with Counters). It can be used, for example, on sequences of XML tags or DNA.

2. Our second structure combines `GCC` with alphabet partitioning (see Section 2.7.4) and is aimed at sequences with larger alphabets. Alphabet partitioning splits the sequence S into subsequences over smaller alphabets. If these alphabets are small enough, we apply `GCC` on them. On the subsequences with larger alphabets, we use representations similar to previous work [NPV14]. The resulting time/space guarantees are as in previous work [NPV14], but the scheme is much faster in practice while using about the same space.

While up to an order of magnitude faster than the alternative grammar-compressed representation, our solutions are still an order of magnitude slower than statistically compressed representations, but they are also an order of magnitude smaller on repetitive sequences.

This chapter is organized as follows: Section 6.1 explains our `rsa` data structures for small alphabets; Section 6.2 presents our solution for `rsa` on large alphabets;

Section 6.3 experimentally evaluates our proposals while Section 6.4 gives conclusions and future research lines.

6.1 Efficient `rsa` for Sequences on Small Alphabets

Our first proposal, dubbed `GCC` *(Grammar Compression with Counters)* is aimed at solving `rsa` queries on grammar-compressed sequences with small alphabets. We first generalize the existing solution for bitmaps (`RPB`, Section 5.5.1), to sequences with $\sigma > 2$. We also introduce several enhancements regarding how we store the additional information to solve `rsa` queries. We also propose two different sampling approaches that yield different space-time tradeoffs, both in theory and in practice.

Let (R, C) be the result of a balanced *RePair* grammar compression of S. We store $S_\ell[X] = \ell(X)$ for each grammar rule $X \in R$. In addition, we store an array of counters $S_a[X]$ for each symbol $a \in \Sigma$: $S_a[X] = \texttt{rank}_a(exp(X), \ell(X))$ is the number of occurrences of a in $exp(X)$.

The input sequence S is also sampled according to the new scenario: each element (p, o, rnk) of $S_n[1, n/s]$ is now replaced by $(p, o, lrnk[1, \sigma])$, where $lrnk[a] = \texttt{rank}_a(S, L(p) - 1)$ for all $a \in \Sigma$, s being the sampling period.

The extra space incurred by σ can be reduced by using the same δ-sampling of `RPB`, which increases the time by a factor δ. In this case we also use the bitmap $B_d[1, r]$ that marks which rules store counters. We further reduce the space by noting that many rules are short, and therefore the values in S_ℓ and S_a are usually small. We represent them using direct access codes (DACs, see Section 2.8), which store variable-length numbers while retaining direct access to them. The o components of S_n are also represented with DACs for the same reason.

On the other hand, the p and $lrnk[1, \sigma]$ values are not small but are increasing. We reduce their space using a two-layer strategy: we sample S_n at regular intervals of length ss. We store $SS_n[j] = S_n[j \cdot ss]$, and then represent the values of $S_n[i] = (p, o, lrnk[1, \sigma])$ in differential form, in array $S'_n[i] = (p', o, lrnk'[1, \sigma])$, where $p' = p - p^*$ and $lrnk'[a] = lrnk[a] - lrnk^*[a]$, with $SS_n[\lfloor i/ss \rfloor] = (p^*, o^*, lrnk^*[1, \sigma])$.

The total space for the p and $lrnk[1, \sigma]$ components is $O(\sigma((n/s)\lg(s \cdot ss) + (n/(s \cdot ss))\lg n))$ bits, whereas the o components use $O((n/s)\lg n)$ bits in the worst case. For example, if we use $ss = \lg n$ and $s = \lg^{O(1)} n$ (a larger value would imply an excessively high query time), the space becomes $O(r\sigma \lg n + (n/s)(\sigma \lg\lg n + \lg n)) + c\lg(\sigma + r)$ bits.

A further improvement is aimed to reduce the space on extremely repetitive sequences. In this scenario, many elements of S_n may contain the same values: if a rule covers a wide range of S, we store the same S_n values for many samples of S. Thus, we sample the vector C instead of sampling the whole sequence S. Instead of $(p, o, lrnk[1, \sigma])$ we store a tuple $(i, lrnk[1, \sigma])$, where i is the position where the sampled cell of C starts in S, and $lrnk$ is computed up to $i - 1$. On the other

hand, the two-layer scheme cannot be applied, because now the samples may cover arbitrarily long ranges of S.

The total space with this sampling then becomes $O(r\sigma \lg n + \sigma(c/s)\lg n) + c\lg(\sigma + r) = O((r+c)\sigma \lg n)$ bits. This removes any linear dependency on n from the space formula. The size of the *RePair* grammar is $g = O(r+c)$, thus the space can be written as $O(g\sigma \lg n)$ bits.

The rsa algorithms stay practically the same as for RPB; now we use the symbol counter of a for $\texttt{rank}_a$ and $\texttt{select}_a$. The resulting data structure solves rsa in time $O(s + \lg n)$. In case C is sampled instead of S, there is an additional $O(\lg c)$ time to binary search for the right sample. This is still within $O(s + \lg n)$. If we choose $s = O(\lg n)$, then the time is $O(\lg n)$. The space is still $O(g\sigma \lg n)$ if we sample C.

When σ is small and the sequence is repetitive, this data structure is very space- and time-efficient. It outperforms WTRP [NPV14] (Section 5.5.2) in time: WTRP takes $O(\lg \sigma \lg n)$ time and our GCC uses $O(\lg n)$. In terms of space, both use $O(g\sigma \lg n)$ bits and perform similarly in practice. In the next section we develop a variant for large alphabets that, although cannot guarantee any better than $O(g\sigma \lg n)$ bits, uses much less in practice.

6.2 Efficient rsa for Sequences on Large Alphabets

Our main idea for large alphabets is to use wavelet trees/matrices or alphabet partitioning as a mechanism to cut Σ into smaller alphabets, which can then be handled with GCC. This is in the same line of WTRP, but we find better solutions to avoid the degradation of repetitiveness due to the partitioning.

The most immediate approach is to generalize WTRP to use a Multi-ary wavelet tree MWT , since now we can use GCC on small alphabets $[1, r]$ to represent the sequences S_v stored at the internal nodes of the MWT. Compared to a binary wavelet tree WT, a MWT takes more advantage of repetitiveness before splitting the alphabet, and reduces the time complexity from $O(\lg \sigma \lg n)$ to $O(\log_r \sigma \lg n)$. The worst-case space is still $O(g\sigma \lg n)$ bits. The use of a WM requires only $\lg_r \sigma$ grammars, one per level, but still the guarantee on their total size is the same.

A less obvious way to use GCC is to combine it with Alphabet-Partitioning AP (Section 2.7.4). Note that the string K is a projection of S, and therefore it retains all its repetitiveness. Further, it contains a small alphabet, of size $\lg \sigma$, and therefore we can use GCC on it. The resulting representation takes at most $O(g \lg \sigma \lg n)$ bits.

The other important sequences are the S_j, which have alphabets of size 2^{j-1}. For the smallest j, this is small enough to use GCC as well. For larger j, however, we must resort to other representations, like WTRP, GMR, or WT/WM, depending on how compressible they are.

An interesting fact of AP is that it groups symbols of approximately the same frequency. The symbols participating in the most repetitive parts of S have a good chance of having similar frequencies and thus of belonging to the same subalphabet

S_j, where their repetitiveness will be preserved. On the other hand, the larger alphabets, where GCC cannot be applied, are likely to contain less frequent symbols, whose representation using faster structures like GMR or WT/WM do not miss very important opportunities to exploit repetitiveness.

Note that, if we do not use WTRP for the larger subalphabets, then the time performance for rsa queries stays within $O(\lg n)$, independently of the alphabet size. In exchange, we cannot bound the size of the representation in terms of the size of the grammar that represents S. Instead, if we use WTRP, our worst-case guarantees are the same as for WTRP itself, but in practice our structure will prove to be much better, especially in time.

6.2.1 AP with GCC in Practice

We introduce two new parameters for the combination of AP and GCC. The first parameter, cut, tells that the $2^{\texttt{cut}}$ most frequent symbols will be directly represented in K. This parameter must be set carefully to avoid increasing too much the alphabet of K, since K is represented with GCC.

Our second parameter is $\texttt{cut}_\texttt{o}$, which tells how many of the first S_j classes are to be represented with GCC. For the remaining sequences S_j we consider two options: (a) if S_j is not grammar-compressible, we use GMR [GMR06] (Section 2.7.3), which does not compress but is very fast, or (b) if S_j is still grammar-compressible, we use WTRP, which is the grammar-based variant that performed best.

6.3 Experimental Results

6.3.1 Setup and Datasets

We used an Intel(R) Xeon(R) E5620 at 2.40GHz with 96GB of RAM memory, running GNU/Linux, Ubuntu 10.04, with kernel 2.6.32-33-server.x86_64. All our implementations use a single thread and are coded in C++. The compiler is g++ version 4.7, with -O9 optimization. We implemented our solutions inside Libcds [Cla] and use Navarro's implementation of *RePair* (www.dcc.uchile.cl/gnavarro/software/repair.tgz).

To experimentally evaluate our proposals, we use all synthetic and real datasets described in Section 5.6.

6.3.2 Parameterizing the Data Structures

We compare our data structures with several others. The list of structures compared, along with the parameters used, is listed next. These parameter ranges are chosen because they have been proved adequate in previous work, or because we have obtained the best space/time tradeoffs with them.

- `GCC.N` is our structure for small alphabets where we sample S at regular intervals. We set the sampling rate to $s = \{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}\}$, the rule sampling to $\delta = \{0, 1, 2, 4\}$, and the superblock sampling to $ss = \{5, 8\}$.

- `GCC.C` is our structure for small alphabets where we sample C at regular intervals. We set the sampling rate to $s = \{2^6, 2^7, 2^8, 2^9, 2^{10}\}$ and the rule sampling to $\delta = \{0, 1, 2, 4\}$.

- {`WT`|`WM`|`WTH`|`WMH`}.{`CM`|`RRR`} is a Wavelet Tree (`WT`) (see Section 2.7.1), a Wavelet Matrix (`WM`) (see Section 4.1.2), a Huffman-shaped wavelet tree (`WTH`) (see Section 2.7.2), or a Huffman-shaped or Compressed Wavelet Matrix (`WMH`) (see Section 4.2) with bitmaps represented either with `CM` or `RRR` (see Section 2.6). For `CM` we use the implementation [GGMN05] with one level of counters over the plain bitmap, while `RRR` corresponds to the implementation [CN08] of `RRR`. In both cases, the sampling rate for the counters was set to $\{32, 64, 128\}$.

- {`WT`|`WM`|`WTH`|`WMH`}.`RP` are the `WT`, `WM`, `WTH` or `WMH`, with the bitmaps compressed with *RePair*. Therefore, `WM.RP` is equivalent to `WTRP` (see Section 5.5.2), but with our improved implementation using a wavelet matrix and `GCC` for the bitmaps. As in `WTRP`, we use several bitmap representations depending on the compressibility of the bitmap: `GCC` varying the parameters as described above, `RRR` or `CM` with sampling set to 32. We choose the one using the least space among these.

- `AP` is a plain alphabet partitioning technique described in Section 2.7.4. We used parameter values $\texttt{cut} = \{2^3, 2^4, 2^5, 2^6\}$ and $\texttt{cut}_\texttt{o} = \{1, 3, 5\}$. The sequence K is represented with `WT.RRR` with sampling set to 32. The sequences S_j are represented with `GMR` using the default configuration provided in the libcds tutorial[1].

- `APRep`.{`WMRP`|`GMR`} is our `AP`-based variant for large alphabets. We use the same values `cut` and $\texttt{cut}_\texttt{o}$ as for `AP`. The sequence K and the first $\texttt{cut}_\texttt{o}$ sequences S_j are represented with `GCC`. The remaining sequences S_j are represented either with `WM.RP` or with `GMR` (see Section2.7.3), using their already described configurations.

- `MWTH.RP` is a Multi-ary Wavelet tree (`MWTH`) (see Section 2.7.1) using *RePair*-compressed sequences in the nodes. As for `APRep`, we use two different representations for the node sequences. The first $\texttt{cut} = \{2, 3, 4\}$ levels are represented with `GCC`, and the rest with a `WT.RRR` with fixed sampling 32. We tested arities in $\{4, 8, 16\}$. We did not try combining with the `WM` because it is slower (requires more operations) and the overhead of σ/r nodes is not as large as for σ nodes of the binary case. Also, the Huffman-shaped variants are shown to be always superior.

[1] `https://github.com/fclaude/libcds/blob/master/tutorial/tutorial.pdf`

Among all the data points resulting from the combination of all the parameters, in the experiments we only show those points which are space/time dominant.

Regarding queries, those for `access` are positions at random in $S[1,n]$. For `rank`, we used a random position p in $S[1,n]$ and the symbol is $S[p]$. Finally, for `select`, we took a random position p in $S[1,n]$, using $S[p]$ and a random rank in $[1, \texttt{rank}_{S[p]}(S,n)]$. We generated $10,000$ queries of each type, reporting the average time for each operation.

In Section 6.1 we proposed two sampling approaches for `GCC`: `GCC.N` is regular in S and `GCC.C` is regular in C. We anticipated that `GCC.C` should use less space on more repetitive sequences, but it could be slower. Now we compare both sampling methods on the repetitive sequences with smaller alphabets described in Section 5.6. Figure 6.1 shows the results for `rank` and `select` (`access` is equivalent to `rank` in our algorithms).

While, as said, `GCC.C` might use less space than `GCC.N` when the sequence is more repetitive, this occurs in practice only slightly on `DNA0001`, and spaces become closer as repetitiveness decreases on synthetic datasets (`DNA001` to `DNA1`). Still, the differences are very slight, and instead `GCC.N` is much faster than `GCC.C` for the same space usage. The same occurs in the real sequences. For the remaining experiments, we will use only `GCC.N`.

6.3.3 Performance on Small Alphabets

We compare our `GCC.N` with `WT.RP`, `WTH.RP`, and `WM.RP`. We also include in the comparison two statistically compressed representations that are the best for small and moderate alphabets: `WTH.CM` and `WTH.RRR`.

Figure 6.2 shows the results for `rank` and `select` on the real collections that have small and moderate alphabets (again, the results for `access` are very similar to those for `rank`). It can be seen that `WTH.RP` generally performs better than `WT.RP` in space and time, as expected. The variant `WM.RP` performs slightly better than `WT.RP` in space, as it represents only one grammar per level and not per node (the difference would be higher on larger alphabets). In exchange, `WM.RP` is slightly slower than `WT.RP` because it performs more `rank`/`select` operations on the bitmaps represented with `GCC`. Finally, `WMH.RP` uses less space than `WM.RP` only in some cases, but it generally outperforms it for the same space. It performs particularly well on `escherichia`, the least repetitive of the datasets.

Recall that `WM.RP` is our improved version of previous work, `WTRP` [NPV14], and it is now superseded by `GCC.N`. The space of `WM.RP` is in most cases similar to that of `GCC.N`, which means that `WM.RP` is actually close to the worst-case space estimation, $O(g\sigma \lg n)$. In some cases, `GCC` is significantly smaller. More importantly, `GCC.N` is 2–15 times faster than `WM.RP`, and also 2–7 times faster than `WTH.RP`, the faster of the competitors in this family, which also uses more space than `GCC.N`. `GCC.N` solves queries in a few microseconds.

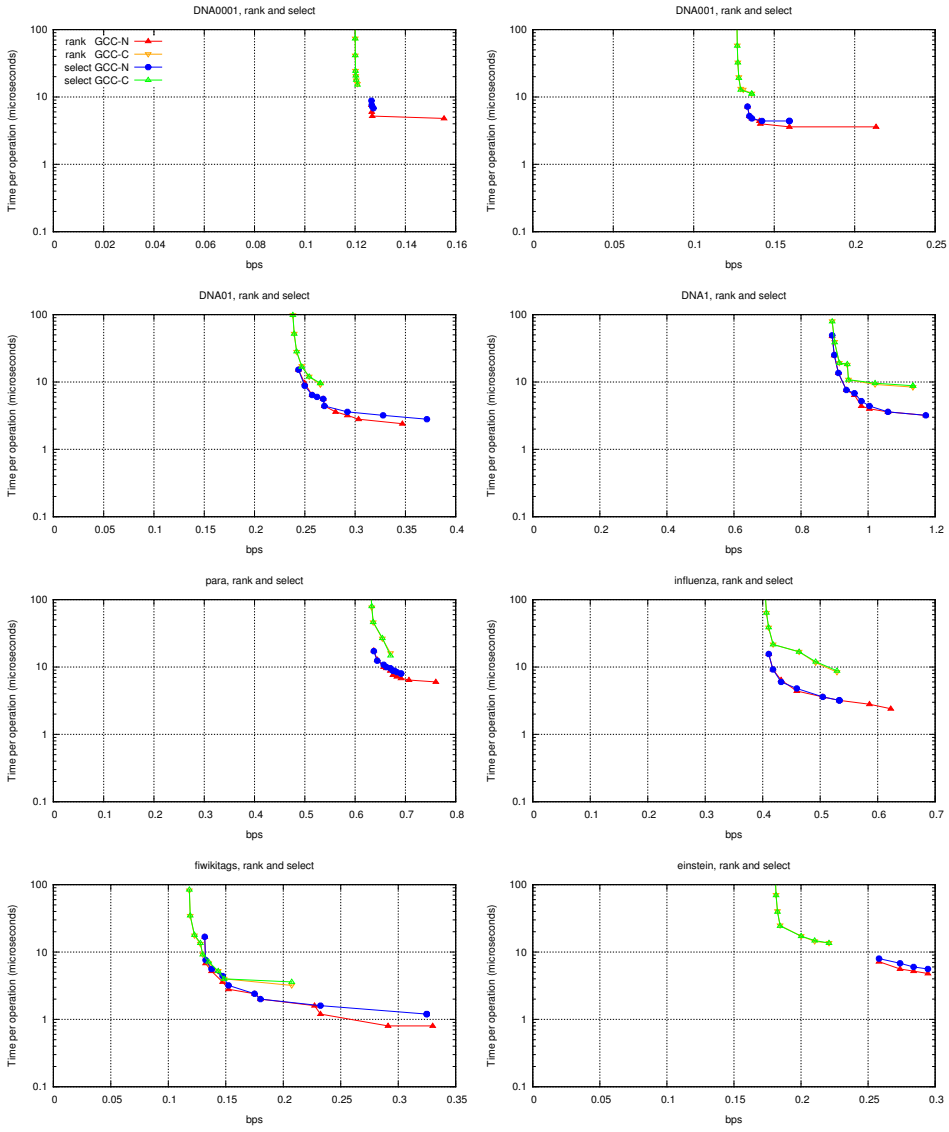


Figure 6.1: Comparison of `rank` and `select` performance of `GCC.N` and `GCC.C`.

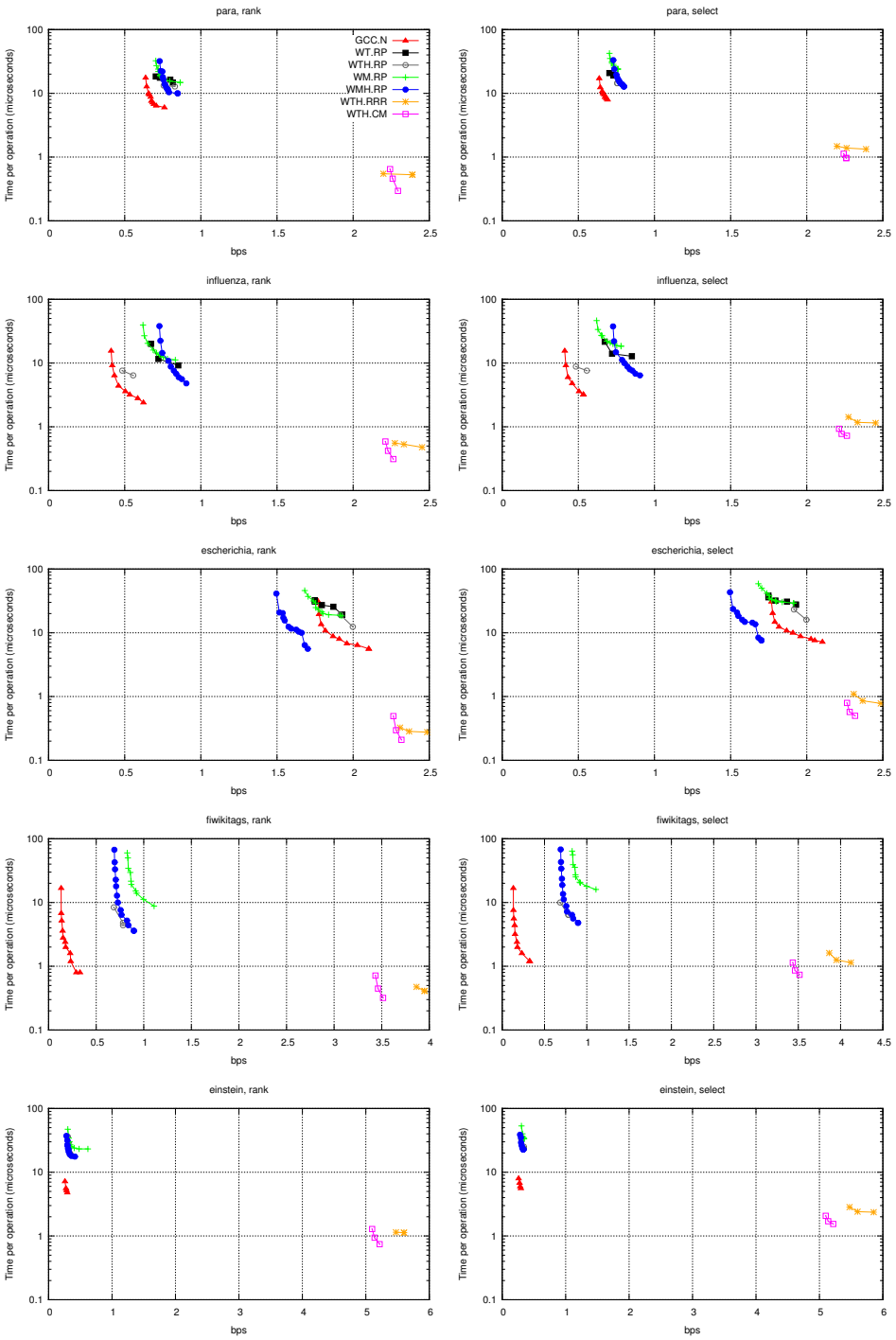


Figure 6.2: Space-time tradeoffs for `rank` and `select` queries over small alphabets (time in logscale).

On the other hand, the representations that compress statistically, WTH.CM and WTH.RRR, are about an order of magnitude faster than GCC.N, but also take 5–15 times more space (except on escherichia, which is not repetitive).

6.3.4 Performance on Large Alphabets

Now we use the collections einstein (again), software, einstein.words, fiwiki, and indochina from Section 5.6, to compare the performance on moderate and large alphabets. We compare the two versions of our APRep, our MWTH.RP, and all the statistically compressed or compact schemes for large alphabets: WM/WMH with CM/RRR and AP (we only exclude WM.CM, which always loses to others). In the first two collections, whose alphabet size is moderate, we also include GCC.N, to allow comparing its performance with our variants for large alphabets in these intermediate cases.

Figure 6.3 shows the results for rank and select queries (once again, access is omitted for being very similar to the results of rank).

Recall that WM.RP is our improvement over the previous work, WTRP [NPV14]. The Huffman-shaped variant, WMH.RP, outperforms it only slightly in time. Our multi-ary version, MWTH.RP, is clearly faster, but not smaller as one could expect. Indeed, it is larger when σ grows, probably due to the use of pointers. What is most interesting, however, is that all those variants are clearly superseded by our AP.RP.WMRP, which dominates them all in time (only reached by MWTH.RP while using much more space) and in space (only reached by WM.RP while using much more time). Compared with previous work [NPV14], AP.RP.WMRP is then 2–4 times faster than WTRP, while using the same space or less. AP.RP.WMRP solves queries in a few tens of microseconds.

Note the particularly bad performance of the Huffman-based versions on indochina. This is because this collection contains inverted lists, which form long increasing sequences that become runs in the wavelet tree; the Huffman rearrangement breaks those runs.

Our second variant, AP.RP.GMR, is not so interesting for repetitive collections. Although it is 2–5 times faster than AP.RP.WMRP, it uses much more space, not so far from that used by statistical representations. Those are, as before, about an order of magnitude faster than AP.RP.WMRP, but also use 3–5 times more space. Also, we can see that GCC.N is competitive on einstein, which is very repetitive, but not so much on software. Both of our new AP.RP versions designed for large alphabets outperform it in space, while they are not slower in time (in some cases they are even faster).

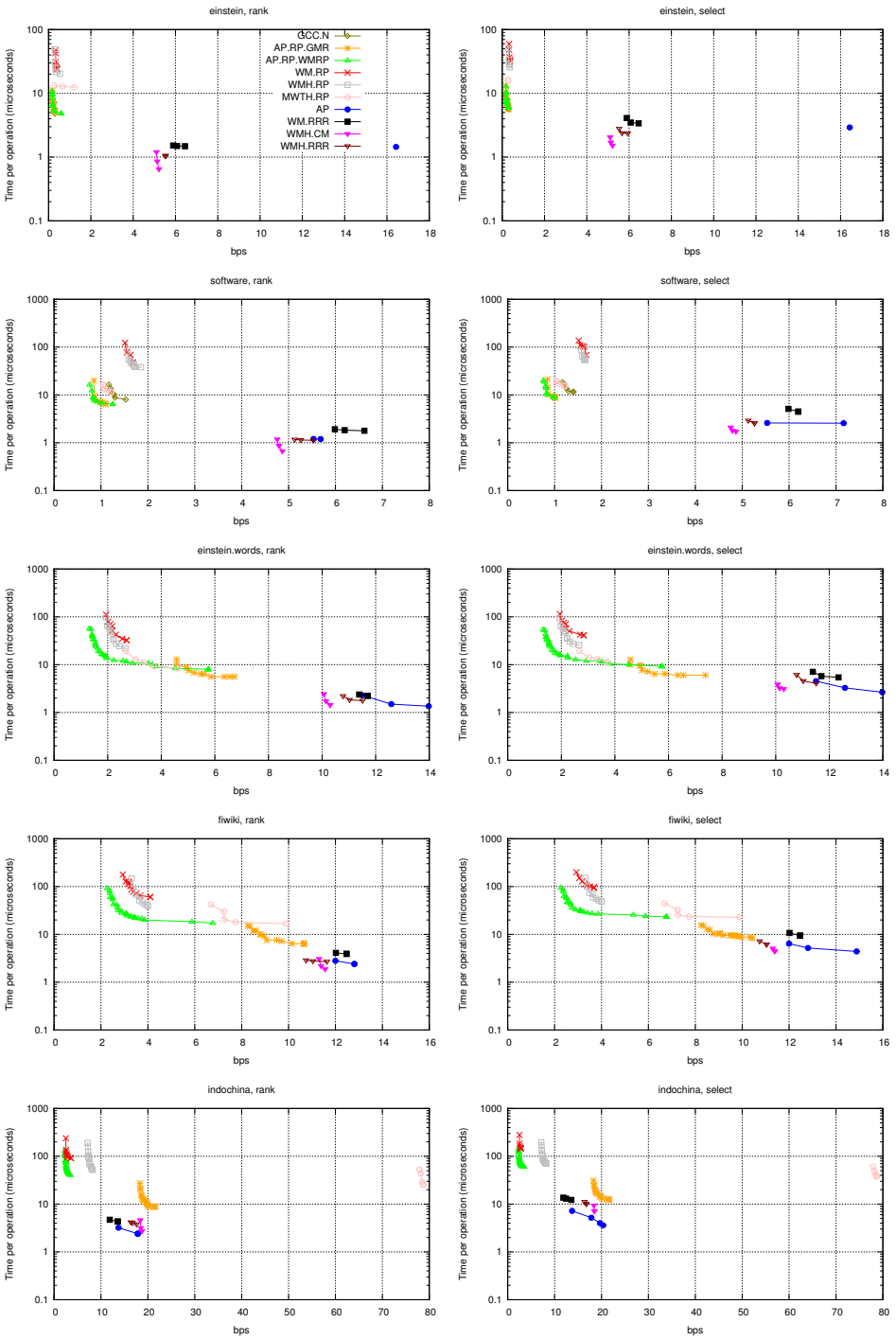


Figure 6.3: Space-time tradeoffs for `rank` and `select` queries over moderate and large alphabets (time in logscale).

6.4 Discussion

We have introduced new sequence representations that take advantage of the repetitiveness of the sequence, by enhancing the output of a grammar compressor with extra information to support efficient direct access, as well as `rank` and `select` operation on the sequence. The only previous grammar-compressed representation [NPV14] is 2–15 times slower and uses the same or more space than our new representations. Our structures answer queries in a few tens of microseconds, which is about an order of magnitude slower than the times of statistically compressed representations. However, on repetitive collections, our structures use 2–15 times less space.

After the publication [NO14b] of the results described in this chapter, Belazzougui et al. [BPT15] gave more theoretical support to our results. They obtained our same $O(\lg n)$ time for `rsa` operations with $O(g\sigma \lg n)$ bits on arbitrary grammars of size g (not only balanced ones). They also show how to obtain $O(\lg n / \lg\lg n)$ time using $O(g\sigma \lg(n/g) \lg^{1+\epsilon} n)$ bits, for any constant $\epsilon > 0$. Most importantly, they prove that it is unlikely that these times for `rank` and `select` can be significantly improved, since long-standing reachability problems on graphs would then be improved as well. This shows that the time complexity of their (and our) solutions are essentially the best one can expect.

A practical aspect where our structures could possibly be improved is in the clustering of the alphabet symbols used when partitioning the alphabet, both in the simple case of alphabet partitioning and in the hierarchical case of wavelet trees and matrices. In the first case, we obtained a significant space improvement by sorting the symbols by frequency, whereas in the second case none of our attempts performed noticeably better than the original alphabet ordering. While unsuccessful for now, we believe that some clever clustering scheme that avoids separating symbols that appear together in repetitive parts of the sequence could considerably improve the space on large alphabets.

Another future goal is to find ways to improve the time of these grammar compressed representations. We believe this is possible, even if known lower bounds suggest that there must be a price of at least an order of magnitude compared with statistically compressed representations.

Finally, it would be interesting to build `rsa` data structures on Lempel-Ziv compressed representations, which is more powerful than grammar compression. This idea is explored in Chapter 7, where we present the first Lempel-Ziv-based `rsa` data structure.

Chapter 7

Block Trees for Sequences

The motivation to obtain an *LZ77*-bounded `rsa` data structure is sharp: Firstly, there is a plenty of applications in highly repetitive scenarios that demand `rsa`-capable solutions (recall Chapter 6). Secondly, being $S[1,n]$ a string and $G^*(S)$ the smallest context-free grammar for S, we know that $|LZ77(S)| \leq |G^*(S)|$ [Ryt03]. That is, the smallest grammar is never smaller than an *LZ77* factorization of the same string, which means that obtaining an *LZ77*-space-bounded `rsa` data structure is potentially better than grammar-based ones in terms of space.

However, and as explained in Section 5.3, just supporting `access` queries on *LZ77*-bounded representations is difficult, and not many solutions are currently known. *LZ-End*-based data structures (see Section 5.3.3) are able to obtain optimal `access` time if queries hit a phrase boundary. If not, query performance becomes dependent on the transitive depth of the parsing, which is not easily bounded (see Section 5.3). An alternative to *LZ-End* are Block Graphs [GGP11, GHP14]. A Block Graph is an *LZ77*-bounded data structure that carries out `access` operations in $O(\lg n)$ time, but uses a factor of $O(\lg n)$ space more than *LZ77*. In any case, none of these solutions are able to support `rank` and `select` queries.

In this chapter we show how to modify a Block Graph [GGP11, GHP14], which is a DAG (direct acyclic graph), to convert it into a tree data structure that supports `rsa` queries efficiently. We dubbed our data structure *Block Tree* (`BT`) and it can store a string $S[1..n]$ over an alphabet of size σ in $O\left(zr \lg n \log_r \frac{n \lg \sigma}{z \lg n}\right)$ bits, where z is the number of phrases in the *LZ77* parse of S and $r \leq n$ is a parameter, such that we can support extraction of a substring of length m in $O\left(\log_r \left(\frac{n \lg \sigma}{z \lg n}\right) \cdot \left(\frac{m \lg \sigma}{\lg n} + 1\right)\right)$ time. Using a σ factor more space, we can support `rank` in $O\left(\log_r \frac{n \lg \sigma}{z \lg n}\right)$ time and `select` in $O\left(\log_r \left(\frac{n \lg \sigma}{z \lg n}\right) \lg \lg n\right)$ time. The resulting data structure is of practical nature, so we demonstrate its efficiency with an experimental evaluation carried out over bitmaps and sequences with small alphabets.

This chapter is organized as follows: Section 7.1 explains what a block graph is; Section 7.2 details our proposal, which we dubbed *Block Tree*, describing how it is construct, how queries are carried out, and providing a space-time theoretical performance analysis; Section 7.4 focuses on the implementation details, which slightly differ from the theoretical conception of the structure; Section 7.5 presents an experimental evaluation in which we compare our proposal with several state of the art references, including our new developments of Chapter 6; and finally Section 7.6 gives our conclusions.

7.1 Block Graphs

A block graph [GGP11, GHP14] for a string $S[1, n]$ is a directed acyclic graph (DAG) in which each node has a fan-in and -out of at most 2 and 3 respectively. Each node of the graph covers a substring of S, which is known as the node's block. The root of the graph covers the whole S and is at depth $d = 0$. Suppose $n = 2^c, c \in \mathbb{N}^+$ (otherwise we pad S until n becomes a power of 2) and that we are at depth d with $0 \leq d < h$. If node v covers the substring $S[i, i + b - 1]$, with $b = 2^{h-d}$, then v's first child will cover $S[i, i + b/2 - 1]$, the second $S[i + b/4, i + 3b/4 - 1]$, and finally the third $S[i + b/2, i + b - 1]$. Besides, if v_{ls} and v_{rs} are, respectively, the immediate left and right siblings of v, then ${v_{ls}}'s$ rightmost child will also cover segment $S[i, i + b/2 - 1]$, while ${v_{rs}}'s$ leftmost child will cover $S[i + b/2, i + b - 1]$. This is why a node may have a fan-in of 2.

We can reduce the size of a block graph by truncating its depth. The sweet point to do so is reached when storing the graph pointers becomes more expansive than just storing the bare blocks. We distinguish two sort of nodes in a block graph: internal and leaves. A node is said to be internal if the string it covers is the first occurrence of that string in S; otherwise it is a leaf. After this, we should remove leaves' children and their descendants and replace them by pointers as follows. Suppose we are at node v which is at depth d. Suppose also v is a leaf, and that one of its children is covering substring $S[i, j]$. Then, we know $S[i, j]$ is not the first occurrence of that substring in S, otherwise v would not be a leaf. Suppose $S[i', j']$ is the first occurrence of $S[i, j]$ at depth d. Note also that $S[i', j']$ is completely contained by an internal node u at the same depth d (to ensure this is why block graphs permit blocks to overlap). Then, we replace v's children pointer to $S[i, j]$ by a backpointer to node u, storing the offset within u where $S[i', j']$ starts. Figure 7.1 shows an example of a block graph for a Fibonacci word truncated at depth 3. Leaves are depicted as rectangular nodes and internal nodes are ovals.

Recalling that z is the number phrases of an *LZ77*-parsing [ZL77] (Section 5.3.1), a block graph takes $O\left(z \lg^2 n\right)$ bits and extracts a pattern $p[1, m]$ in $O\left(\lg n + m\right)$ time. If we truncate the block graph at depth d, the space becomes $O\left(z(\lg n - d) + 2^d)\right)$ while extraction time decreases to $O\left(\lg n - d + m\right)$.

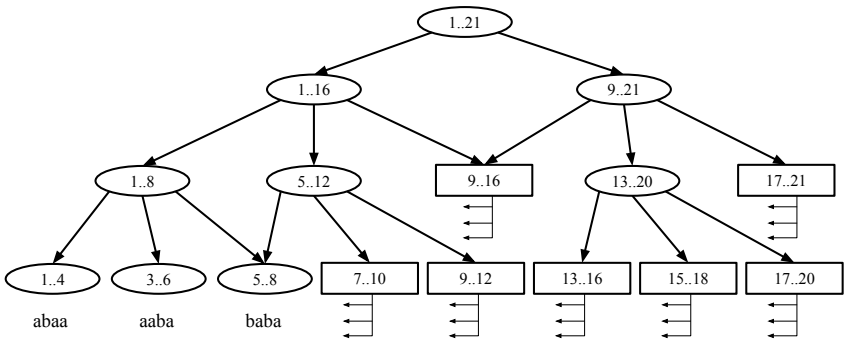


Figure 7.1: A block graph truncated at depth 3 for the first 21 elements of the Fibonacci word 'abaababaabaababaababa'.

7.2 Block Trees

In this section we describe our proposal, which we dubbed Block Tree (`BT`). We explain how we build it by providing a construction algorithm, how queries are carried out, as well as providing a space-time analysis of the data structure and operations.

7.2.1 Block Trees Structure

Suppose we are given a string $S[1, n]$ drawn over an alphabet of size σ. Let $1 < r \leq n$ be the arity of the the block tree (`BT`). If $n < r$ then our `BT` is a single node covering the whole S. Otherwise we start at depth $d = 0$ by dividing S into r blocks $S^1, S^2, \ldots S^r$ such that $|S^1| = \cdots = |S^{n \bmod r}| = \lceil n/r \rceil$ and $|S^{(n \bmod r)+1}| = \cdots = |S^r| = \lfloor n/r \rfloor$. That is, each node v_i covers a substring S^i of length at most $\lceil n/r \rceil$ without permitting overlapping between nodes. As in the case of block graphs, the string covered by a node v is called its block. We also distinguish two kind of nodes: internal and leaves. A node v_i at depth d is marked as internal if there is no node v_j or pair of nodes $v_{j-1}v_j$ with $1 < j < i$ that contains the substring covered by v_i. Otherwise v_i is not marked and hence, it is a leaf.

If v_i is a leaf, then we need to store the following information:

- a pointer to its left siblings (which must be marked as internal) whose corresponding blocks contain the leftmost occurrence of S^i.
- the offset of that occurrence within those blocks.

Then, at level $d + 1$ we concatenate all segments S^i of level d that have been marked as internal, dividing again each block S^i into r sub-blocks as evenly as

possible such that larger sub-blocks precede smaller ones. After that, this process of finding the first occurrence of each substring is repeated until we obtain a structure with $\log_r n$ levels.

Note each level of the block tree contains blocks of size n/r^{d+1}, being the first level of our block tree at $d = 0$. We can recursively divide a segment until (a) it cannot be divided any more, or (b) the cost of storing a block becomes less than that of storing a pointer, which happens when the blocks have size $O(\lg n / \lg \sigma)$. Note that option (b) shrinks the number of levels of our block tree to $\log_r \frac{n \lg \sigma}{\lg n}$.

If we know z (the number of *LZ77* phrases of S), then we can further reduce the height to $\log_r \frac{n \lg \sigma}{z \lg n}$ by dividing S into rz blocks and then recursing as before; this skips the first $\log_r z$ rounds of the recursion and levels in the tree, at the cost of increasing the size by an $O(zr)$ term (which will not change our asymptotic analysis).

Figure 7.2 shows an example of a block tree with 4 levels for a Fibonacci word. Note the original string has been split into blocks of length 4 at depth $d = 0$ and the arity was set to $r = 2$. We shadowed those blocks that have been marked as internal. For the rest we store backpointers (black arrows) telling the position where a copy of its block previously appeared. Dotted arrows show how we map blocks from a level to the next.

7.2.1.1 Analysis

As said, the first level of the block tree contains blocks of size n/r, the second level blocks of size n/r^2, and so on, until the last level, which has blocks of size $\lg n / \lg \sigma$. Those lowest blocks are stored in plain form[1], as their original substring. By construction, the total number of blocks at any level never exceeds zr, where z is the number of phrases in the *LZ77*-parsing of the string.

At any level i but the last, the t_i blocks are encoded using $O(t_i \lg n)$ bits of space, for storing pointers of $\lg n$ bits each into level i data. At the last level, each block is simply encoded as plain text, i.e., $\lg n / \lg \sigma$ symbols of size $\lg \sigma$ bits each, which is $\lg n$ bits per block.

Since the upper $\log_r z$ levels contain a geometrically increasing number of blocks upper bounded by z, their total encoding size is $O(z \lg n)$ bits. Then, each of the remaining $\log_r \frac{n \lg \sigma}{z \lg n}$ levels will be encoded using $O(zr \lg n)$ bits each, for a total of $O\left(zr \lg n \log_r \frac{n \lg \sigma}{z \lg n}\right)$ bits.

The query time of the block tree will be upper bounded by the number of levels. As mentioned in Section 7.2.1, in order to improve the time without sacrificing our asymptotic space bound, we will start the construction of the block tree from level $\log_r z$. Then, the number of levels is reduced to $\log_r \frac{n \lg \sigma}{z \lg n}$ and the bound $O(zr)$

[1] If `rank` and `select` are pursued functionalities, we should prepare this bare string to support these kind of queries efficiently.

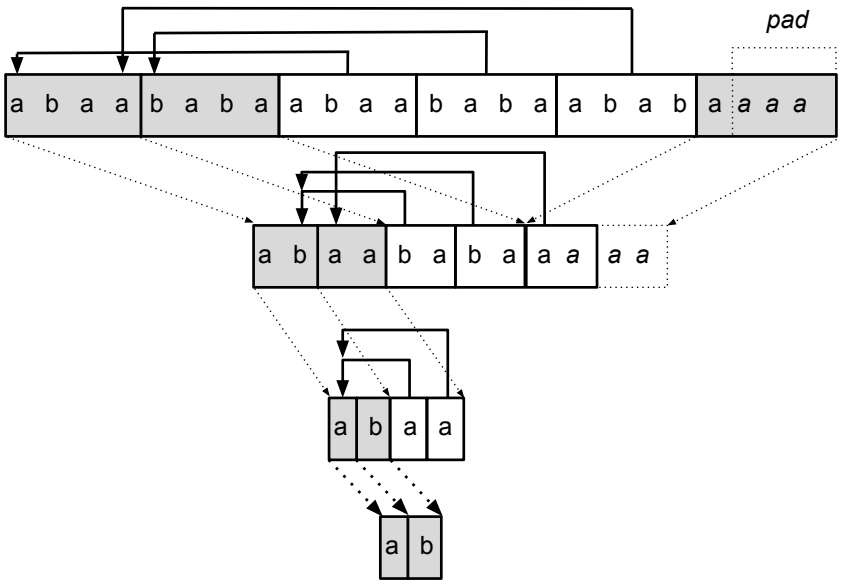


Figure 7.2: An example of our block tree for the first 21 elements of the Fibonacci word 'abaababaabaababaababa' padded with 3 symbols.

on the number of blocks per level still applies. This analysis is summarized in Theorem 4.

Theorem 4. *Given a string S of length n over an alphabet of size σ and a parameter r, we can build a block tree with $\log_r \frac{n \lg \sigma}{z \lg n}$ levels, where z is the number of phrases in the LZ77 parsing of S. The block tree occupies a total of $O\left(zr \lg n \log_r \frac{n \lg \sigma}{z \lg n}\right)$ bits of space.*

We note that $\frac{n \lg \sigma}{z \lg n}$ is actually the compression ratio. An interesting setting for the arity is $r = \left(\frac{n \lg \sigma}{z \lg n}\right)^{\epsilon}$ for some constant $\epsilon < 1$. This makes the space usage $O\left(\frac{(z \lg n)^{1-\epsilon}(n \lg \sigma)^{\epsilon}}{\epsilon}\right)$ and the number of levels $O(1/\epsilon)$. This space usage is a weighted geometric average between $z \lg n$ (the space usage achievable by the *LZ77* parsing) and the original space $n \lg \sigma$. The parameter ϵ allows us to give an arbitrarily large weight to the space usage at the cost of increasing the number of levels (and thus query time).

7.2.2 Construction

The key point of a block tree construction is to identify the first occurrence of each block in S. Being b the block size, finding first occurrences of blocks in $O\left(bn^2\right)$ time is straightforward. However we seek for linear time solutions in order to be practical in any scenario. To do so, we propose a new algorithm that relies in a technique previously proposed by Karp and Rabin [KR87] for string pattern matching.

Given a substring $S[i, j]$, we recursively define its Rabin-Karp *fingerprint* or simply its RK as:

$$RK(S[i,j],k) = \begin{cases} S[k] \ mod \ q & \text{if } k = i \\ (RK(S[i,j],k-1) \cdot \sigma + S[k]) \ mod \ q & \text{otherwise} \end{cases} \tag{7.1}$$

being q the largest prime smaller than $\lg(n/\sigma)$. We also define the Rabin-Karp *shift* or simply the *RK-s* of $S[i, j]$ as

$$RK\text{-}s(S[i,j]) = ((RK(S[i,j]) - S[i] \cdot h) \cdot \sigma + S[j+1]) \ mod \ q$$

where $h = \sigma^{j-i} \ mod \ q$ and $j < n$. Computing $RK(S[i, j])$ takes $O(j - i + 1)$ time while an *RK-s* can be done in $O(1)$.

Our algorithm to build a block tree in RAM memory consist of two phases. In a first stage, we divide the sequence $S[1, n]$ into blocks $S^1, \ldots, S^r$ computing the RK *fingerprint* for each S^i. Then, we store each of them in a hash table along with the starting position of each block using its RK *fingerprint* as key. This process is carried out in linear time and space.

In a second phase, we initially mark the first block S^1 as internal. Then we compute the Rabin-Karp *fingerprint* of each substring $S[i, i + b]$ in $O(1)$ time per substring by carrying out RK shifts starting at S^1. For each substring we use its RK *fingerprint* to access the hash table an check if there is any other block with the same key. If so, it means we potentially have two blocks that cover the same string. We say potentially because two substrings may have the same RK *fingerprint* even though they are different (which happens with very low probability though). If there is no match, then we do nothing. If however, there exists another block $S[j, j + b]$ with the same fingerprint, we have to deal with several options:

- If the block $S[j, j + b]$, which is in the hash table, is already marked as internal, then it means we had found its first occurrence previously and we do nothing.
- If $i > j$, we do nothing because the current block $S[i, i + b]$ is to the right of $S[j, j + b]$, which means $S[i, i + b]$ cannot be the first occurrence of $S[j, j + b]$.
- If $i < j$ and $S[j, j + b]$ is not marked as internal, we need to compare both blocks $S[i, i + b]$ and $S[j, j + b]$ to check if they are actually the same. If so, it means the first occurrence of $S[i, i + b]$ in S is $S[j, j + b]$. We mark those

blocks covered by $S[j, j+b]$ and we set a pointer that tells the first occurrence of $S[i, i+b]$ is at position i. If $S[i, i+b] \neq S[j, j+b]$ we do nothing.

After this, we carry out an RK shift and we continue with the next substring. At the end of the process, we have marked as internal all those blocks that contain first occurrences of the substring they cover, and for those not marked (leaves), we have set up backpointers to those marked blocks that contain their first occurrences. This second stage of our algorithm takes linear space and worst-case $O(bn)$ time, on average it does in $O(n)$ time. Therefore, we can build a `BT` in pseudolinear time and linear space.

Note also we can run a Monte-Carlo approach of this algorithm in $O(n)$ worst-case time by just considering that if the two RK *fingerprints* of two blocks match, then both blocks cover the same string. This avoids comparing blocks directly, reducing the construction time by a factor of b although queries may return the wrong answer with very low probability. This is very interesting, for instance, if we want to build a block tree in the external memory model, since we avoid random scans of the input sequence.

7.2.3 Queries on a Block Tree

The simplest query to answer with a block tree is to return a symbol $S[i]$ given i. To do this, we start at the root and descend to the child whose corresponding block contains $S[i]$, then to the grandchild whose block contains $S[i]$, etc. If we reach a leaf v, then either v stores its block explicitly, and so we can return $S[i]$ immediately, or v stores pointers to its left siblings whose blocks contain the leftmost occurrence in S of v's block, and the offset of that occurrence in those blocks. In the latter case, in $O(1)$ time we can identify a symbol $S[i']$ in one of those left siblings' blocks such that $S[i'] = S[i]$, then start descending from that left sibling to find $S[i']$. Returning $S[i]$ takes a total of $O\left(\log_r \frac{n \lg \sigma}{z \lg n}\right)$ time, proportional to the height of the tree.

For example, to return $S[17]$ with the block tree shown in Figure 7.2 we first see that the block that contains $S[17]$ is not marked, thus we have to follow a backpointer to its first occurrence. In this case, its first occurrence is in the first block but with an offset of 3 inside that block. Thus, we have to descend the block tree through its first child but with an offset of 3 starting at the beginning of that block. That finally maps to the second position of the second block at level 2. That block is marked, so we have to follow the second child pointer with an offset of 1 from the beginning. This maps the the fourth block of level 3, which is not marked. Thus, we follow the backpointer again, which redirects us to the first block of level 3, which contains a single element and maps to the first block of the fourth level with offset 0. This block in the fourth level is marked but is also a leaf, thus we know it has stored the bare string it covers. We finally access that string with the given offset, returning $S[17] = a$.

7.2.3.1 Access and Extract

The original paper on block graphs [GGP11] showed how to store S in $O(z \lg(n/z))$ words such that any substring of length m can be extracted in $O(\lg n + m)$ time. In this section we describe a better result, showing how to extract an arbitrary substring from a block tree with ℓ levels in $O(\ell(m/\lg_\sigma n + 1))$ time.

In order to achieve the improved bounds, we store the first and the last $\lg_\sigma n$ symbols for every block at every level. This adds $\lg n$ bits per block and does not increase the space asymptotically. We extract a substring $S[i \dots i+m-1]$ with $m \leq \lg_\sigma n$ as follows. At the upper level, we check whether $S[i \dots i+m-1]$ spans two blocks or is contained in one single block. If it spans two blocks, then we can extract the part of the string that lies in the first block in constant time, since we have stored the last $\lg_\sigma n$ symbols of the block. The same goes for the part that lies in the second block, since we have stored the first $\lg_\sigma n$ symbols of the block. If $S[i \dots i+m-1]$ is fully contained in a block, then we descend to the next level, either directly if the block exists at the next level, or by following a pointer if the block was copied. We continue recursively in this way, stopping either at the first level at which $S[i \dots i+m-1]$ spans two blocks, or when we reach the last level of the block tree (where all the blocks are of length $\lg_\sigma n$ and their content is stored explicitly). Overall, the time spent is $O(\ell)$. To solve $\texttt{extract}(i, i+m-1)$ when $m > \lg_\sigma n$, we simply divide it into pieces of length $\lg_\sigma n$ (except that the last may be shorter) and extract each piece separately. From this explanation, Theorem 5 and Corollary 2 follow.

Theorem 5. *Given a string S of length n over an alphabet of size σ and a parameter r, we can build a data structure occupying $O\left(zr \lg n \log_r \frac{n \lg \sigma}{z \lg n}\right)$ bits of space that allows extraction of any substring of S of length m in time*

$$O\left(\log_r \left(\frac{n \lg \sigma}{z \lg n}\right) \cdot \left(\frac{m \lg \sigma}{\lg n} + 1\right)\right).$$

Setting $r = (n \lg \sigma / (z \lg n))^\epsilon$, we obtain the following corollary.

Corollary 2. *Given a string S of length n over an alphabet $[1..\sigma]$ and a constant $\epsilon < 1$, we can build a block tree with $O(1/\epsilon)$ levels. The block tree occupies a total of $O\left(\frac{(z \lg n)^{1-\epsilon}(n \lg \sigma)^\epsilon}{\epsilon}\right)$ bits of space, where z is the number of phrases in the LZ77 parsing of S, and allows extraction of any substring of S of length m in time $O\left(\lceil \frac{m}{\lg_\sigma n} \rceil / \epsilon\right)$.*

7.2.3.2 Rank

To support `rank` quickly on S, for each symbol a, we store at each node the number of occurrences of a in the prefix of S preceding the corresponding block. This

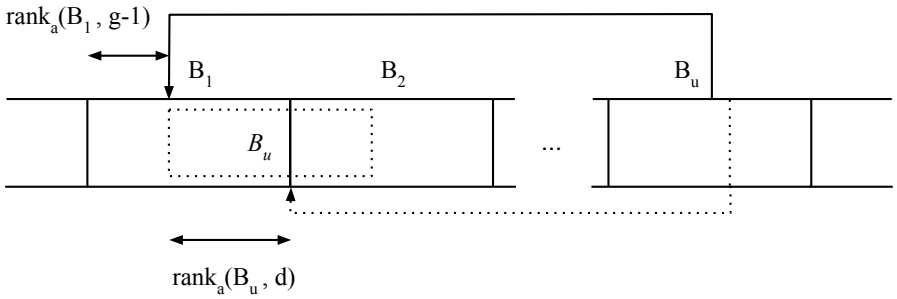


Figure 7.3: To be able to turn a rank query on the unmarked block B_u into a rank query on one of the consecutive pair of marked blocks B_1 and B_2 that contain B_u's first occurrence in S, we store $\texttt{rank}_a(B_1, g-1)$ and $\texttt{rank}_a(B_u, d)$. We already have stored the offset g of the occurrence of B_u in B_1B_2 and we can compute from g and $|B_1|$ the length d of the prefix of B_u that is a suffix of B_1.

takes $O\left(\sigma zr \lg n \log_r \frac{n \lg \sigma}{z \lg n}\right)$ bits of space. This sample of rank values lets us turn any `rank` query on S into a `rank` query on a block in $O(1)$ time. We also store information that lets us turn any `rank` query on an unmarked block into a `rank` query on a marked block in $O(1)$ time. With our sample, we can also turn a `rank` query on a marked block for an internal node, into a `rank` query on one of its children, also in $O(1)$ time. We store a rank data structure for leaves, which takes $O(zr \lg(\sigma)/\lg n) = O(zr)$ words, so we can answer `rank` queries on those blocks directly in $O(1)$ time, and can thus answer `rank` queries on S in $O\left(\log_r \frac{n \lg \sigma}{z \lg n}\right)$ total time.

The information that lets us change any `rank` query on an unmarked block B_u into a `rank` query on a marked block is, first of all, the pointers to the marked block B_1 or consecutive pair B_1 and B_2 of marked blocks containing the first occurrence of B_u, and the offset g of that occurrence in B_1B_2. Secondly, for each symbol a, we store the number $\texttt{rank}_a(B_1, g-1)$ of occurrences of a in the prefix of B_1 before the occurrences of B_u. Notice we can compute, from the offset g and the length of B_1, the length d of the prefix of B_u that is a suffix of B_1. Finally, we store the number $\texttt{rank}_a(B_u, d)$ of occurrences of a in this prefix of B_1. If $i < d$ then $\texttt{rank}_a(B_u, i) = \texttt{rank}_a(B_1, g+i) - \texttt{rank}_a(B_1, g-1)$. If $i = d$ then we have the answer stored. If $i > d$ then $\texttt{rank}_a(B_u, i) = \texttt{rank}_a(B_u, d) + \texttt{rank}_a(B_2, i-d)$. See Figure 7.3 for a graphical example.

7.2.3.3 Select

To support `select` quickly on S, we store a predecessor data structure on the rank samples at the beginnings of the blocks. If we use a trie with branching factor $n^{\epsilon/2}$, we can use $O(zn^{\epsilon})$ words in total for the whole block tree and support predecessor queries in $O(1)$ time. On the other hand, if we use an $O(zr \log_r n)$-space data structure, then predecessor queries take $O(\lg \lg n)$ time. This predecessor data structure lets us turn any `select` query on S into a `select` query on a block, and turn any `select` query on a marked block for an internal node into a `select` query on one of its children. The information we already have stored lets us turn any `select` query on an unmarked block into a `select` query on a marked block: if $j \leq \texttt{rank}_a(B_u, d)$ then $\texttt{select}_a(B_u, j) = \texttt{select}(B_1, j + \texttt{rank}_a(B_1, g-1)) - g$. If $j > \texttt{rank}_a(B_u, d)$ then $\texttt{select}_a(B_u, j) = \texttt{select}(B_2, j - \texttt{rank}_a(B_u, d)) + d$. It follows that answering `select` queries on S takes an $O(\log_r n)$-factor more time than answering a predecessor query.

Combining the bounds for all the queries, we obtain the following result:

Theorem 6. *We can store a string $S[1..n]$ over an alphabet of size σ in*

$$O\left(zr \lg n \log_r \frac{n \lg \sigma}{z \lg n}\right)$$

bits, where z is the number of phrases in the LZ77 parse of S and $r \leq n$, such that we can support extraction of a substring of length m in

$$O\left(\log_r \left(\frac{n \lg \sigma}{z \lg n}\right) \cdot \left(\frac{m \lg \sigma}{\lg n} + 1\right)\right)$$

time. Using a σ factor more space, we can support `rank` *in $O\left(\log_r \frac{n \lg \sigma}{z \lg n}\right)$ time and* `select` *in $O\left(\log_r \left(\frac{n \lg \sigma}{z \lg n}\right) \lg \lg n\right)$ time. In particular, if we use $O(\sigma z n^{\epsilon})$ space, then* `rank` *and* `select` *take $O(1)$ time and extraction takes optimal $O(m \lg(\sigma)/\lg n + 1)$ time.*

7.3 Block Trees for Sequences for Large Alphabets

Note the block tree scales perfectly if we only need to carry out `access` or `extract` operations (Theorem 6). However, if supporting `rank` and `select` queries becomes necessary, a σ factor in the space overhead is induced. This multiplicative penalty may be affordable in case of collections with very small alphabet like bitmaps, otherwise, it quickly goes beyond the space of statistical-based approaches.

Thus, in order to support `rank` and `select` queries for collections with $\sigma > 2$, we have two options: (a) to use a regular block tree or (b) to use a *wavelet tree* or

a *wavelet matrix* (see Section 2.7.1 and Chapter 4 respectively) data structure in which bitmaps are compressed with `BT`.

The first option (a) is of interest in two specific scenarios. It will behave well if the alphabet size is actually small, as we can expect from theory results. But it will also perform well if the repetitiveness of the input data is high, since `BT` will quickly detect repeated blocks, significantly reducing the size of a sequence from a level to the next of the `BT`, and hence the number of `rank` samples, resulting in a very compact data structure.

If none of these scenarios hold, then we can use option (b), since wavelet tree or matrix bitmaps combine well with `BT`, but repetitiveness is quickly destroyed as we move downwards in the wavelet tree, as shown in the previous chapter. Note that destroying repetitiveness is what we try to avoid in Chapter 6 with the introduction of `GCC` and its alphabet partitioning version for large alphabets. However, in this case the alphabet size of the alphabet partitioning internal sequences is larger than what `BT` can manage, and as a consequence using a wavelet tree is the only reasonable option for `BT` when the alphabet σ increases.

7.4 Block Trees in Practice

On one hand, we have to decide which data structures do we actually use to implement and store all the information necessary for the block tree, and how we actually implement all the operations independently of how they are defined in theory.

Regarding the first aspect, we may decide how do we store pointers to siblings, what structures do we use to mark the blocks, and how do we store the samplings for `rank`. In our case, pointer to siblings were implemented using two data structures. If a block S^j points back to a pair of blocks $S^i S^{i+1}$ with offset x from the starting point of S^i, we will store (a) the block identifier, which is the super-index i, and (b) the offset within the beginning of S^{i+1}. Being b the block size and n' the length of the sequence at the current level of the block tree, for (a) we use an array of $\lg(\lfloor n'/b \rfloor + 1)$ bits per element, while for (b) an array of $\lfloor \lg b \rfloor + 1$ bits per pointer.

All samples for `rank` were stored using DACs (Section 2.8). To identify the marked blocks, we use an uncompressed bitmap like `CM` (see Section 2.6).

To determine the depth and arity of the block tree, in practice we use three parameters: The number of levels of the tree (nl), the arity of each node (r), and the block length we want to obtain a the deepest levels (bll). For instance, the block tree of Figure 7.2 was obtained by setting the following configuration: $nl = 4$, $r = 2$, and $bll = 1$.

Regarding the operations, `access` was implemented without the theoretical improvement explained in Section 7.2.3.1. Operation `rank` was implemented as explained in the corresponding section. However, `select` was actually implemented

by carrying out a binary search on the `rank` samples to identify the block that contains the answer. After knowing the block, we use the information stored in it to guide the search until we reach a leaf, which contains the bare string prepared to solve `select` queries efficiently, permitting us to report the proper answer.

7.5 Experimental Results

Along this section, we evaluate the performance of our proposal (`BT`) when applied on bitmaps and on sequences with small alphabets (small $\sigma > 2$).

We used an Intel(R) Xeon(R) E5620 at 2.40GHz with 96GB of RAM memory, running GNU/Linux, Ubuntu 10.04, with kernel 2.6.32-33-server.x86_64. All our implementations use a single thread and are coded in `C++`. The compiler is `g++` version 4.7, with `-O9` optimization. We implemented our solutions inside LIBCDS [Cla]

For our block tree (`BT`) we used several configurations resulting from the combination of its construction parameters, showing in the charts only those points which are pareto-optimal (the same is done with the rest of data structures). We tested block trees with depths $nl = \{5, 6, 7\}$, arity $r = \{2, 4, 8\}$, and block length at leaves of $bll = \{16, 32, 64, 128\}$. We also introduce a variant which we dubbed `WMH.BT`, which is a compressed wavelet matrix (see Chapter 4) in which the bitmaps are compressed either with `BT`, `RRR`, or `CM`, whichever reported the best space performance.

In order to support `rsa` queries on leaves, they were concatenated and stored in an array in which `rank`, `select`, and `access` were solved by brute force.

In general, we used the same data structures (with the same configuration too) as in Section 6.3. To measure the performance on bitmaps, we used techniques `RRR` and `CM` (see Section 2.6), but also `GCC`, all of them with the configurations explained in Section 6.3.

We used the same datasets and technique to extract queries as in Section 6.3. We ran $10,000$ queries, each repeated at least 10 times. The space is measured in bits per symbol (bps) while query time is reported in nanoseconds per query for bitmaps, and microseconds per query for collections with small alphabets ($\sigma > 2$).

7.5.1 Performance on Bitmaps

We firstly evaluate the performance of `BT` when applied on bitmaps. We use synthetic datasets to carry out the experimental evaluation in a controlled scenario, and real datasets to see how it performs on real applications. Figures 7.4, 7.5, and 7.6 show the results for `access`, `rank`, and `select` respectively.

Focusing on synthetic datasets, we can see in any of these three figures that the space performance of `BT` clearly improves as the repetitiveness of the input data does. For the least repetitive synthetic dataset, `DNA.1.st`, `BT` almost obtains the same space performance than statistical and uncompressed proposals, `RRR` and `CM`.

However, as repetitiveness increases, the space performance of BT improves, as we can see in the rest of DNA datasets. However, the solution that systematically obtains the best space performance is GCC. Regarding query times, Figure 7.4 shows that BT is able to match RRR times for operation access, while for rank and select (Figures 7.5 and 7.6), BT performs several times slower than RRR and CM. Comparing BT and GCC on synthetic datasets, the first is systematically faster on the three type of queries, especially on access, in which BT is almost an order of magnitude faster than GCC.

Analyzing the results for real datasets, we can see that BT obtains worse space performance than a plain representation such as CM for escherichia.st, the less repetitive dataset, and for para.st, practically matching the space performance of RRR. For influenza.st and einstein.st, BT uses 20%-40% of the space of CM, and even less when compared with RRR. Comparing BT with GCC, we see that GCC is space-dominant over BT in practically all the real datasets, almost obtaining the same space performance for collections influenza.st and einstein.st. Regarding time performance, as expected, BT is defeated by CM in all operations. If we compare BT with RRR, the situation is slightly different, since BT is extremely efficient at access operation, as we can see in Figure 7.4, slightly beating RRR. For the rest of operations, RRR and CM obtain similar results, both being better than BT by approximately the same margin. When comparing BT with GCC in terms of time, BT overcomes GCC by an order magnitude in access, being several times faster in rank and select operations.

7.5.2 Performance on Sequences With Small Alphabets

As in case of bitmaps, we evaluate the performance of BT in a synthetic and controlled scenario (DNA datasets) as well as with real datasets. Figures 7.7, 7.8, and 7.9 show the performance of access, rank, and select operations respectively.

Focusing on synthetic datasets (DNA), Figures 7.7, 7.8, and 7.9 clearly show how the space performance of BT improves with repetitiveness. For DNA.1, BT obtains worse space performance than statistical competitors like WTH.CM, and WTH.RRR, but for those more repetitive datasets, like DNA.001 and DNA.0001, BT obtains the same performance than grammar-based solutions like GCC. We can also see that WMH.BT obtains practically the same space performance than GCC in the most repetitive synthetic datasets, but is less space efficient when the repetitiveness vanishes, as it is the case of collection DNA.1. Regarding times, we can see in Figure 7.7 that BT excels in access operation. It is even faster than statistical-based approaches (WTH.RRR and WTH.CM) but being up to an order of magnitude more compact than them. Compared to grammar-based approaches, BT is an order of magnitude faster than GCC. Comparing rank and select query times (Figures 7.8 and 7.9), BT almost obtains the same space performance than WTH.RRR, being slightly slower than WTH.CM. When we compare BT with grammar-based approaches, BT is several times faster than GCC, the best grammar based approach. Regarding WMH.BT, it is not as

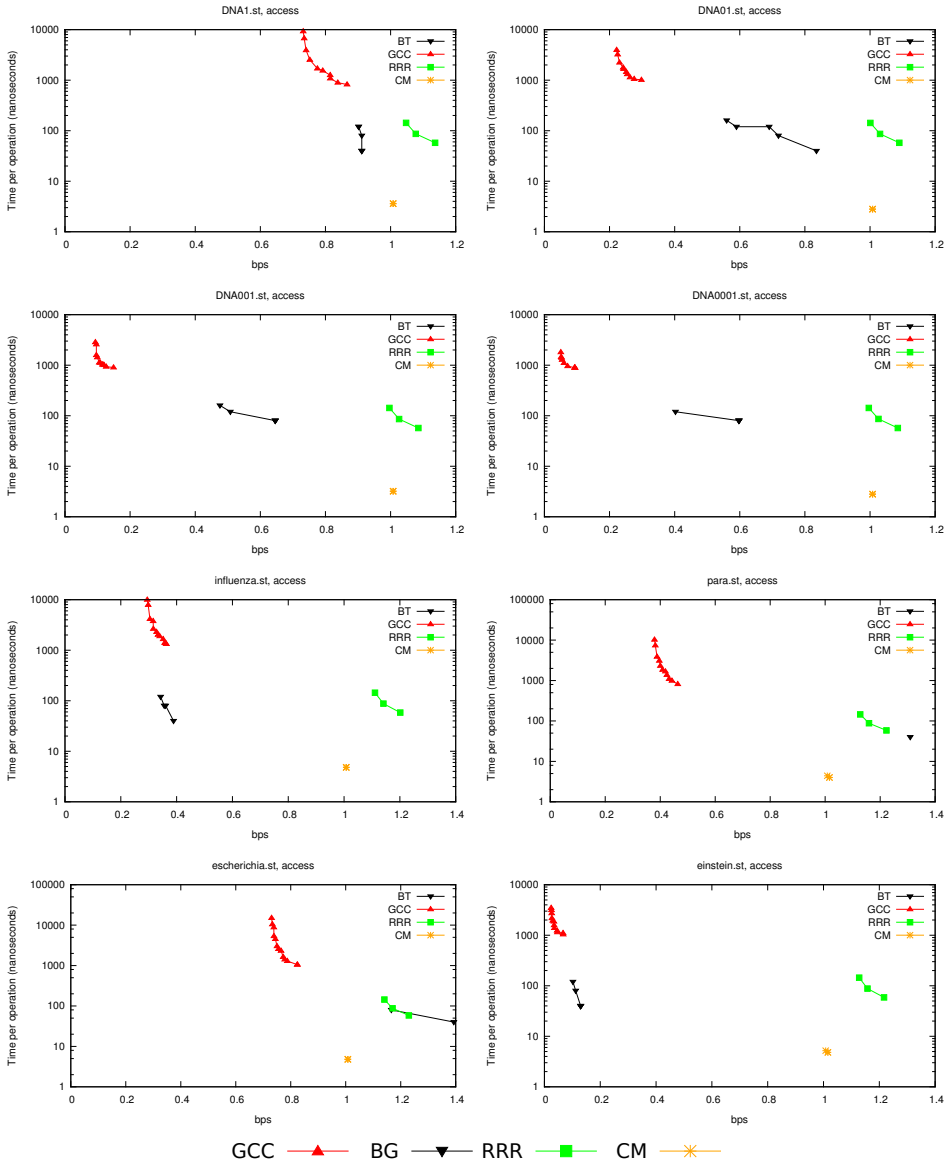


Figure 7.4: Space-time tradeoffs for `access` on bitmaps (note logscale in time).

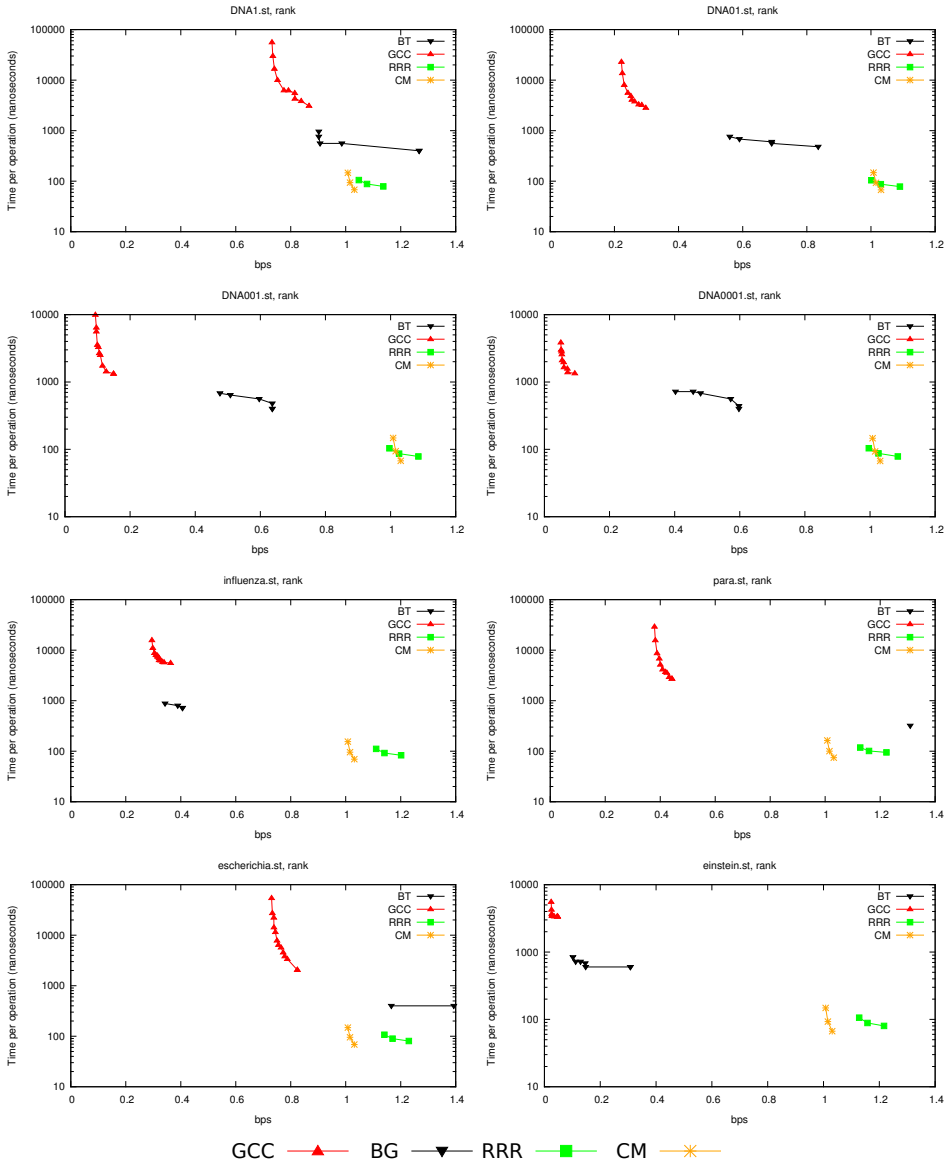


Figure 7.5: Space-time tradeoffs for `rank` on bitmaps (note logscale in time).

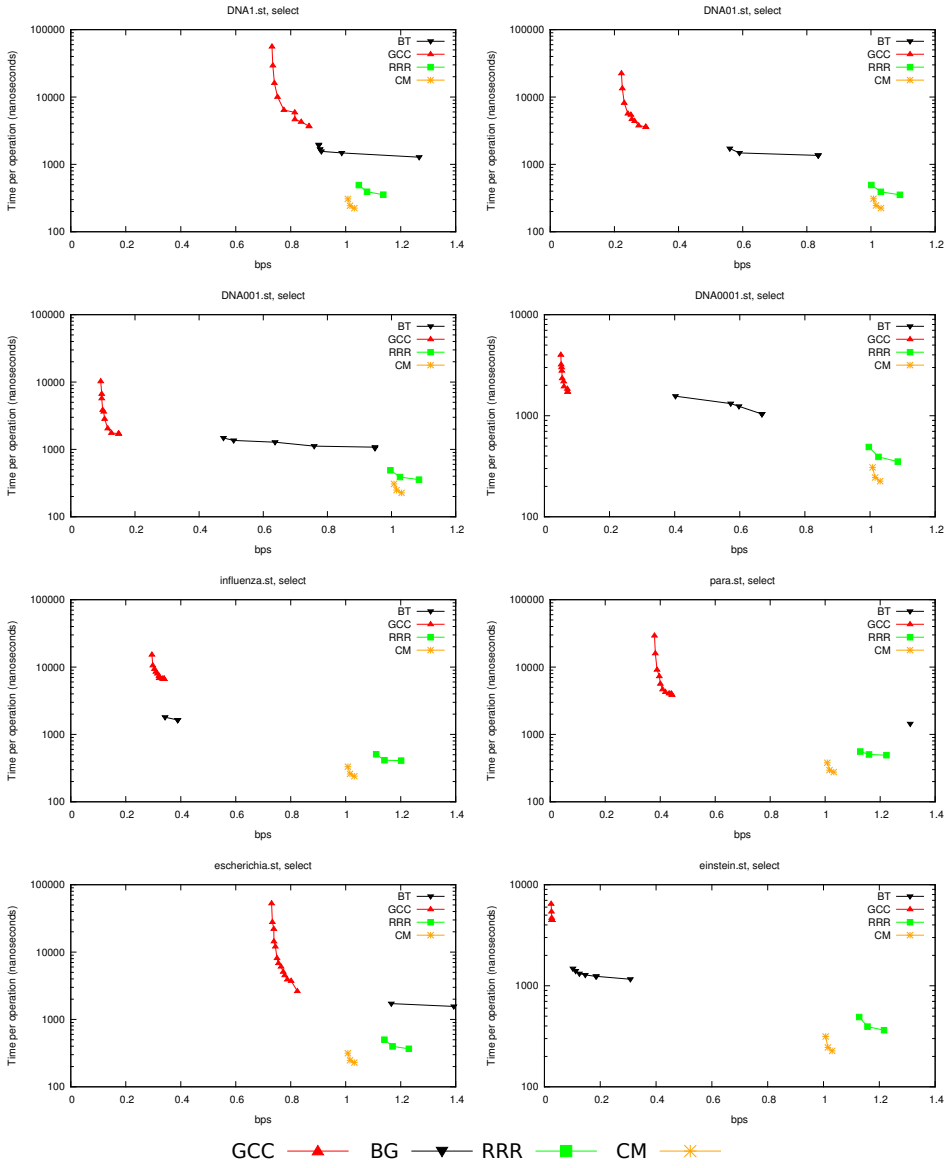


Figure 7.6: Space-time tradeoffs for `select` on bitmaps (note logscale in time).

good as BT, but it practically obtains the same search performance than GCC for all operations.

When we analyze the situation for real datasets, BT obtains a space performance comparable to that of GCC in collections para and einstein, but for influenza, BT needs up to 3 times more space. For escherichia, which is not repetitive, the space performance of BT degrades until reaching that of statistical-based approaches like WTH.CM and WTH.RRR. However, WMH.BT is much more resilient to both the increase of σ (as it is the case of influenza) and to the decrease of repetitiveness (as it is the case of escherichia), but at the cost of being much less efficient on query performance. Again, we can see in Figure 7.7 that BT excels in access performance when compared with statistical-based approaches. For rank, and select (Figures7.8, and 7.9), BT is again slightly slower than statistical-based solutions, but several times faster than grammar-based approaches. Note also WMH.BT obtains similar search performances on rank and select than GCC.

7.6 Discussion

In this chapter we have presented the *Block Tree* (BT), the first *LZ77*-space bounded rsa data structure. We have provided an analysis of its space and time performance, as well as an implementation and a detailed experimental evaluation. Our data structure takes $O\left(\sigma z r \lg n \log_r \frac{n \lg \sigma}{z \lg n}\right)$ bits of space, supports rank and select in $O\left(\log_r \frac{n \lg \sigma}{z \lg n}\right)$ time, and extract in $O\left(\log_r \left(\frac{n \lg \sigma}{z \lg n}\right) \cdot \left(\frac{m \lg \sigma}{\lg n} + 1\right)\right)$ time, m being the length of the extracted snippet. If we use $O(\sigma z n^{\epsilon})$ space ($\epsilon < 1$), we can solve rank and select in $O(1)$ time and extract in optimal $O(m \lg(\sigma)/\lg n + 1)$ time.

In practice and in terms of space, BT is typically overcomed by grammar-based approaches like GCC, although there exist some exceptions. However, when BT uses the same space as GCC (which is not always possible), BT is from several times to an order of magnitude faster. Additionally, BT runs in the same order of magnitude than statistical-based approaches, which is a significant step forward on highly repetitive scenarios. However, the space performance quickly degrades if we increment the alphabet size and the repetitiveness does not increase accordingly. For these scenarios, we have introduced WMH.BT, which is a compressed wavelet matrix with bitmaps compressed with BT (or RRR or CM, depending on the space performance), which also obtains competitive results.

Summing up, both in theory and practice, BT-based approaches should be taken into consideration for highly repetitive sequences. They are faster than their direct competitors when the input data is very repetitive and the alphabet is not large, and, which is more important, they almost match query times of statistical-based approaches while using significantly less space. However, grammar-based options, like those based on GCC, almost systematically obtains better space performance.

One of the challenges that remain open is to continue improving the query

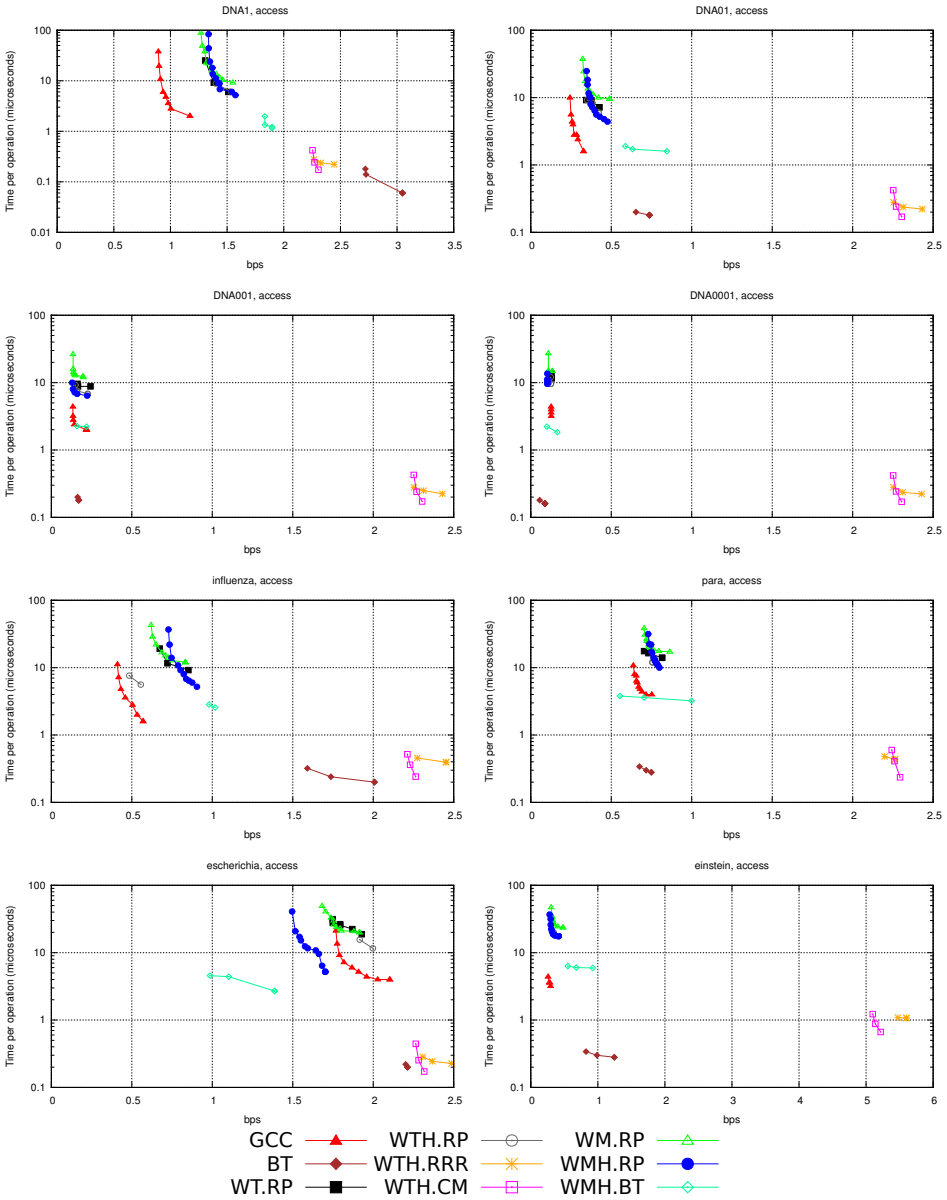


Figure 7.7: Space-time tradeoffs for `access` queries over small alphabets (note logscale in time).

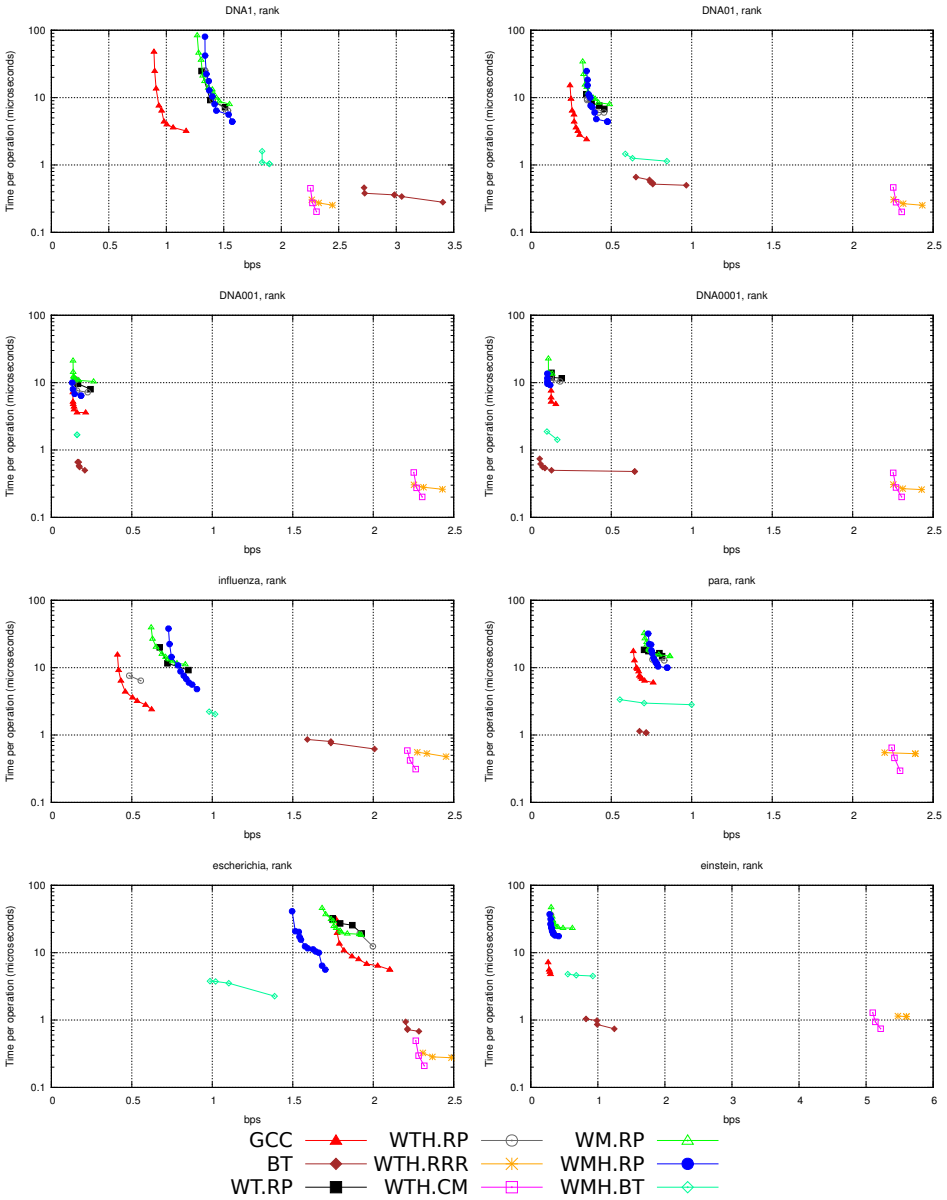


Figure 7.8: Space-time tradeoffs for `rank` queries over small alphabets (note logscale in time).

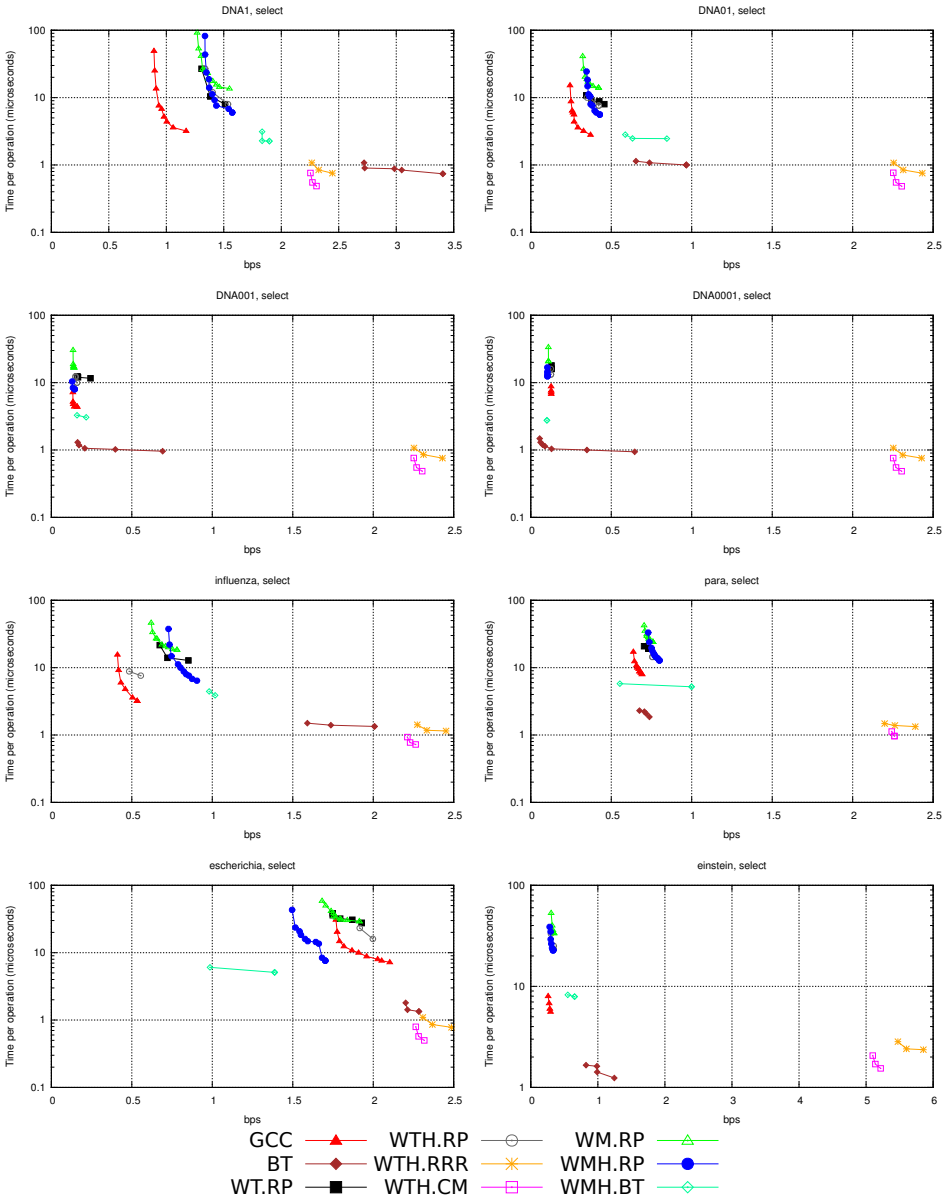


Figure 7.9: Space-time tradeoffs for `select` queries over small alphabets (note logscale in time).

performance, which is probably the biggest limitation of highly repetitive sequence representations. Additionally, we may study new sampling strategies that mitigate the impact of increasing the alphabet size. Alternatively, we may think of extending the algorithm to find previous occurrences of strings to more complex scenarios like graphs.

Chapter 8

Grammar Compressed Trees

Trees are a fundamental data structure in Computer Science. As explained in Section 2.9, a naive pointer based implementation of an ordinal tree $\mathcal{T}_n$ needs $O(n \lg n)$ bits just to store the tree topology. However, succinct representations are able to reduce that space to $2n + o(n)$ bits and answer a wide number of operations in constant time (many of them enumerated in Table 2.2).

This result is of major interest since it has a direct impact on many real applications. For instance, Sadakane [Sad07b] obtained the first (and fastest to date) compressed *suffix tree* representation (refer to Chapter 13 for more information about suffix trees) by storing the tree topology in compact form (among other structures). More recently, Arroyuelo et al. [ACM+15] used compact tree topologies in the SXSI system, which efficiently answers XPath queries on XML collections represented in compressed form.

Even though the $2n+o(n)$ bit may seem a sufficiently compact solution, there exist scenarios in which it may be too much. That is the case, for instance, of suffix trees for highly repetitive datasets (DNA, versioning systems, etc.) in which the suffix tree topology shows up tree isomorphisms that may be potentially compressible. These isomorphisms also appear when we try to index large XML collections or repositories of documents stored in that format. In both contexts, being able to capture that tree repetitiveness is fundamental in order to keep the whole representation in main memory (or even cache), with the known benefits derived.

Grammar compression is a useful tool to deal with that repetitiveness. Although it is generally used in the context of string compression, we can also use grammars that generate trees instead of strings [CDG+07]. The simplest tree grammar is one that replaces repetitions of full trees, so the associated grammar compression seeks for the minimal DAG (Directed Acyclic Graph) equivalent to the tree (an example is shown in Figure 8.1). More powerful variants like *TreeRePair* [LMM13] allow nonterminals with variables or wild-cards, permitting to plug different subtrees where the wild-card is. These grammar compression techniques [MB04, LMM13]

aim at replacing connected subgraphs of the tree, as Figure 8.2 shows. In general, supporting even the most basic traversal operations on these compressed trees is nontrivial, even with the simplest DAG compression.

Alternatively, Bille et al. [BLR+11] sketch an idea that grammar compresses a tree while retaining all the full power of navigational operations of succinct trees. They basically propose to grammar-compress the string of parenthesis that describes the tree topology (recall Section 2.9), attaching the necessary data to the nonterminals in order to support efficient navigation. They prove that this compression is as powerful as the simple DAG tree compression, provided some small fixes are applied to the grammar.

Continuing the line opened by Bille et al. [BLR+11], in this chapter we present the `GCT` (Grammar Compressed Tree), which is a practical grammar compressed tree representation. We rest on the idea of Bille et al. [BLR+11] but using balanced grammars [Sak05] to obtain their same theoretical bounds. Our proposal is of practical nature and has been implemented and tested against one of the best balanced parenthesis representation. In fact, we experimentally show our `GCT` is the most compact tree representation we are aware of, being also competitive in terms of time in some applications.

This chapter is organized as follows: Section 8.1 explains in detail the Balanced Parentheses Tree representation; Section 8.2 details our proposal; Section 8.3 presents the experimental evaluation of the `GCT`; and finally Section 8.4 gives our conclusions.

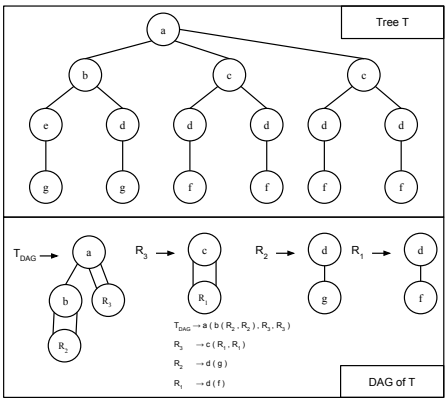


Figure 8.1: A DAG representation of a tree T.

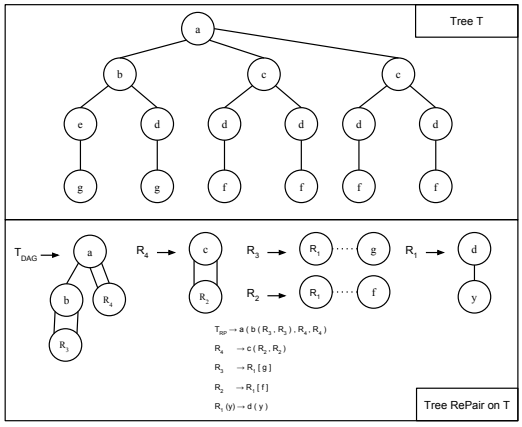


Figure 8.2: *TreeRepair*[LMM13] applied to tree T.

8.1 Related Work

Among many succinct tree representations (see Section 2.9), we describe the proposal of Navarro and Sadakane [NS14] (FF)for *Balanced Parentheses* on which we build our proposal. We choose this representation because it implements a large number of tree operations on top of a simple representation. As explained in Section 2.9, to build a *BP* we traverse the suffix tree in preorder, writing an opening parenthesis when we first arrive at a node, and a closing one when we leave its subtree. Thus a tree of t nodes is represented with $2t$ parentheses, as a binary sequence $P[1, 2t]$. Each node is identified with the offset of its opening parenthesis in P, so we can speak of "node i" to refer to the one represented by $P[i] =' ('$.

We define the *excess* of a position, $E(i)$, as the number of opening minus closing parentheses in $P[1, i]$. Note that $E(i)$ is the depth of node i. Many tree navigation operations can be carried out with two operations related to the excess: $fwd(i, d)$ is the smallest $j > i$ such that $E(j) = E(i) - d$, and $bwd(i, d)$ is the largest $j < i$ such that $E(j) = E(i) - d$. For example, the parenthesis closing the one that opens at position i is at $fwd(i, 1)$, so the next sibling of node i is $j = fwd(i, 1) + 1$ if $P[j] =' ('$, else i is the last child of its parent. Analogously, the previous sibling is $bwd(i-1, 0)+1$ if $P[i-1] =')'$, else i is the first child of its parent. A node i is a leaf if $P[i+1] =')'$, otherwise its first child is $i+1$. The number of nodes in the subtree rooted at i is $(fwd(i, 1) - i + 1)/2$. Node i is an ancestor of j if $i \leq j \leq fwd(i, 1)$. The parent of node i is $bwd(i, 2) + 1$ and the h-th level ancestor is $bwd(i, h + 1) + 1$. The preorder value of a node, $preorder(i)$, is the number of opening parentheses in $P[1, i]$; note that $preorder(i) = (E(i) + i)/2$. The inverse of $preorder$ is $node(j)$, which gives the node with preorder j and is solved analogously to fwd, this time

looking for a certain value of $i+E(i)$. A more complex operation is to find the lowest common ancestor of two nodes, $LCA(i,j)$. Unless one is the ancestor of the other, computing LCA requires operation RMQ (range minimum query) on the virtual array of depths: $RMQ(i,j)$ is the position of a minimum in $E(i)E(i+1)\ldots E(j)$, and then $LCA(i,j) = parent(RMQ(i,j)+1)$. Many other operations are available with the primitives E, fwd, bwd, and RMQ [NS14].

To implement those primitives, the sequence $P[1,2t]$ is cut into blocks of $b\lg t$ parentheses, for a parameter b (we use base 2 logarithms by default). For each block k we store $m[k]$, the minimum excess within the block, and $e[k]$, the total excess within the block. The blocks are the leaves of a perfect binary tree of higher-level blocks, for which we also store $m[k]$ and $e[k]$. See Figure 8.3 for an example.

In this representation, operation $fwd(i,d)$ can be solved in $O(b+\lg t)$ time as follows. Let k be the block where position i belongs. First, we scan P from $i+1$ to the end of the block, to see if the desired excess difference is reached within the block. The block can be scanned by chunks of $(\lg t)/2$ parentheses by using global precomputed tables of just $\sqrt{t}$ entries, which store the total and minimum excess in every possible chunk. If the answer is not inside the block, let d' be d plus the accumulated excess between $i+1$ and the end of the block; then d' is the new excess difference sought to the right of block k (recall that we seek for the smallest $j>i$ such that $E(j) = E(i)-d'$). Now we move to the parent of block k in the balanced tree. If k is the left child of its parent and k' is the right sibling of k, then if $d' > -m[k']$, we know that the desired excess is not reached within block k', thus we set $d' \leftarrow d' + e[k']$ and continue recursively with the parent node of block k. If, instead, k is the right child of its parent, we simply continue recursively with its parent.

This upward traversal continues until we find a right sibling k' for which $d' \leq -m[k']$, thus the desired excess difference is reached within block k'. Now we start a downward traversal. We check whether the difference is reached inside the left child k'' of k': if $d' \leq -m[k'']$, then we descend to k''; otherwise we set $d' \leftarrow d' + e[k'']$ and descend to the right child of k'. When we finally arrive at a leaf block, we complete the operation $fwd(i,d)$ by scanning its parentheses from the beginning of the block until we reach excess difference d'.

Operations bwd and RMQ are solved analogously, and computing $E(i)$ is simpler; see Navarro and Sadakane [NS14] for more details[1]. By using, for example, $b = \Theta(\lg t)$, one obtains $O(\lg t)$ time for all the operations and $2t+o(t)$ bits to store the the parentheses plus the balanced tree of $m[]$ and $e[]$ values.[2]

[1] They describe a variant where the $e[]$ and $m[]$ values are absolute, not relative to the block; our description here is more similar to their dynamic variant. Also, they store a few more values to support other operations not usually required on suffix trees, and thus not considered in this paper.

[2] The theoretical proposal of Navarro and Sadakane [NS14] obtains constant times, but the practical implementation of Arroyuelo et al. [ACNS10] reaches logarithmic times.

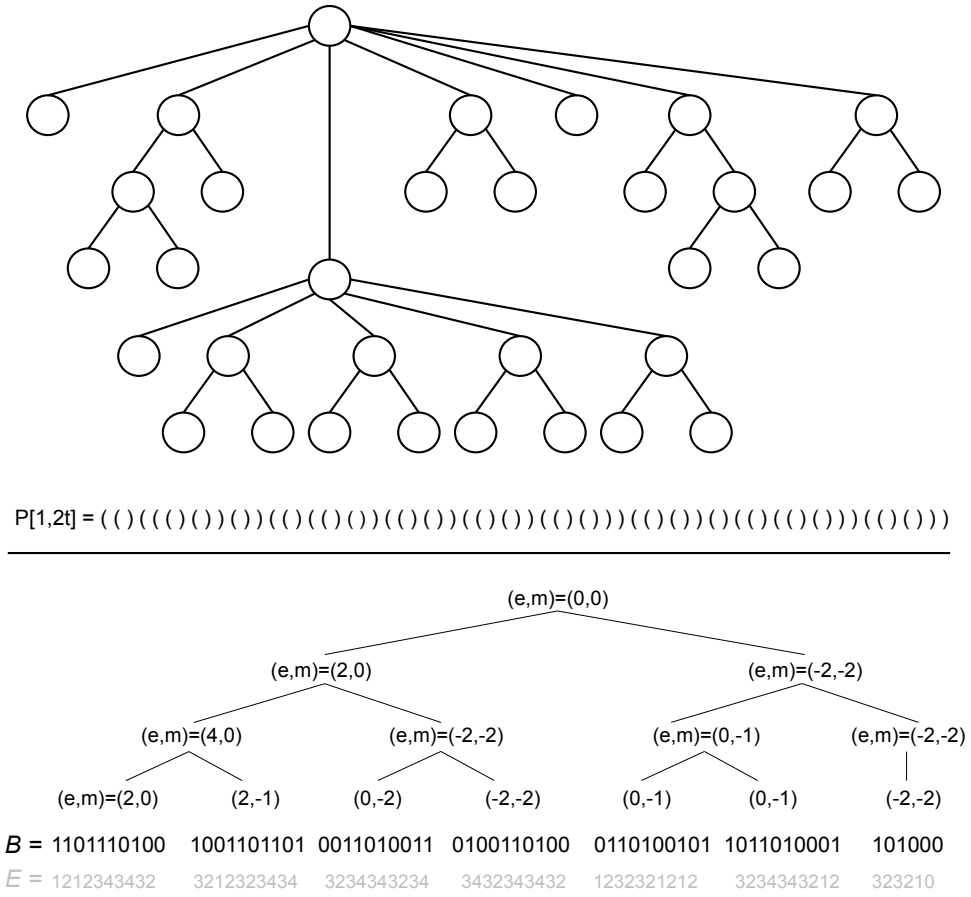


Figure 8.3: On the top, an ordinal tree with its BP representation $P[1, 2t]$. On the bottom, the data structure for the succinct representation of a parentheses sequence $P[1, 2t]$. It is formed by bitvector $B[1, 2t]$, which represents P with 1-bits for the '('s and 0-bits for the ')'s, and the tree of block summaries on top of it. We show in gray the values $E(i)$, which are not represented explicitly.

8.2 Grammar Compressed Tree

In this section we explain our proposal to grammar compress trees, developing the theoretical idea introduced by Bille et al. [BLR+11] and focusing only on tree topologies rather than on general trees. The basic idea relies on first obtaining the balanced-parentheses representation of the tree and then enhance it so that we can solve all operations a succinct representation does. We start by explaining the `GCT` structure, focusing on the algorithmics related with the tree operations on the compressed representation afterwards.

8.2.1 GCT Structure

Let $R[1, r]$ be the rules (including initial rules generating the terminals '(' and ')') and $C[1, c]$ the final sequence resulting from applying *RePair* (see Section 5.4) compression to the parentheses sequence $P[1, 2t]$. We use a version of *RePair* that yields balanced grammars (i.e., of height $O(\lg t)$) in most cases.[3] We describe how we store R and C.

8.2.1.1 Storing the Rules R

A plain storage of the rules $R[1, r]$ requires $2r \lg r$ bits, as a simple array $R[k] = (i, j)$ meaning $R[k] \to R[i]R[j]$. Although this plain storage is sufficiently compact for many applications, as it is the case of grammar compressed sequences (see Section 6), grammar compressed trees are an exception. Tree operations are more complex than just `rsa` queries, which means we typically need more additional data structures to have a fully-functional data structure. Additionally, when $\sigma = 2$, which is the case of balanced parentheses representations, the parentheses sequences need n bits, and thus the impact of storing the dictionary may be larger than for sequences with $\sigma > 2$. This space overhead may limit the use of the data structure, which means we should try to save as much space as possible, but still having reasonable query performance. Therefore, and in order to mitigate the space impact of storing the dictionary R, instead of a plain storage of R, we will use the technique described by Tabei et al. [TTS13], which uses only $r \lg r + O(r)$ bits and permits extracting the right hand of any rule in time $O(\lg r)$.

The grammar is now seen as a DAG where the nodes are the nonterminals, and each rule $R_k \to R_i R_j$ induces two arrows, from R_k to R_i and from R_k to R_j. Now all the arrows from nodes to their left children, seen backward, form a tree T_L, and those to their right children, seen backward, form a tree T_R. We represent T_L and T_R, using a succinct tree representation, in $O(r)$ bits (recall Section 8.1). The identifiers of the nonterminals will be their preorder values in T_L: rule R_k will refer to the node with preorder k in T_L.

[3] From `www.dcc.uchile.cl/gnavarro/software`. There exist algorithms that ensure balanced grammars [Sak05], but they are more complicated.

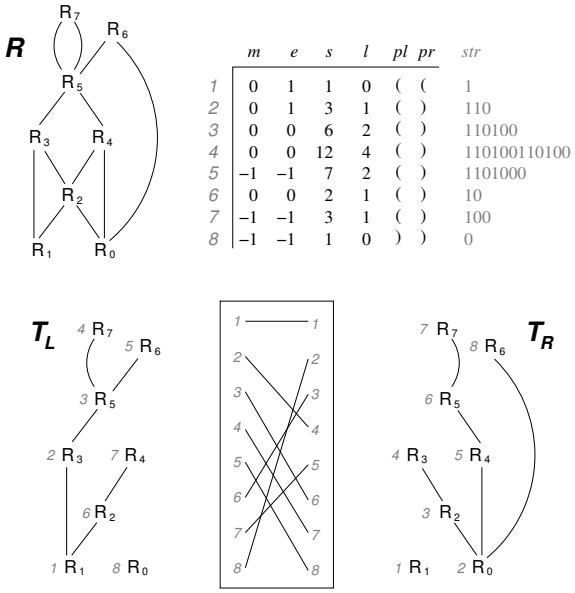


Figure 8.4: The grammar R of Figure 5.2 in DAG form (top left), its representation as two trees T_L and T_R (with preorders in slanted gray) plus a permutation mapping preorders (bottom), and the data we store on the nonterminals (top right, node identifiers correspond to preorders in T_L).

In addition, we need to map between a preorder value of a nonterminal in T_L and the preorder value of the same nonterminal in T_R, and back. We use a practical technique by Munro et al. [MRRR03] that represents a permutation π of $\{1, 2, \ldots, r\}$ using $r \lg r + O(r)$ bits. It stores the array $\pi = [\pi(1), \pi(2), \ldots, \pi(r)]$ explicitly (thus $\pi(i)$ is computed in constant time), and adds $O(r)$ bits of data that allows computing any $\pi^{-1}(i)$ in time $O(\lg r)$. Figure 8.4 illustrates this representation.

The main operation carried out on this representation is, given a nonterminal k such that $R_k \to R_i R_j$, find i and j. The procedure is as follows:

1. Compute $x_L \leftarrow \mathit{node}(T_L, k)$, the node of T_L that represents R_k.
2. Find $y_L \leftarrow \mathit{parent}(T_L, x_L)$, the parent of x_L, which represents R_i in T_L.
3. Compute $i \leftarrow \mathit{preorder}(T_L, y_L)$, the identifier of nonterminal R_i.
4. Map $k_R \leftarrow \pi(k)$, the preorder of R_k in T_R.
5. Compute $x_R \leftarrow \mathit{node}(T_R, k_R)$, the node of T_R that represents R_k.

6. Find $z_R \leftarrow \mathit{parent}(T_R, x_R)$, the parent of x_R, which represents R_j in T_R.
7. Compute $j_R \leftarrow \mathit{preorder}(T_R, z_R)$, the preorder of z_R in T_R.
8. Map back $j \leftarrow \pi^{-1}(j_R)$, the identifier of nonterminal R_k.

In practice, the structure described in Section 8.1 is too powerful for the few operations we need on T_L and T_R. We use instead the so-called LOUDS representation [Jac89] (Section 2.10), which supports operation *parent* and an equivalent to operations *preorder* and its inverse *node* (that is, it assigns a distinct number in $[1, r]$ to each node, although it is not its preorder value). The LOUDS representation is smaller and faster than the one described in Section 8.1, albeit it supports fewer operations.

8.2.1.2 Storing Information on the Rules

We enrich the grammar R with additional information on the nonterminals, to allow for fast operations on the represented tree. For each nonterminal R_k, and being $\mathit{str}(R_k)$ be the sequence of bits expanded by R_k, we store the following arrays (see Figure 8.4).

1. $m[k]$, the minimum excess of '('s in $\mathit{str}(R_k)$. It holds $m[k] \leq 0$ because the excess of the empty prefix of the string is always 0.
2. $e[k]$, the total excess of $\mathit{str}(R_k)$.
3. $s[k] = |\mathit{str}(R_k)|$, the length of the string R_k generates.
4. $l[k]$, the number of leaf nodes represented inside $\mathit{str}(R_k)$, that is, the number of substrings of the form '()' (or '10', in bits) present in $\mathit{str}(R_k)$.
5. $pl[k]$ and $pr[k]$, the leftmost and rightmost parentheses (bits) in $\mathit{str}(R_k)$.

Since $e[k]$ can be positive or negative, we rather store $e[k] - m[k] \geq 0$. On the other hand, $m[k]$ is stored as $-m[k] \geq 0$. Since $s[k] \geq 2l[k] - m[k]$, we represent $s[k] - 2l[k] + m[k] \geq 0$ to induce smaller numbers. Many values in these arrays are expected to be small (and even smaller after these transformations), so we store them using a variable-length representation that uses fewer bits for smaller numbers. The representation we choose, called directly addressable codes (DACs) (see Section 2.8), allows direct access to any cell value (we use the DAC variant that uses minimum space). Of course, the arrays pl and pr are stored using one bit per cell.

To further save space, only some nonterminals k will store this information. Let $R_k \rightarrow R_i R_j$. Then, it holds $m[k] = \min(m[i], e[i] + m[j])$, $e[k] = e[i] + e[j]$, $s[k] = s[i] + s[j]$, $l[k] = l[i] + l[j] + [1 \text{ if } pr[i] =' (' \wedge pl[j] =')']$, $pl[k] = pl[i]$, and $pr[k] = pr[j]$. These recurrences allow us computing the desired values for nonterminals R_k that do not store them. We use a technique [NPV14] that, given

a parameter y, chooses a set of nonterminals that will store the array values, guaranteeing that we will never recursively expand more than y nonterminals in order to obtain any such value.

8.2.1.3 Storing the Array C

Sequence $C[1, c]$ is stored as an array of nonterminals, that is, the corresponding preorder values in T_L, using $c \lg r$ bits. In addition, the parentheses sequence P will be sampled every z positions. For the sth sample ($s \geq 0$), we store the following values.

1. $C_p[s]$, the position in C whose expansion contains $P[zs+1]$, that is, $C_p[s] = \min\{w, \sum_{v=1}^{w} |str(C[v])| > zs\}$.
2. $C_o[s]$, the distance from $zs+1$ to the beginning of $C[C_p[s]]$, that is, $C_o[s] = zs - \sum_{v=1}^{C_p[s]-1} |str(C[v])|$.
3. $C_e[s] = \sum_{v=1}^{C_p[s]-1} e[C[v]]$, the cumulative excess up to the beginning of block $C[C_p[s]]$.
4. $C_l[s]$, the number of leaves (occurrences of '10') up to the beginning of $C[C_p[s]]$, that is, $C_l[s] = l[C[1]] + \sum_{v=2}^{C_p[s]-1}(l[C[v]] + [1 \text{ if } pr[C[v-1]] =' (' \wedge pl[C[v]] =')'])$ if $C_p[s] > 1$, and $C_l[s] = 0$ otherwise.

On top of this array of samples, we form a balanced binary tree, where each node stores the minimum excess reached within the range of C it covers (the sth leaf covers the range $P[z(s-1)+1, zs]$). This excess is represented in absolute form, not relative to the range of blocks. Figure 8.5 illustrates the representation of C.

This sampled data adds $O((t \lg t)/z)$ bits to the $c \lg r$ bits used to store sequence C. If we choose $z = \Theta((t \lg t)/c)$ and y large enough for the marked blocks be $O(r/\lg t)$, then the space of our grammar representation is $(r+c)\lg r + O(r+c)$, which is asymptotically equal to the space of a plain grammar-compressed representation the sequence $P[1, 2t]$. Since the c nonterminals in C expand to $2t$ characters, we will be able to scan the cells of C between two samples in $O(\lg t)$ time on average[4]. We cannot bound the value of y required to have $O(r/\lg t)$ samples, however, but our experiments will show that reasonable space/time tradeoffs are achieved.

8.2.2 Basic Operations

We start by describing some basic operations on the `GCT`. The following procedure computes $E(p)$, the excess in $P[1, p]$, which is needed to compute $tDepth(p) = E(p)$ and $preorder(p) = (p + E(p))/2$.

[4]This can be made worst-case by regularly sampling $C[1, c]$ instead of $P[1, 2t]$, but this entails a binary search to find the sample corresponding to a position in P, and turns out to be slower than the sampling we chose, for the same space usage.

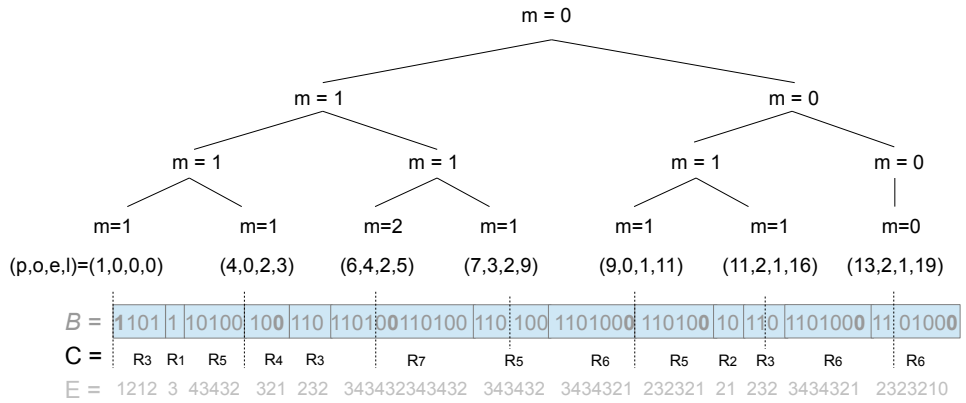


Figure 8.5: Tree structure built on top of the C array of Figure 5.2, for a sampling step of $z = 10$. The positions of B in bold show where is the minimum excess reached, within each block.

1. We compute $s \leftarrow \lfloor (p-1)/z \rfloor$ and then position $u \leftarrow C_p[s]$ in C. Then $C[u]$ starts in P after position $q \leftarrow sz - C_o[s]$, and the excess up to q is $e \leftarrow C_e[s]$.

2. We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$ and $e \leftarrow e + e[k]$, where we remind that $s[k]$ and $e[k]$ are the total length and excess, respectively, of $str(R_k)$. We stop at the position $C[v]$ where q would exceed p if we processed v.

3. Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \to R_i R_j$ (i and j are found with the method described in Section 8.2.1.1). If $q + s[i] \leq p$, then we add $q \leftarrow q + s[i]$ and $e \leftarrow e + e[i]$, and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, we know the excess e up to position $q = p$. Then we return $E(p) = e$.

If we store $e[]$ and $m[]$ values for all the nonterminals, then the sequential traversal in point 2 requires on average $O(\lg t)$ time, as discussed at the end of Section 8.2.1.3. Point 3 takes time $O(\lg^2 t)$, because the grammar is balanced and thus has height $O(\lg t)$, and each time we expand $R_k \to R_i R_j$ in the downward traversal we take time $O(\lg t)$ to find i and j with the representation of Section 8.2.1.1. Thus the total time is $O(\lg^2 t)$. Instead, if we sample the values $e[]$ and $m[]$ with parameter y, then each computation of those values requires $O(y)$ symbol expansions, each of which still costs $O(\lg t)$ time. This raises the total time to $O(y \lg^2 t)$.

Another basic operation is to find the value of $P[p]$. This is necessary for *isLeaf* and *fChild*, and also participates in operations *nSibling* and *pSibling*. The recursion

described for operation $E(p)$ ends up at a terminal in point 3, which is $P[p+1]$. Therefore, the following variation returns $P[p]$.

1. We compute $s \leftarrow \lfloor (p-1)/z \rfloor$ and then position $u \leftarrow C_p[s]$ in C. Then $C[u]$ starts in P after position $q \leftarrow sz - C_o[s]$.

2. We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$. We stop at the position $C[v]$ where q would reach p if we processed v.

3. Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \rightarrow R_i R_j$. If $q + s[i] < p$, then we add $q \leftarrow q + s[i]$ and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, it is at position $q = p$ in P, so we return it.

8.2.3 Operations *fwd* and *bwd*

These are the two most important operations on the `GCT`, needed to implement *parent*, *nSibling*, *pSibling*, *ancestor*, *subtree*, and *tAncestor*. They also participate in operations *LCA*, *sLink*, *LAQs*, *sDepth*, *LAQs*, and *child*. We describe how to solve operation $fwd(p, d)$, where p is a position in P. This follows the same spirit of the description of the operation on a balanced tree, recall Section 8.1. The scheme of the algorithm, with the corresponding steps, is depicted in Figure 8.6. Operation $bwd(p, d)$ is analogous.

1. We compute $s \leftarrow \lfloor (p-1)/z \rfloor$ and then position $u \leftarrow C_p[s]$ in C. Then $C[u]$ starts in P after position $q \leftarrow sz - C_o[s]$, and the excess up to q is $e \leftarrow C_e[s]$.

2. We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$ and $e \leftarrow e + e[k]$. We stop at the position $C[v]$ where q would exceed p if we processed v.

3. Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \rightarrow R_i R_j$. If $q + s[i] \leq p$, then we add $q \leftarrow q + s[i]$ and $e \leftarrow e + e[i]$, and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, we have the excess e up to position $p = q$. Now we start looking for a negative difference d of excess to the right of position q.

4. We traverse back (returning from recursion) the path in the grammar followed in point 3. If we went towards the right child R_j of a rule $R_k \rightarrow R_i R_j$, we just return. If, instead, we went towards R_i, then we check whether $d \leq -m[j]$. If so, the answer is within R_j, otherwise we add $q \leftarrow q + s[j]$ and $d \leftarrow d + e[j]$, and return to the parent in the recursion.

5. If in the previous point we have established that the answer is within a nonterminal R_j, we traverse the expansion of $R_j \rightarrow R_l R_m$. If $d > -m[l]$, then

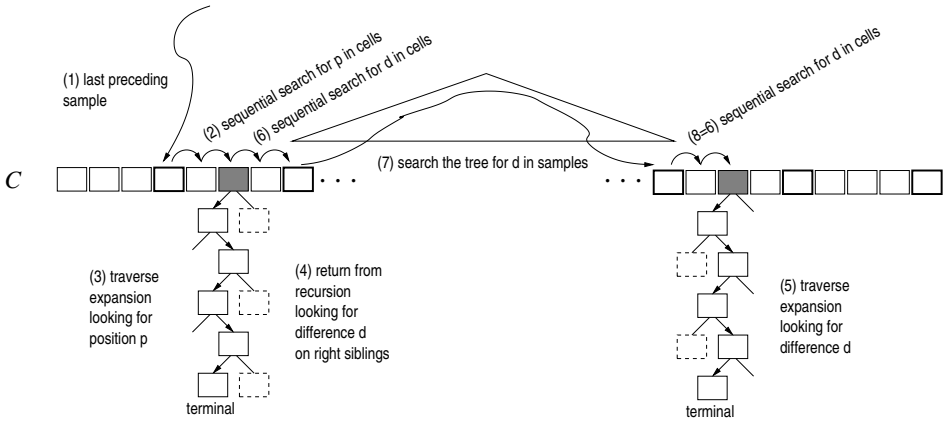


Figure 8.6: General scheme of the algorithm for $fwd(p, d)$, assuming d is found after another sample. Sampled blocks have thick borders. Grayed blocks are nonterminals that become partially expanded; dashed ones are skipped after considering their $e[]$ and $m[]$ values. The middle tree is that of Figure 8.5.

we add $q \leftarrow q + s[l]$ and $d \leftarrow d + e[l]$, and continue recursively with R_m; else we continue recursively with R_l. When we reach a leaf, the answer is q.

6. If in point 4 we return from the recursion up to the root symbol $C[v]$ without yet finding the desired excess difference d, we update $e \leftarrow e + e[C[v]]$ and scan the nonterminals $C[v + a]$ for $a = 1, 2, \ldots$, increasing $q \leftarrow q + s[C[v + a]]$, $e \leftarrow e + e[C[v + a]]$, and $d \leftarrow d + e[C[v + a]]$, until finding an a such that $d \leq -m[C[v + a]]$. At this point we look for the final answer within the nonterminal $C[v + a]$ just as in point 5.

7. If we reach the next sampled position, $q \geq sz$, without yet finding the answer, we jump to the $\lfloor q/z \rfloor$th leaf of the balanced tree we built on the samples of C (the leftmost leaf is the 0th), and traverse it upwards until the current node has a right sibling whose (absolute) minimum value m satisfies $e - d \geq m$. Then we descend from that right sibling. If its left child's minimum value m_l satisfies $e - d \geq m_l$, we descend to the left child, otherwise to the right child.

8. We eventually reach the s'th leaf of the tree, thus we know that the desired answer can be found from position $v \leftarrow C_p[s']$ in C. Note that $C[v]$ starts after position $q \leftarrow s'z - C_o[s']$ in P, and the excess up to q is $C_e[s']$. Thus we recompute $d \leftarrow d + C_e[s'] - e$ and $e \leftarrow C_e[s']$, and continue traversing the cells $C[v + a]$ sequentially, for $a \geq 0$, just as in point 6 (and eventually finishing as in point 5).

The complexity of this procedure is the same as for the simpler operations. The only new cost is $O(\lg t)$ to traverse the balanced tree, which is not significant.

8.2.4 Operation *RMQ*

This operation is fundamental for the *LCA* queries, among others. To solve $RMQ(p, p')$ we traverse all the $O(\lg t)$ grammar nodes between positions p and p' and locate the point where the minimum excess occurs.

1. We compute $s \leftarrow \lfloor (p-1)/z \rfloor$ and then position $u \leftarrow C_p[s]$ in C. Then $C[u]$ starts in P after position $q \leftarrow sz - C_o[s]$, and the excess up to q is $e \leftarrow C_e[s]$.

2. We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$ and $e \leftarrow e + e[k]$. We stop at the position $C[v]$ where q would reach p if we processed v.

3. Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \to R_i R_j$. If $q + s[i] < p$, then we add $q \leftarrow q + s[i]$ and $e \leftarrow e + e[i]$, and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, we are at position p and initialize $min \leftarrow e$. Now we update $e \leftarrow e + 1$ if the terminal is a '(' or $e \leftarrow e - 1$ if the terminal is a ')'. Finally, we update $min \leftarrow \min(min, e)$ and $q \leftarrow q + 1$.

4. We traverse back (returning from recursion) the path in the grammar followed in point 3. If we went towards the right child R_j of a rule $R_k \to R_i R_j$, we just return. If, instead, we went towards R_i, then we check whether $q + s[j] > p'$. If so, position p' is within R_j, otherwise we update $min \leftarrow \min(min, e + m[j])$, $q \leftarrow q + s[j]$, and $e \leftarrow e + e[j]$, and return to the parent in the recursion.

5. If in the previous point we have established that p' is within a nonterminal R_j, we traverse the expansion of $R_j \to R_l R_m$. If $q + s[l] \leq p'$, then we update $min \leftarrow \min(min, e + m[l])$, $q \leftarrow q + s[l]$, and $e \leftarrow e + e[l]$, and continue recursively with R_m; else we continue recursively with R_l. When we reach a leaf, we have $q = p'$ and the minimum excess is min.

6. If in point 4 we return from the recursion up to the root symbol $C[v]$ without yet reaching position p', we scan the nonterminals $C[v+a]$ for $a = 1, 2, \ldots$, updating $min \leftarrow \min(min, e + m[C[v+a]])$, $q \leftarrow q + s[C[v+a]]$, and $e \leftarrow e + e[C[v+a]]$, until finding an a such that $q + s[C[v+a]] > p'$. At this point we complete the calculation of min within the nonterminal $C[v+a]$ just as in point 5.

7. If we reach the next sampled position, $q \geq sz$, without yet reaching position p', we jump to the $\lfloor q/z \rfloor$th leaf of the balanced tree we built on the samples of C, and traverse it upwards until the the current node has a right sibling that covers position p'. Along the upward traversal, for each right sibling (that

does not yet cover p') with minimum value m_r, we set $min \leftarrow \min(min, m_r)$. Once we find a right sibling that covers p' we descend from it. If its left child covers p', we just descend to the left, else we descend to the right and set $min \leftarrow \min(min, m_l)$, where m_l is the minimum value stored at the left child.

8. We eventually reach the s'th leaf of the tree, thus we know that p' can be found from position $v \leftarrow C_p[s']$ in C. Note that $C[v]$ starts after position $q \leftarrow s'z - C_o[s']$ in P, and the excess up to q is $e \leftarrow C_e[s']$. Thus we continue traversing the cells $C[v+a]$ sequentially, for $a \geq 0$, just as in point 6.

9. We finally have the min value, but not the position where it was reached. If min was set at a node of the balanced tree, we descend by its left or right children, whichever matches the minimum value of its parent, until reaching a leaf s''. Then we scan $k \leftarrow C[C_p[s''], \ldots]$, starting with $q \leftarrow s''z - C_o[s'']$, $e \leftarrow C_e[s'']$ and $m \leftarrow 0$, updating $q \leftarrow q + s[k]$, $m \leftarrow \min(m, e + m[k])$ and $e \leftarrow e + e[k]$, until we reach the value $e + m = \min$ for some k (before q reaches the next sampled block).

10. Either because we computed it in point 9, or because min was reached within a nonterminal R_k starting after position q, we proceed as follows. If $R_k \rightarrow R_i R_j$, then if $m[k] = m[i]$, we continue recursively with R_i; otherwise we set $q \leftarrow q + s[i]$ and continue recursively with R_j. When we reach a terminal, the answer is $RMQ(p, p') = q$.

8.2.5 Mapping with Leaves

Many applications requires the ability to count the number of leaves up to some position $P[p]$, and to find the lth leaf in P. The storage of $l[k]$, $pl[k]$ and $pr[k]$ serves this purpose. We first show how to compute the number of leaves up to position p.

1. We compute $s \leftarrow \lfloor (p-1)/z \rfloor$ and then position $u \leftarrow C_p[s]$ in C. Then $C[u]$ starts in P after position $q \leftarrow sz - C_o[s]$, and the number of leaves up to that position is $l \leftarrow C_l[u]$. We set $pr \leftarrow pr[C[u-1]]$ (if $u > 0$, otherwise $pr \leftarrow')'$).

2. We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$, $l \leftarrow l + l[k] + [1 \text{ if } pr =' (' \wedge pl[k] =')']$, and $pr \leftarrow pr[k]$. We stop at the position $C[v]$ where q would exceed p if we processed v.

3. Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \rightarrow R_i R_j$. If $q + s[i] \leq p$, then we add $q \leftarrow q + s[i]$, $l \leftarrow l + l[i] + [1 \text{ if } pr =' (' \wedge pl[i] =')']$, and $pr \leftarrow pr[i]$, and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, we know the number of leaves l up to position p. Then we return l.

Finding the lth leaf is the inverse of the above operation.

1. We binary search C_l to find the largest s such that $C_l[s] < l$. This points to position $u \leftarrow C_s[s]$ in C, which starts after $q \leftarrow sz - C_o[s]$ in P. The number of leaves up to position q is $l' \leftarrow C_l[s]$, and we set $pr \leftarrow pr[C[u-1]]$ (if $u > 0$, otherwise $pr \leftarrow')'$).

2. We sequentially traverse the nonterminals $k \leftarrow C[u \ldots]$, updating $q \leftarrow q + s[k]$, $l' \leftarrow l' + l[k] + [1 \text{ if } pr =' (' \ \wedge \ pl[k] =')']$, and $pr \leftarrow pr[k]$. We stop at the position $C[v]$ where l' would reach l if we processed v.

3. Now we navigate the expansion of the nonterminal $k = C[v]$. Let $R_k \rightarrow R_i R_j$. If $l' + l[i] + [1 \text{ if } pr =' (' \ \wedge \ pl[i] =')'] < l$, then we set l' to this value, update $q \leftarrow q + s[i]$, set $pr \leftarrow pr[i]$, and continue recursively with R_j; otherwise we continue recursively with R_i. When we finally reach a terminal, we know the number of leaves up to position q is $l' < l$ and that their number reaches l at position $q + 1$. Then we return q, the starting position of the opening parenthesis that starts the lth leaf.

8.3 Experimental Results

8.3.1 Environmental Set-up and Datasets

We used an Intel(R) Xeon(R) E5620 at 2.40GHz with 96GB of RAM memory, running GNU/Linux, Ubuntu 10.04, with kernel 2.6.32-33-server.x86_64. All our implementations use a single thread and are coded in `C++`. The compiler is `g++` version 4.7, with `-O9` optimization. We implemented our solutions inside LIBCDS [Cla] and use Navarro's implementation of RePair (`www.dcc.uchile.cl/gnavarro/software/repair.tgz`).

We used various *suffix tree topologies* of several repetitive datasets described in Section 5.6. Concretely, we use collections `DNA.1`, `DNA.01`, `DNA.001`, `DNA.0001`, `influenza`, `escherichia`, `para`, and `einstein` (all DNA datasets except `einstein`).

We use three different strategies based on previous work [NS14, ACN13] to extract queries, each defined to correctly measure the performance of a specific type of queries. These strategies are described as follows:

(a) We randomly pick a leaf and collect and report all those nodes in the path from that leaf to the root.

(b) We randomly select couples of leaves.

(c) We randomly select leaves with *tDepth* values larger than 10, and then we report that leaf and a value in the range $[1, tDepth - 1]$.

Queries for operations *fChild*, *tDepth*, *nSibling*, and *parent* were extracted following strategy (a); those for *LCA* following strategy (b); and those for

tAncestor following strategy (c). For each operation, we averaged each data point over 10,000 queries.

8.3.2 Parameterizing the Data Structures

We compared our `GCT` with an implementation of the Balanced Parentheses of Sadakane [ACNS10] (Sada). We used various combinations of parameters y and z for the `GCT`, concretely, $y \in \{2^0, 2^1, 2^2, 2^4, 2^8\}$ and $z \in \{2^8, 2^{10}, 2^{12}, 2^{14}\}$. In most cases, this implies leaving y at sampling every nonterminal and using z to reduce the space. The combination of those parameters would generally yield a cloud of points in the charts, although we only show those space-time dominant. We used the balanced version of *RePair* (see Section 5.4), which consistently gave us better results.

8.3.3 Space Performance

Figure 8.7 gives a space breakdown of our `GCT` representation for the 8 collections, in bpn (bits per node). The breakdown has three parts: (1) the representation of the rules R of the `GCT`; (2) the representation of sequence C of the `GCT`; and (3) the extra data we store for R and C associated to samples. Obviously, we only obtain variability for the third part when we vary y and z.

In all collections, the space decreases as repetitiveness increases. The space for the R and C samplings varies significantly with parameter z, but not so much for y. This suggests that the rule sampling does not decrease the space significantly, whereas it does increase the time. Our best space/time combinations generally store the rule data for all the nonterminals, and use z to obtain space/time tradeoffs. Note also all charts are upper-limited to 2.5 bpn, which is approximately the space Sada obtains. On the real data, the situation is the same. Using reasonable values for z, the space is about 1.5 bpn for `escherichia`, the least repetitive collection. However, it decreases to less than 1 bpn on `para` and less than 0.8 on `influenza`, which is much more repetitive. On `einstein`, the most repetitive collection, this space is typically below 0.2 bpn.

8.3.4 Time Performance

Figures 8.8 to 8.12 show the time-space performance for operations *fChild* (requiring just an access to the parentheses), *tDepth* (requiring simple parenthesis operations), *nSibling*, *parent* and *tAncestor* (requiring the more complex *fwd* and *bwd* operations on the parentheses). For *tAncestor* we test with a random depth between 1 and the tree node depth.

For Sada operation times go from around one nanosecond (ns) to some microsecond (μs). The fastest, running in at most 10 ns, are *fChild*, *tDepth*, and *parent*. Instead, *nSibling* and *tAncestor* are slower, requiring 0.5–1 μs.

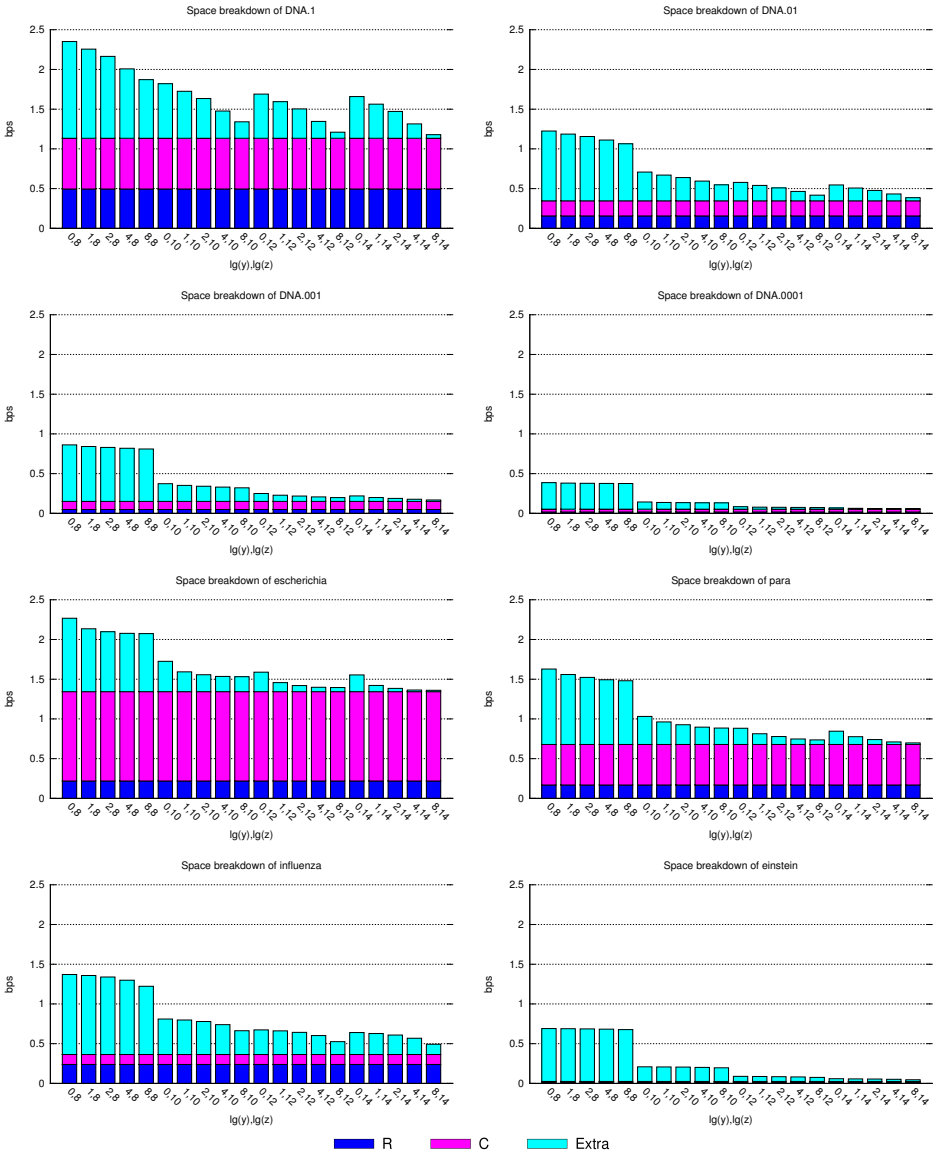


Figure 8.7: Space breakdown of our `GCT` representation for the different collections and combinations of parameters y (rule sampling) and z (sampling of C).

The GCT solves *fChild*, *tDepth* and *parent* in 5–10 μs, *tAncestor* in 10–30 μs, and *nSibling* 20–50 μs. That is 1–3 orders of magnitude slower than a plain parentheses representation.

Regarding *LCA* operation, which is the more complex, Figure 8.13 shows that the GCT uses 30–100 μs to solve it. The heaviest part of this operation is a *RMQ*, for which Sada has explicit structures, thus they solve it fast, in 4–5 μs.

We have left out other less important operations from the experiments: *root* is trivial in all implementations; *preorder* is similar to *tDepth*; *pSibling* is similar to *nSibling*; *isLeaf* costs the same as *fChild*; *ancestor* is similar to *nSibling*; *subtree* is also similar to *nSibling*.

8.4 Discussion

We have introduced the grammar compressed tree (GCT), a representation of arbitrary tree topologies that exploits repetitiveness, that is, identical subtrees, in a way that full navigation functionality is retained. In fact, any operation that can be solved on the sequence of parentheses [NS14] can also be solved on the GCT.

We have shown, in particular, that the GCT allows representing explicitly the topology of compressed suffix trees within very little space on repetitive sequence collections, using from 1.4 to less than 0.1 bits per node (bpn) both for synthetic and actual repetitive DNA sequence collections.

Two important challenges remain open:

1. Very large collections must reside on disk before they are compressed to fit in main memory. The main obstacle to handle them with our techniques is that the compression itself is not yet engineered to run on secondary memory. For example, *RePair* compression performs well only in main memory (but it can be replaced by other grammar compressors). This is an important future challenge in order to address massive repetitive text collections.

2. Lempel-Ziv compression, especially the LZ77 variant, is more powerful than grammar compression, but more difficult to manipulate [Nav12]. An interesting approach would be to apply block trees (see Chapter 7) instead of *RePair* to the tree topology, as long as we are able to perform the navigation operations.

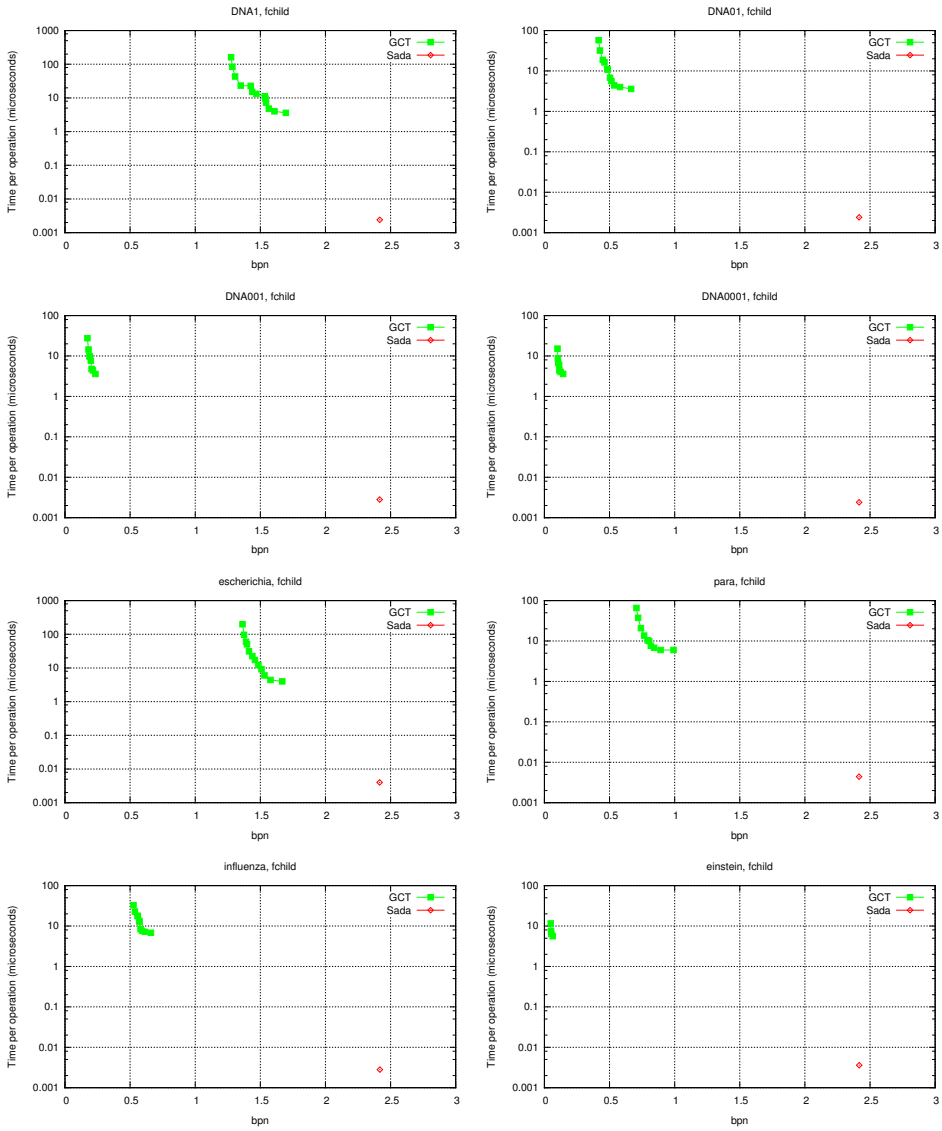


Figure 8.8: Space-time tradeoffs for operation *fChild*.

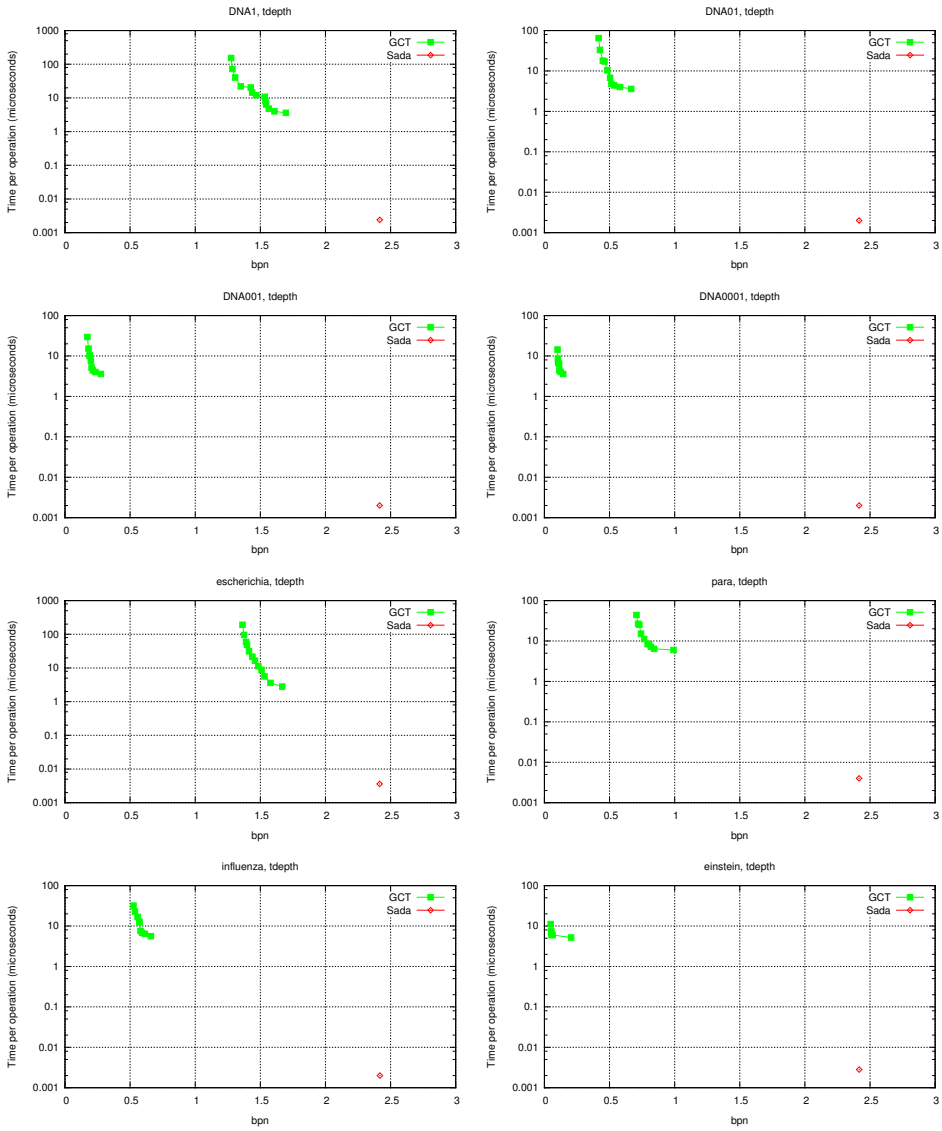


Figure 8.9: Space-time tradeoffs for operation *tDepth*.

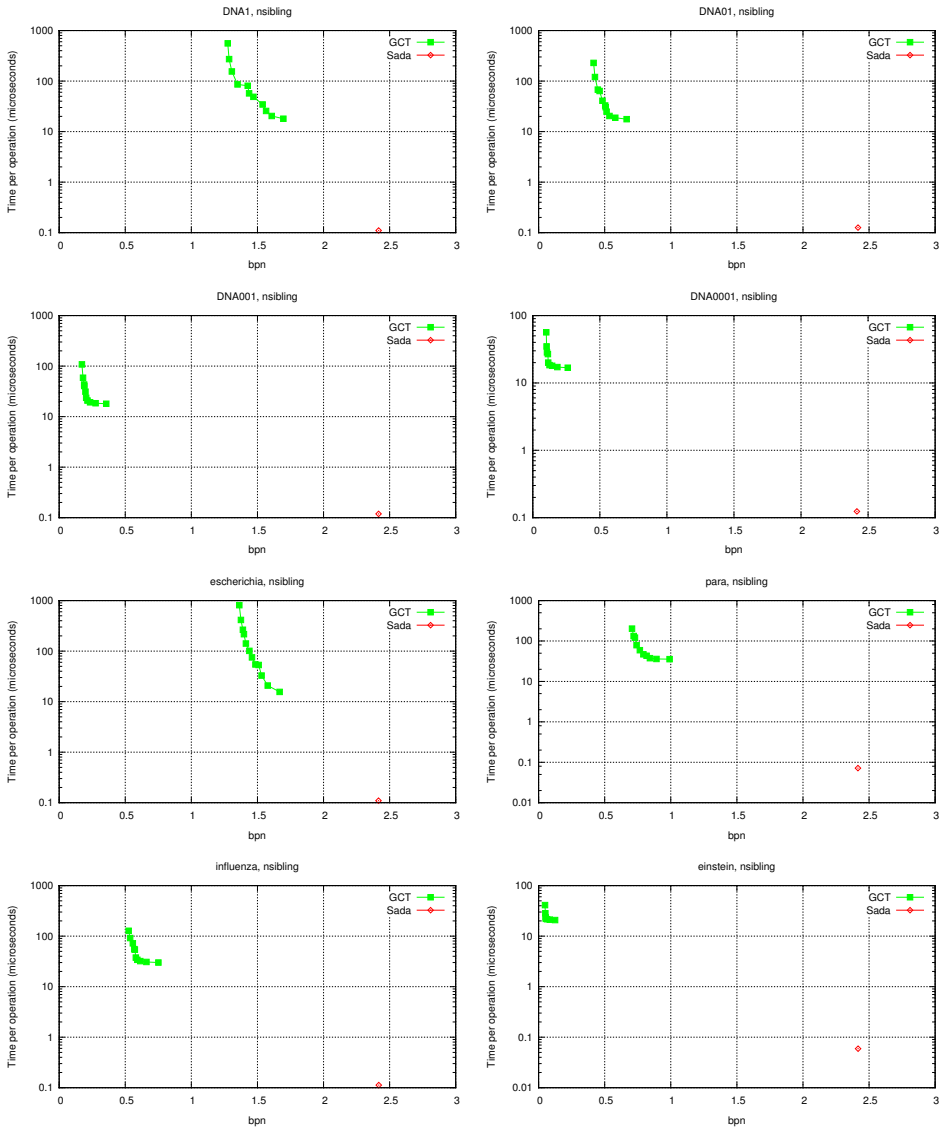


Figure 8.10: Space-time tradeoffs for operation *nSibling*.

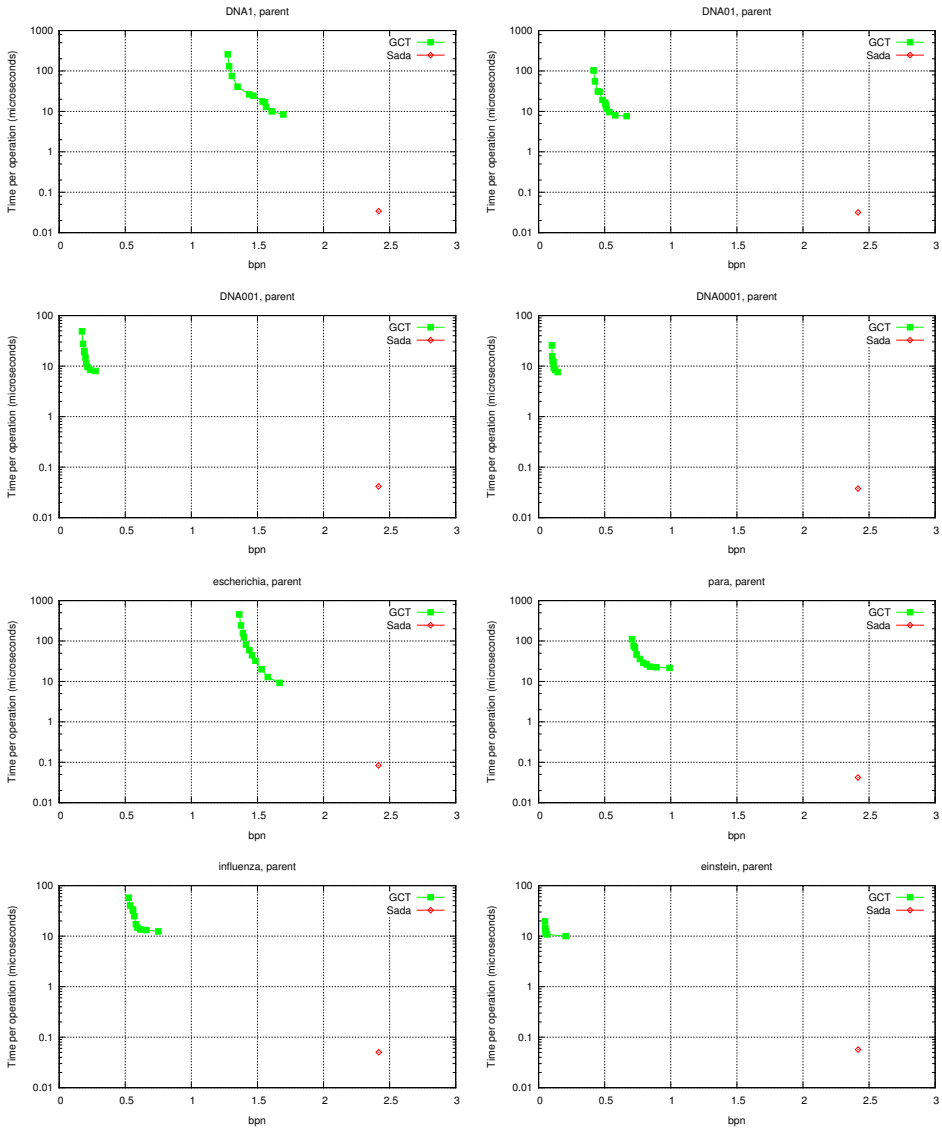


Figure 8.11: Space-time tradeoffs for operation *parent*.

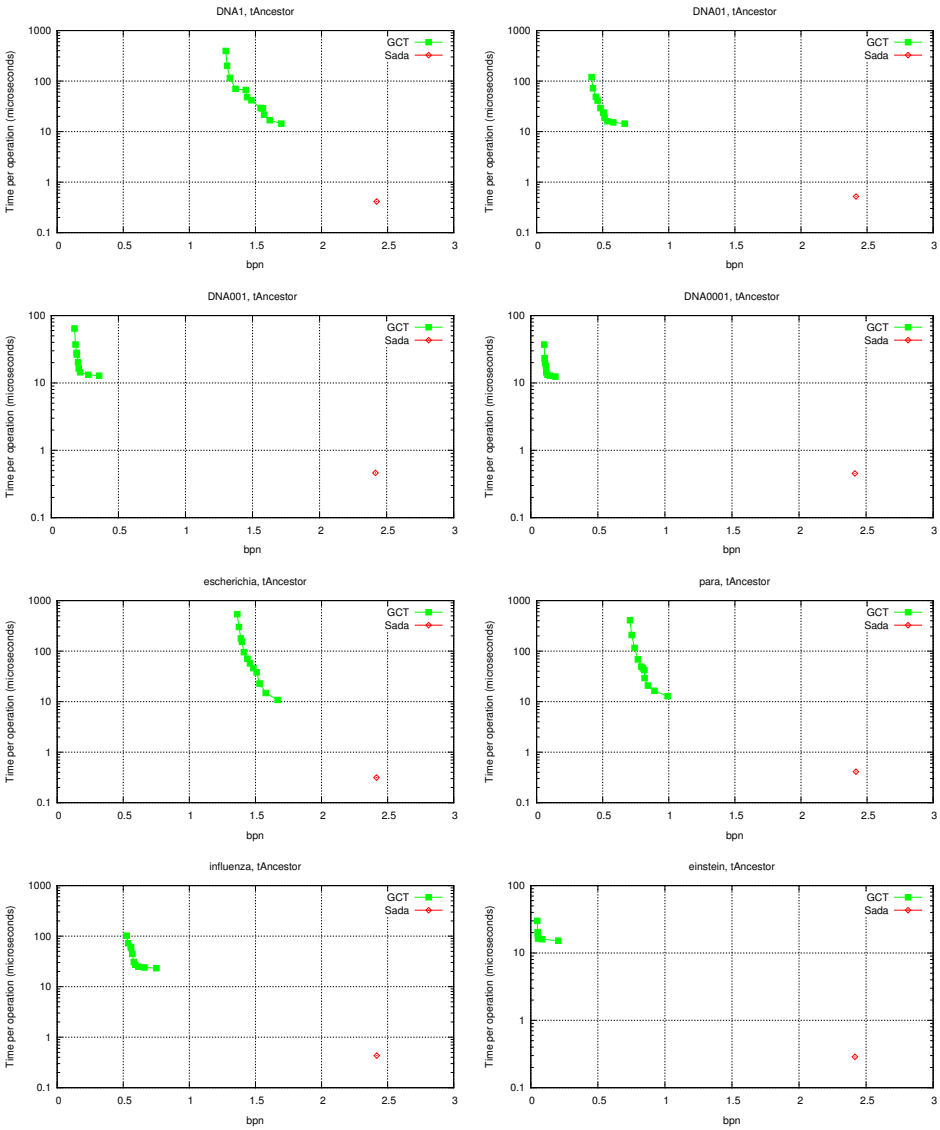


Figure 8.12: Space-time tradeoffs for operation *tAncestor*.

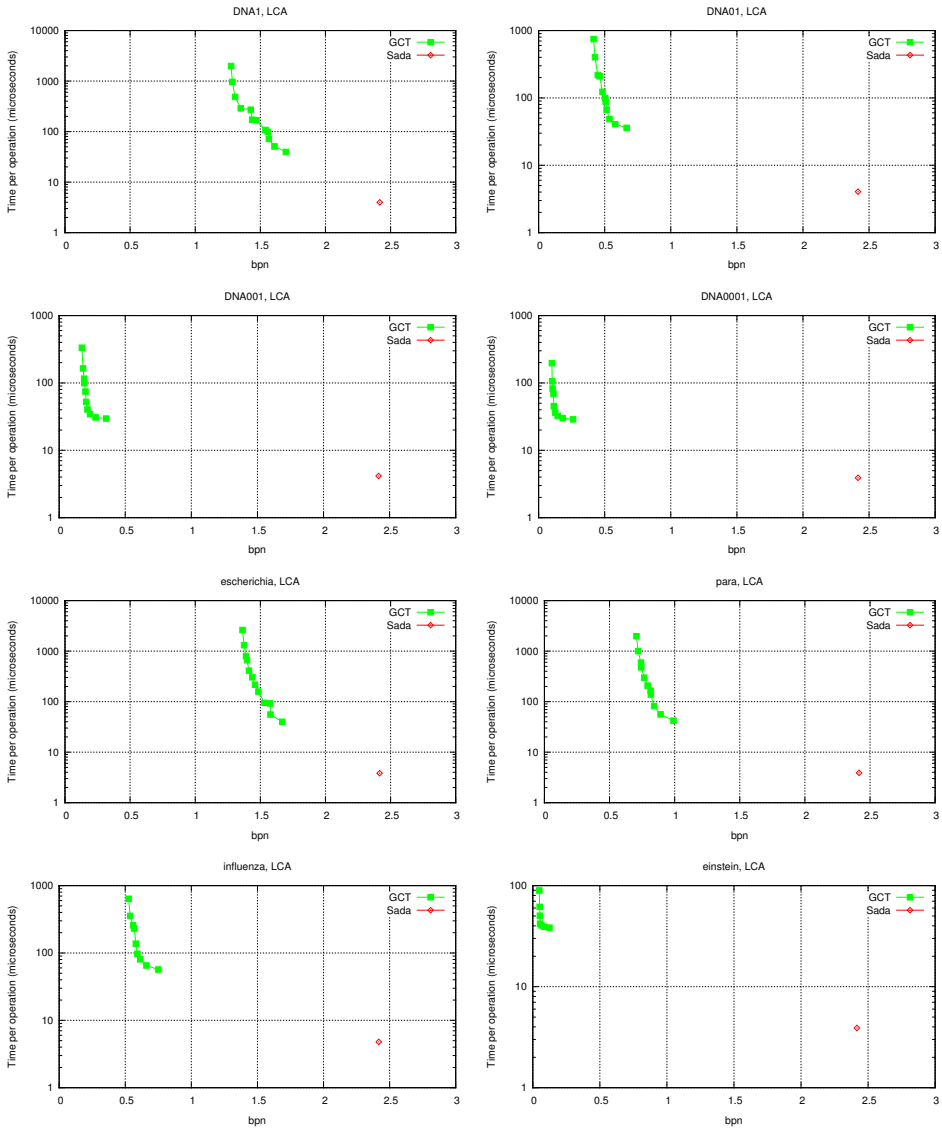


Figure 8.13: Space-time tradeoffs for operation *LCA*.

Part III

Applications

Chapter 9

Representation of Point Grids

Grids are abstract data types that arise in multiple applications, from geographic information systems (GIS), to computational geometry problems or web graphs, as well as for internal components of other data structures.

Typically, a grid is a square matrix of $n \times n$ cells that contains n points, exactly one point per row and per column (other arrangements are routinely mapped to this simplified case). Figure 9.1 shows an example.

Although a wide number operations are usually of interest when talking about grids, probably the most significant are `count` and `report`. Given a sub-rectangle $[x_1, x_2] \times [y_1, y_2]$, where $1 \leq x_1 \leq x_2 \leq n$ and $1 \leq y_1 \leq y_2 \leq n$, a `count` query in a grid tells how many points are contained in the given sub-rectangle, while `report` identifies each of them.

The first data structure to represent $n \times n$ grids and solve these two operations was the *range tree* [Ben79, Lue78, LW80, Ben80]. It takes $O(n \lg^2 n)$ bits and is able to carry out `count` queries in $O(\lg^2 n)$ time and `report` in $O(\lg^2 n + occ)$, where occ is the number of points reported. Later, Chazelle [Cha88] modified the range trees to start using bitmaps, replacing some operations in the range trees by `rank` queries in the bitmaps. The space was reduced to $O(n \lg n)$ bits, `count` query times to $O(\lg n)$ and `report` to $O((1 + occ) \lg n)$. Note these are exactly the same space and time bounds we can obtain by representing the grid with a wavelet tree-like data structure [MN06].

In this chapter we propose an algorithm to carry `count` and `report` queries in a *wavelet matrix* (see Section 4.1.2) and we compare it with the strict variant of a pointerlesss wavelet tree (see Section 4.1.1). We then experimentally evaluate both proposals, showing the wavelet matrix outperforms the wavelet tree both in real and synthetic datasets for both kinds of queries.

This chapter is organized as follows: Section 9.1 describes the algorithms for `count` and `report` queries on wavelet trees and matrices; Section 9.2 compares them experimentally; and finally Section 9.3 presents our conclusions.

9.1 Representing Grids with Wavelet Trees

As mentioned at the beginning of Chapter 4, wavelet trees and matrices are not only useful to support `access`, `rank` and `select` operations on sequences. They are also frequently used to represent point grids [Cha88, MN06, Nav12], where they can solve `count` and `report` queries efficiently. To do so, they consider a $n \times n$ matrix with only a cell marked per row and column as a sequence $S[1, n]$ over an alphabet $\sigma = n$. For instance, the grid of Figure 9.1 can be seen as a sequence $S[1, n] = 12, 7, 10, 6, 2, 11, 5, 9, 3, 8, 4, 1$. Note the alphabet size is $\sigma = n$, which means pointer-based wavelet trees will perform poorly, as the space for the tree pointers will become dominant. Similarly, zero-order compression is ineffective since S is a permutation of $[1, n]$, having thus maximal zero-order entropy. For these reasons, balanced pointerless wavelet tree-like data structures are recommendable, as Mäkinen and Navarro previously suggested [MN06].

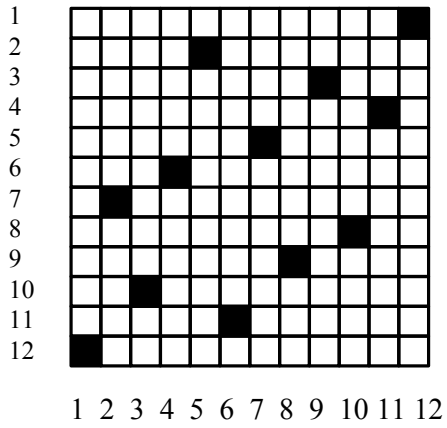


Figure 9.1: Example of a grid of 12×12 with only a marked square per row and column.

Algorithm 9 shows the pseudocode for range searches using standard wavelet trees (see Section 2.7.1). The main problem of pointer based wavelet trees when applied on grids is that the alphabet size is $\sigma = n$. This adds an $O(n \lg \sigma) = O(n \lg n)$ space

Algorithm 9 Range Search Algorithms on a Wavelet Tree: **count**(v, x_1, x_2, y_1, y_2) returns `count`(P, x_1, x_2, y_1, y_2) on the wavelet tree of sequence P rooted at v; and **report**(v, x_1, x_2, y_1, y_2) outputs all those y, where a point with coordinate $y_1 \leq y \leq y_2$ appears in $P[x_1, x_2]$. We have assigned a plain encoding to symbols taking values in $[0, \sigma)$.

count(v, x_1, x_2, y_1, y_2)
  if $x_1 > x_2 \vee [\alpha_v, \omega_v] \cap [y_1, y_2] = \emptyset$ **then**
    return 0
  else if $[\alpha_v, \omega_v] \subseteq [y_1, y_2]$ **then**
    return $x_2 - x_1 + 1$
  else
    $x_1^l \leftarrow \texttt{rank}_0(B_v, x_1 - 1)$
    $x_2^l \leftarrow \texttt{rank}_0(B_v, x_2)$
    $x_1^r \leftarrow x_1 - x_1^l + 1$
    $x_2^r \leftarrow x_2 - x_2^l$
    return count$(v_l, x_1^l, x_2^l, y_1, y_2)$
      +**count**$(v_r, x_1^r, x_2^r, y_1, y_2)$
  end if

report(v, x_1, x_2, y_1, y_2)
  if $x_1 > x_2 \vee [\alpha_v, \omega_v] \cap [y_1, y_2] = \emptyset$ **then**
    return
  else if $\omega_v - \alpha_v = 1$ **then**
    output α_v
  else
    $x_1^l \leftarrow \texttt{rank}_0(B_v, x_1 - 1) - l + 1$
    $x_2^l \leftarrow \texttt{rank}_0(B_v, x_2) - l$
    $x_1^r \leftarrow x_1 - x_1^l + 1$
    $x_2^r \leftarrow x_2 - x_2^l$
    report$(v_r, x_1^l, x_2^l, y_1, y_2)$
    report$(v_l, x_1^r, x_2^r, y_1, y_2)$
  end if

term just to store the tree pointers, becoming this approach not recommendable in practice.

Instead, Algorithm 10 shows the pseudocode adapted to pointerless wavelet trees, which remove that space overhead. The time complexities can be shown to be $O(\lg n)$ for `count` and $O(k \lg(n/k))$ if `report` outputs k points. These time bounds are matched by both Algorithms 9, and 10. However, in practical terms Algorithm 10 requires twice the number of `rank` operations than Algorithm 9.

Focusing on wavelet matrices, range searches for rectangles $[x_1, x_2] \times [y_1, y_2]$ essentially require that we are able to track the points x_1 and x_2 downwards in the tree. Thus, the same wavelet matrix mechanism for `rank` can be used (see Algorithm 7). Since we are only interested in the value $x_2 - x_1$ at the traversed nodes, we do not need to keep track of p, even in the strict variant (the extended variant requires too much space in this scenario). As a result, we need the same number of `rank` operations as in a pointer-based representation (see Algorithm 9), and we get rid of the two extra `rank` operations required by the pointerless wavelet tree. That is, wavelet matrices get the best of pointer-based and pointerless wavelet tree-based approaches. Algorithm 11 gives the pseudocode of `count` and `report` on wavelet matrices.

Algorithm 10 Range search algorithms on pointerless wavelet trees: **count**$(0, x_1, x_2, y_1, y_2, 0, n)$ returns $\texttt{count}(P, x_1, x_2, y_1, y_2)$ on the wavelet tree of sequence P; and **report**$(0, x_1, x_2, y_1, y_2, 0, n)$ outputs all those y, where a point with coordinate $y_1 \leq y \leq y_2$ appears in $P[x_1, x_2]$. For simplicity we have omitted the computation of $[\alpha_v, \omega_v)$. We have assigned a plain encoding to symbols taking values in $[0, \sigma)$.

count$(\ell, x_1, x_2, y_1, y_2, p, e)$

if $x_1 > x_2 \vee [\alpha_v, \omega_v) \cap [y_1, y_2] = \emptyset$ **then**
  return 0
else if $[\alpha_v, \omega_v) \subseteq [y_1, y_2]$ **then**
  return $x_2 - x_1 + 1$
else
  $l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, p)$
  $r \leftarrow \texttt{rank}_0(\tilde{B}_\ell, e)$
  $x_1^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_1 - 1) - l + 1$
  $x_2^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_2) - l$
  $x_1^r \leftarrow x_1 - x_1^l + 1$
  $x_2^r \leftarrow x_2 - x_2^l$
  return count$(\ell{+}1, x_1^l, x_2^l, y_1, y_2, p, p{+}r{-}l)$
    $+$**count**$(\ell{+}1, x_1^r, x_2^r, y_1, y_2, p{+}r{-}l, e)$
end if

report$(v, x_1, x_2, y_1, y_2, p, e)$

if $x_1 > x_2 \vee [\alpha_v, \omega_v) \cap [y_1, y_2] = \emptyset$ **then**
  return
else if $\omega_v - \alpha_v = 1$ **then**
  output α_v
else
  $l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, p)$
  $r \leftarrow \texttt{rank}_0(\tilde{B}_\ell, e)$
  $x_1^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_1 - 1) - l + 1$
  $x_2^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_2) - l$
  $x_1^r \leftarrow x_1 - x_1^l + 1$
  $x_2^r \leftarrow x_2 - x_2^l$
  report$(\ell{+}1, x_1^l, x_2^l, y_1, y_2, p, p{+}r{-}l)$
  report$(\ell{+}1, x_1^r, x_2^r, y_1, y_2, p{+}r{-}l, e)$
end if

9.2 Experimental Results

Our implementations build over the wavelet tree implementations of Libcds [Cla], a library implementing several space-efficient data structures. For each wavelet tree/matrix variant we present two versions, `CM` and `RRR` (recall Section 2.6).

The variants compared are the following:

- `WTNP`: the (strict) pointerless wavelet tree (Section 4.1.1);
- `WM`: the (strict) wavelet matrix (Section 4.1.2);

Each data structure is appended with the name of the bitmap implementation it uses. For example, we call `WTNP.RRR` the pointerless wavelet tree with all bitmaps represented with `RRR`.

Note we cannot use Huffman compression, because the order of the symbols is not maintained at the leaves. Alphabet Partitioning techniques (see Section 2.7.4) also shuffles the alphabet, so it cannot be used in this scenario. Extended variants are not a good option either, because in this case it holds $\sigma = n$. Thus we test only the strict variants of `WTNP` and `WM`.

In order to evaluate the range search performance over discrete grids, we use the following three datasets formed by synthetic and real collections of MBRs (Minimum

Algorithm 11 Range search algorithms on the wavelet matrix: **count**$(0, x_1, x_2, y_1, y_2)$ returns $\texttt{count}(P, x_1, x_2, y_1, y_2)$ on the wavelet tree of sequence P; and **report**$(0, x_1, x_2, y_1, y_2)$ outputs all those y, where a point with coordinate $y_1 \leq y \leq y_2$ appears in $P[x_1, x_2]$. For simplicity we have omitted the computation of $[\alpha_v, \omega_v)$. We have assigned a plain encoding to symbols taking values in $[0, \sigma)$.

count$(\ell, x_1, x_2, y_1, y_2)$

if $x_1 > x_2 \vee [\alpha_v, \omega_v) \cap [y_1, y_2] = \emptyset$ **then**
  return 0
else if $[\alpha_v, \omega_v) \subseteq [y_1, y_2]$ **then**
  return $x_2 - x_1 + 1$
else
  $x_1^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_1 - 1) + 1$
  $x_2^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_2)$
  $x_1^r \leftarrow x_1 - x_1^l + 1$
  $x_2^r \leftarrow x_2 - x_2^l$
  return count$(\ell{+}1, x_1^l, x_2^l, y_1, y_2)$
    $+$**count**$(\ell{+}1, x_1^r, x_2^r, y_1, y_2)$
end if

report(v, x_1, x_2, y_1, y_2)

if $x_1 > x_2 \vee [\alpha_v, \omega_v) \cap [y_1, y_2] = \emptyset$ **then**
  return
else if $\omega_v - \alpha_v = 1$ **then**
  output α_v
else
  $x_1^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_1 - 1) + 1$
  $x_2^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_2)$
  $x_1^r \leftarrow x_1 - x_1^l + 1$
  $x_2^r \leftarrow x_2 - x_2^l$
  report$(\ell{+}1, x_1^l, x_2^l, y_1, y_2)$
  report$(\ell{+}1, x_1^r, x_2^r, y_1, y_2)$
end if

Bounding Rectangles of objects). We insert the two opposite corners of each MBR as points in our dataset.

- `Zipf`: A synthetic collection of 1,000,000 MBRs with a Zipfian distribution (world size = 1,000 × 1,000, $\rho = 1$).[1]
- `Gauss`: A synthetic collection of contains 1,000,000 MBRs with a Gaussian distribution (world size = 1,000 × 1,000, $\mu = 500, \sigma = 200$).[1]
- `Tiger`: A real collection of 2,249,727 MBRs from California roads, available at the U.S. Census Bureau.[2]

To measure the performance on point grids, for synthetic collections we generate sets of queries covering 0.001%, 0.01%, 0.1%, and 1% of the grid area. The sets contain 1,000 queries, each with a ratio between both axes varying uniformly at random between 0.25 and 2.25. For the real data set `Tiger`, we use as queries the following four collections (also available for downloading at the Web site of `Tiger`): `Block` (groups of buildings), `BG` (block groups), `SD` (elementary, secondary, and unified school districts), and `COUSUB` (country subdivisions).

The machine used is an Intel(R) Xeon(R) E5620 running at 2.40GHz with 96GB of RAM memory. The operating system is GNU/Linux, Ubuntu 10.04, with kernel

[1] `http://lbd.udc.es/research/serangequerying`
[2] `http://www.census.gov/geo/www/tiger`

2.6.32-33-server.x86_64. All our implementations use a single thread and are coded in `C++`. The compiler is `gcc` version 4.4.3, with `-O9` optimization.

Figures 9.2 and 9.3 show the performance of `WTNP` and `WM` for `count` and `report` queries, respectively. It turns out that, in the first level of each wavelet tree, the number of zeros and ones is highly unbalanced when the grid size is far from the next power of 2. This makes the entropy of the first bitmap rather low, whereas the other bitmaps are more balanced. On the other hand, the range search algorithms spend just a few `rank` operations on the first bitmap. To take advantage of this feature, we compress the bitmap of the first level of both data structures, `WTNP` and `WM`, with `RRR` and with a sampling of 32. The rest of bitmaps are represented using `CM` with sampling rates of 32, 64, and 128.

In both figures 9.2 and 9.3 we append to the name of the data structure the name of the query set. This takes values in $\{Q0001, Q001, Q01, Q1\}$ in case of synthetic collections. In case of the real collection `Tiger`, it takes values in {BLock, BG, SD, COUSUB}.

The results for the counting queries show that the time worsens as the queries are less selective. The wavelet matrix is always faster than the pointerless wavelet tree, while using the same space. The difference in time is proportional to the cost for each selectivity, but additive with respect to the sampling. For example, it becomes about 25% faster when using the most space. We note in passing that the space is basically 21 bps for the synthetic spaces and 23 bps for the `Tiger` dataset, which is essentially $\lg \sigma = \lg n$.

In the case of reporting queries, we show the time per reported item, which decreases as the query is less selective. Once again the wavelet matrix is faster than the pointerless wavelet tree, albeit this time by a smaller margin: usually below 10%.

9.3 Discussion

In this chapter we have shown how to represent grids of points with the wavelet matrix. We compared its performance for `count` and `report` queries with pointerless wavelet trees, resulting in improved search performance, systematically matching the same space. Therefore, we can consider the wavelet matrix a superior approach to wavelet trees to represent grids of points and to carry out orthogonal range queries.

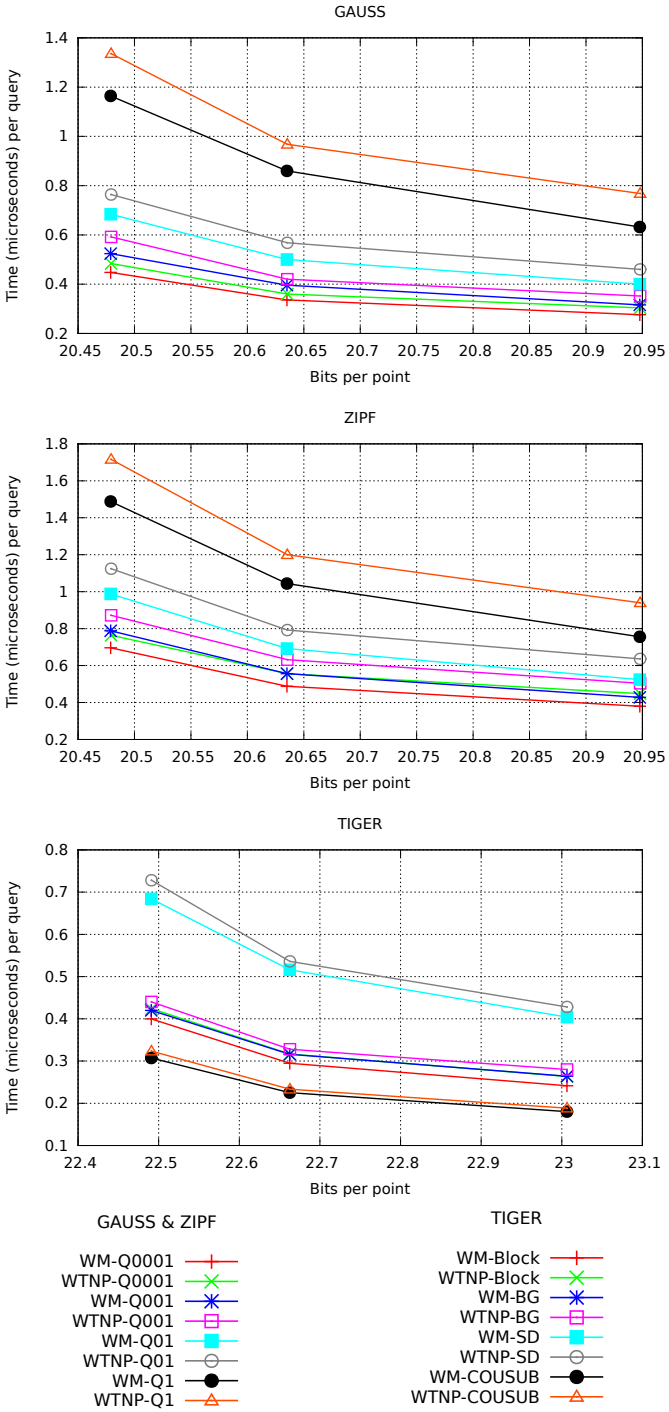


Figure 9.2: Running time per `count` query over the three datasets.

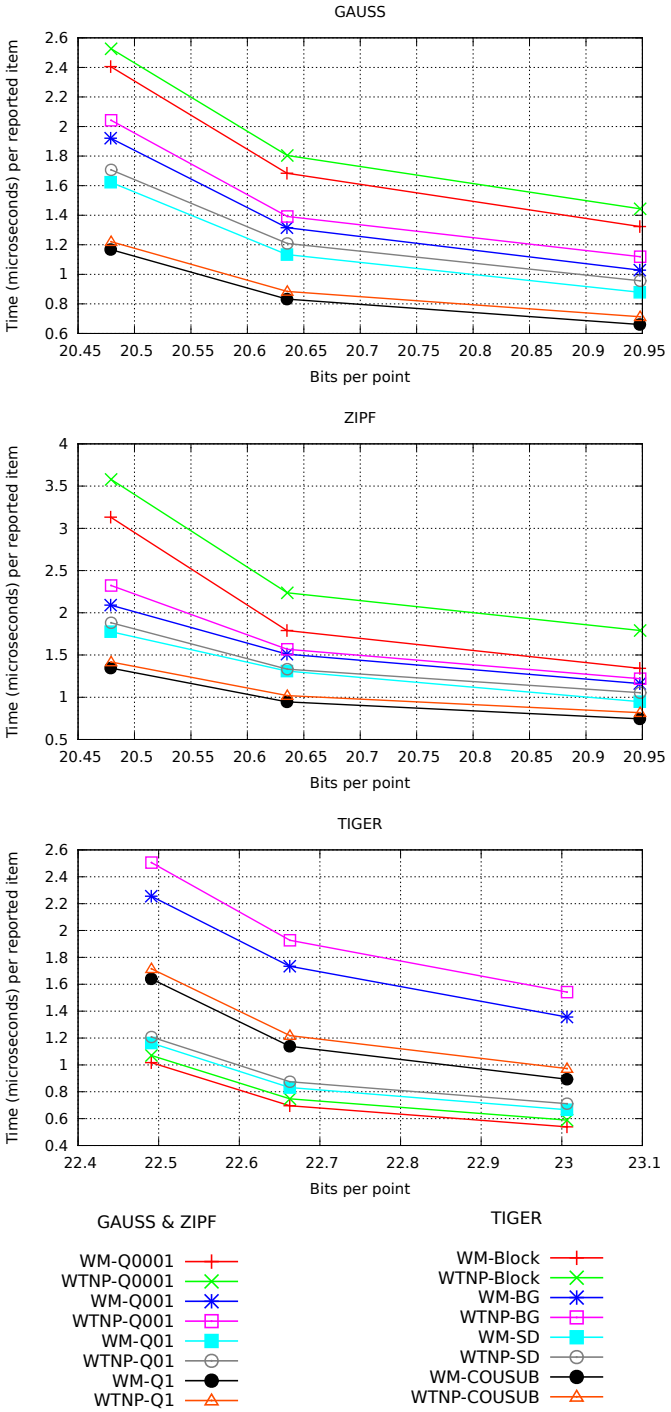


Figure 9.3: Running time of `report` query over the three datasets.

Chapter 10

Inverted Indexes

Along the past few decades, Inverted Indexes have been the core data structure to speed up searches on natural language text collections [BYRN11, WMB99]. They are behind most search engines and typically provide two kinds of functionalities: *pattern-search* and *document retrieval*. Regarding the first, they permit to search the documents for text patterns, obtaining the positions of their occurrences quite efficiently. With regard to *document retrieval*, they are highly demanded and useful when using boolean and ranked retrieval models, since they permit to obtain all the documents where a term appears.

A *positional inverted index* is a data structure that stores the positions where each word occurs at each document in increasing order. Instead, a *non-positional* or *document-oriented inverted index* only stores the list of documents where each word appears. Typical implementations of these indexes differentially encode the lists of each word, and compress them using some encoding that favors small numbers [BYRN11, WMB99]. Compression is obtained because longer lists have smaller gaps. Using proper codes, the size of the positional index can reach the zero-order entropy of the input text T [NM07], and the non-positional is usually much smaller. When both indexes are stored, each non-positional entry also points to the first corresponding positional entry. The text can be stored in some compressed form so that one can extract arbitrary snippets, for example using an `access`-capable sequence representation that reaches zero-order entropy as well (see Section 2.7 and Chapter 4). In this case, the space of the text plus the indexes is at least $2nH_0(T)$ bits.

Inverted indexes are used to list the positions of a word in a document, the documents where a word appears, the documents where two words appear simultaneously, the positions where a pair of words appear as a phrase, etc. Using little space in inverted indexes has always been of interest [WMB99], and the recent trend is to maintain the index in the main memory of the computer (or of a cluster of computers) [SWYZ02, SC07, CM07, ST07].

Algorithm 12 **ExtractPosting**(t, d) reports the positions of term t within document d. **ExtractDocs**(t) reports the documents that contain t.

ExtractPosting(T, t, d)	**ExtractDocs**(T, t)
$s \leftarrow \texttt{select}_\$(T, d)$	$j \leftarrow 1$
$e \leftarrow \texttt{select}_\$(T, d+1)$	$n_t \leftarrow \texttt{rank}_t(T, \|T\|)$
$r \leftarrow \texttt{rank}_t(T, s) + 1$	$res \leftarrow \{\}$
$res \leftarrow \{\}$	**while** $j \leq n_t$ **do**
$next \leftarrow \texttt{select}_t(T, r)$	$p \leftarrow \texttt{select}_t(T, j)$
while $next < e$ **do**	$r \leftarrow \texttt{rank}_\$(T, p)$
$res \leftarrow res : (next - s)$	$res \leftarrow res : r$
$r \leftarrow r + 1$	$p \leftarrow \texttt{select}_\$(T, r+1)$
$next \leftarrow \texttt{select}_t(T, r)$	$j \leftarrow \texttt{rank}_t(T, r) + 1$
end while	**end while**
return res	**return** res

10.1 Inverted Indexes with `rsa` Data Structures

Let us regard a natural language text collection as a set of documents $D_1, D_2, \ldots, D_d$. Let us call $T[1, n] = \$D_1\$D_2\$ \ldots D_d\$$ the concatenation of the documents, where each position of T is a word and we use a special separator word \$ preceding and following each document. The alphabet Σ of T is large, as it consists of the distinct words in the collection.

There has been some research around the idea of just representing the text collection as a sequence and using `rsa` operations to simulate the functionalities of inverted indexes, thus using basically $nH_0(T)$ bits [BCG+14, AGO10, BFLN12]. Algorithms 12 and 13 detail some of the most common operations. We dub this solution `RSAII`.

In some applications, the text collection is versioned. For instance, if we index Wikipedia, each article has many versions. The result will be a highly repetitive dataset where most articles are very similar from one snapshot to the next. Inverted indexes for versioned collections have been studied for a while [AF92, BEF+06, HYS09, HZS10, CFMPN11], exploiting the redundancies that repetitions in the collection induce in the inverted indexes.

In this chapter we present an experimental evaluation in which we use our data structures for large alphabets and highly repetitive inputs to implement an `RSAII` for the text T.

10.2 Experimental Results

In order to carry out the comparison between different inverted index implementations, we use collection `fiwiki` (see Section 5.6), which is precisely the concatenation of versioned documents. We add the separators \$ (which does not alter the numbers

Algorithm 13 ReportDocsWithTerms(T, t_1, t_2) reports the list of documents that contain both t_1 and t_2.

ReportDocsWithTerms(t_1, t_2)
  $n_1 \leftarrow \texttt{rank}_{t_1}(T, |T|)$
  $n_2 \leftarrow \texttt{rank}_{t_2}(T, |T|)$
  $res \leftarrow \{\}$
  $d_1 \leftarrow Next(T, t_1, n_1, 1)$
  $d_2 \leftarrow Next(T, t_2, n_2, 1)$
  while $d_1 \neq -1$ **and** $d_2 \neq -1$ **do**
    if $d_1 = d_2$ **then**
      $res \leftarrow res : d_1$
      $d_1 \leftarrow Next(T, t_1, n_1, d_1 + 1)$
      $d_2 \leftarrow Next(T, t_2, n_2, d_2 + 1)$
    else if $d_1 < d_2$ **then**
      $d_1 \leftarrow Next(T, t_1, n_1, d_2)$
    else
      $d_2 \leftarrow Next(T, t_2, n_2, d_1)$
    end if
  end while
  return res

Next(T, t, n_t, d)
  $p \leftarrow \texttt{select}_{\$}(T, d)$
  $r \leftarrow \texttt{rank}_t(T, p)$
  if $r = n_t$ **then**
    return -1
  end if
  $next \leftarrow \texttt{select}_t(T, r + 1)$
  return $\texttt{rank}_{\$}(T, next)$

in Table 5.1 negligibly), and compare the following solutions:

- `RSAII` implemented with:
  - `AP`, the Alphabet Partitioning technique described in Section 2.7.4;
  - `APRep.RP.WMRP`, the solution for large alphabets and highly repetitive sequences described in Chapter 6;
  - `WMH.RP`, a variation of the wavelet tree for highly repetitive inputs described in Section 5.5 which uses the compressed wavelet matrix of Chapter 4 instead of a pointerless wavelet tree.
  - `WMH.RRR` the compressed wavelet matrix (Chapter 4) with bitmaps compressed with `RRR` (see Section 2.6);
- `II-VByte`, an inverted index with the list gaps encoded using VByte (see Chapter 2.5.2).
- `II-RePair`, an inverted index with the list gaps compressed with *RePair* [CFMPN11] to exploit the repetitiveness of the collection.

The implementation we use for these indexes [CFMPN11] includes a non-positional and a positional variant. However, instead of charging the space of both variants and the additional pointers, we only show the space of the positional variant for each case, independently of the needs of the operation. That is, we are

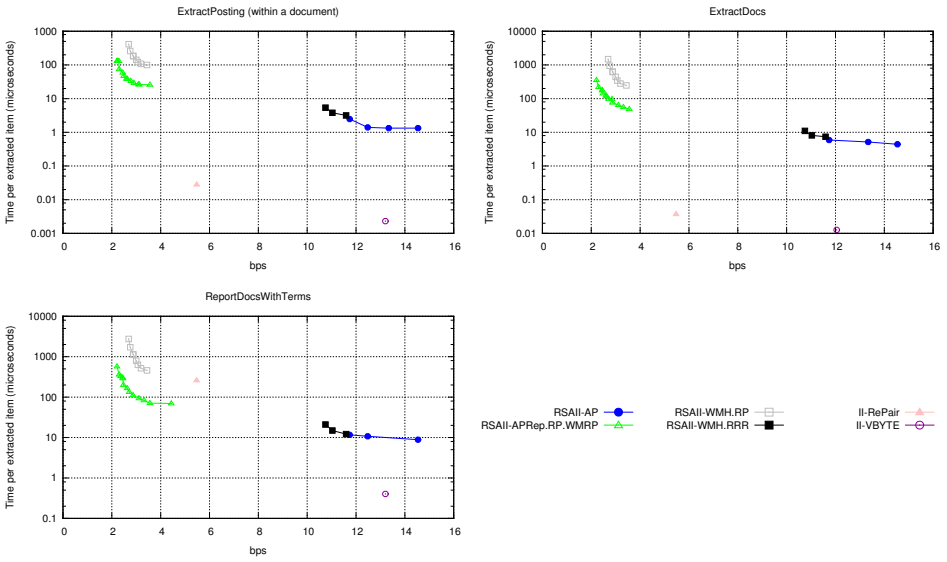


Figure 10.1: Space-time tradeoffs for inverted index operations (time in logscale).

charging to `II-VByte` and `II-RePair` less space that the strictly necessary (the difference is not much anyway).

We used an Intel(R) Xeon(R) E5620 at 2.40GHz with 96GB of RAM memory, running GNU/Linux, Ubuntu 10.04, with kernel 2.6.32-33-server.x86_64. All our implementations use a single thread and are coded in `C++`. The compiler is `g++` version 4.7, with `-O9` optimization. We implemented our solutions inside LIBCDS [Cla] and use Navarro's implementation of RePair (`www.dcc.uchile.cl/gnavarro/software/repair.tgz`).

For the queries we extract documents and term identifiers at random from the text sequence. We ran 1,000 queries of each type, showing the time required per output item. Figure 10.1 shows the results.

First of all we highlight that `RSAII-APRep.RP.WMRP` outperforms `RSAII-APRep.BT.WMBT` in terms of space but the second is slightly faster in *ExtractPosting* and *ExtractDocs*. With regard to *ReportDocsWithTerms*, both solutions almost match in time, but again `RSAII-APRep.RP.WMRP` is more compact.

Comparing `RSAII-APRep.`$*$ with the `RSAII-WM.RP`, the second is outperformed both in terms of space and time in all the operations. Actually, `RSAII-APRep.`$*$ are almost an order of magnitude faster than `RSAII-WM.RP` when using the same space. Compared with statistically compressed `RSAIIs`, `RSAII-AP` and `RSAII-WM`, our solution is up to 6 times smaller but an order of magnitude slower.

With regard to classical implementations of inverted indexes, our `RSAII-APRep.*` are again up to 6 times smaller than `II-VByte`, which also obtains zero-order compression. In terms of time, `II-VByte` clearly outperforms `RSAII-APRep.*` by several orders of magnitude (around 3). Compared with `II-RePair`, the `RSAII-APRep.*` are never more than 3 times smaller but they are several orders of magnitude slower, except for *ReportDocsWithTerms*, in which `RSAII-APRep.*` obtains better performance.

10.3 Discussion

In general, simulating inverted indexes with `rsa` data structures seems a good idea if the space is the critical aspect of the application or if the search performance is not a limitation. Besides, if we are reaching for richer functionalities, the fact of simulating the inverted index operations with `rsa` data structures opens a window to more complex operations [GNP12]. However, if the search performance is critical, classical approaches should prevail, as the experiments have shown.

Chapter 11

Self-Indexes on Highly Repetitive Sequences

Given a sequence $T[1,n]$ with alphabet $\Sigma = [1,\sigma]$, in Section 2.10 we described the concept of *self-index* as a data structure that permits to represent T and to solve a wide number of operations without actually accessing T.

Although there exist plenty of compressed data structures for this purpose, the most well known may be classified into two big families: CSAs (Compressed Suffix Arrays) [GV06, Sad03] and FM-Indexes (Full-text in Minute space-Indexes) [FM05]. Both support essentially the same functionalities but obtain different search and space performances, although typically space-bounded by $\mathcal{H}_0(T)$ or $\mathcal{H}_k(T)$, $k < \log_\sigma n$ [MN08]. Note this is a major improvement with regard to suffix arrays, which need $O(n \lg n)$ bits [MM93] to just solve the same operations.

Even though this statistical space-bounds may suffice for most applications, there exists scenarios in which that performance is no enough. For instance, DNA analysis is a trend nowadays, and being able to carry out pattern matching queries on these kind of datasets is a first order need. However, and as we previously mentioned, DNA shows a highly repetitive structure that statistical compressor are not able to exploit. This result in unnecessarily large indexes, inducing higher investments on hardware and poor performance if the data structures do not fit in main memory. Therefore, in order to obtain better *self-indexes* on highly repetitive scenarios, a wide number space efficient solutions were proposed over the last decade [MNSV10, CN10b, CN12, GGK$^+$12, MNKS13, Nav04, TTS14, KN13].

Probably, one of the most successful solutions are those based on combining run-length encodings with CSAs and FM-Indexes (RLCSA or RLFM-Indexes) [MNSV10]. They perform very well in practice since they are able to exploit some regularities that appear in the internal components of CSAs and FM-Indexes when faced to highly repetitive scenarios.

In this chapter we propose a new practical grammar compressed *self-index* on highly repetitive scenarios. The idea is to use the sequence representations on highly repetitive scenarios proposed along this thesis (recall Chapters 6 and 7) to represent the internal structures of the FM-Index, permitting us to reduce the space with regard to current implementations at the cost of increasing the time. This results in a new proposal that may become of interest depending on the scenario and the space/time requirements of the application. We dubbed our proposal `GFMI` or `BTFMI`, from Grammar compressed and Block-Tree FM-Index respectively.

The chapter is organized as follows: Section 11.1 describes CSAs and FM-Indexes; Section 11.2 shows several *self-indexes* adapted to highly repetitive datasets; Section 11.3 presents our proposals for *self-indexes* on highly repetitive scenarios; Section 11.4 provides an experimental evaluation of our techniques; and finally Section 11.5 gives our conclusions.

11.1 Statistically-bounded Self-Indexes

Let $T[1,n]$ be a text (or the concatenation of the texts in a collection) over alphabet $\Sigma = [1,\sigma]$. Then, the *suffix array* [MM93] of T, $A[1,n]$, is a permutation such that $A[i]$ tells the position in T of the ith lexicographically smallest suffix of T. The *inverse suffix array* of T, $A^{-1}[1,n]$, is also a permutation such that $A^{-1}[A[i]] = i$. These two arrays are fundamental for CSAs and FM-Indexes in order to be fully-functional.

In the rest of this section, we will explain in detail how these data structures can be compressed, as well as how CSAs and FM-Indexes use them, along with other internal data structures, to carry out the expected functionalities.

11.1.1 Compressed Suffix Arrays

A compact representation of a *suffix array* requires $O(n \lg n)$ bits [MM93]. However, a compressed representation achieves $O(n \lg \sigma)$ bits and, in most cases, close to the empirical entropy of T [Man01]. Given a pattern of length m, most CSAs achieve times of the form $O(m)$ to $O(m \lg n)$ for operation $\texttt{count}(p)$, $O(\text{polylog}\, n)$ to access $A[i]$ or $A^{-1}[i]$, and at most $O((l-r)\lg\sigma + \text{polylog}\, n)$ to $\texttt{extract}(T,l,r)$. Operation *locate* can be carried out in $O(\text{polylog}\, n)$ time per reported occurrence.

The main component of a CSA, is the function or permutation Ψ, that is used, along with other data structures, to simulate A and A^{-1}. Ψ is defined as $\Psi[i] = A^{-1}[A[i]+1]$ or $\Psi[i] = A^{-1}[1]$ if $A[i] = n$. That is, given a position i in the suffix array A, $\Psi[i]$ tells the position in A where the suffix starting at i continues if we remove its first symbol. Or in other words, Ψ permits to virtually move forward in the text without actually accessing the text.

To solve typical operations, additionally to Ψ, A, and A^{-1}, we need to store a vector $C[1,\sigma+1]$ such that $C[a]$ stores the number of symbols lexicographically smaller than a in T, and $C[\sigma+1] = n$. Additionally, we need an array $D[1,\sigma]$

that contains the symbols in Σ (including the terminator \$) sorted in increasing lexicographical order. Note C can also be represented as a bitmap $W[1,n]$ such that $W[i] = 1$ iff $i = C[a] + 1$ for some $a \in \Sigma$.

How to efficiently represent Ψ, A, and A^{-1} is explained next.

11.1.1.1 Compressing Ψ

Regarding Ψ as a permutation, it is known that it has at most σ runs of increasing values [MN05]. This is reasonable because in the suffix array suffixes are sorted in lexicographical increasing order, which means that if we take a region of suffixes that start with the same symbol and we remove it, those suffixes are still sorted. Therefore, for a region of suffixes starting with the same symbol, Ψ must contain increasing values, and, as we have exactly σ regions that start with the same symbol, we have at most σ increasing runs. Having this properties into account, several works related with Ψ compression have been published [Sad03, GGV03, GV06, FBN$^+$12, BN13]. Some of them consist of sampling Ψ at regular intervals and use different compression techniques to differentially encode those values of Ψ between samples [Sad03].

Regarding the functionality, Algorithm 14 shows how to find the suffix array interval associated with a pattern $P[1,m]$ by using Ψ and C. It starts by finding the suffix array interval $[sp, ep]$ that covers the last pattern symbol $P[m]$. At the beginning of the ith iteration of the loop, we have in $[sp, ep]$ the suffix array interval associated with suffix $P[i+1, m]$ (i starts at $m-1$). In the loop's body, we compute the largest interval of $[C[P[i]]+1, C[P[i]+1]]$ that contains values of Ψ that point to the current interval $[sp, ep]$. That is, we are searching for the interval that contains suffixes $P[i, m]$. The search inside $[C[P[i]] + 1, C[P[i] + 1]]$ can be done in $\lg n$ time since those values are sorted. This loop iterates until we exhaust the pattern, or until we cannot find those j values described in the Algorithm 14, which means the suffix $P[i, m]$ does not exist, and hence, that $P[1, m]$ does not occur in the text.

Algorithm 15 shows how to extract the string $T[SA[i], SA[i] + m]$ given i, Ψ, W (or C), and D. The key point is to report the first symbol of a suffix ($T[SA[i]]$). This can be done by accessing the bitmap W, and then using the reported index to access the vector D. After that, we should virtually move forward in the text to report $T[SA[i] + 1]$, which can be done by applying Ψ. This process is repeated m times, or $n - SA[i]$ if $|T[SA[i], n]| < m$.

11.1.1.2 Compressing the Suffix Array

Instead of storing the whole $A[1,n]$ array, which would require $O(n \lg n)$ bits, A is sampled at regular intervals in the text, and only these samples are stored in a vector $A_s[1, \lceil n/\delta \rceil]$, being $\delta > 0$ the sampling period (typically $\delta = O(\text{polylog}\, n)$). Concretely, we will store $A[i]$ if $A[i] \bmod \delta = 0$ or $A[i] = n$. As A is sampled at regular intervals of the text and not of the suffix array, we need to keep track of those marked positions, otherwise we would not know which ones were sampled. To

Algorithm 14 Searching for a pattern $P[1,m]$ in a suffix array using the Ψ function.

SearchInterval$(P[1,m], \Psi, C)$
  $sp \leftarrow C[P[m]] + 1$
  $ep \leftarrow C[P[m]+1]$
  for $i = m-1$ **downto** 1 **do**
    if $\exists\ j \in [C[P[i]]+1, C[P[i]+1]]$ s.t. $\Psi[j] \in [sp, ep]$ **then**
      $sp, ep \leftarrow min(j), max(j)$
    else
      return $[0,0]$
    end if
  end for
  return $[sp, ep]$

Algorithm 15 Returns $T[SA[i], SA[i]+m]$ given i, Ψ, W, and D.

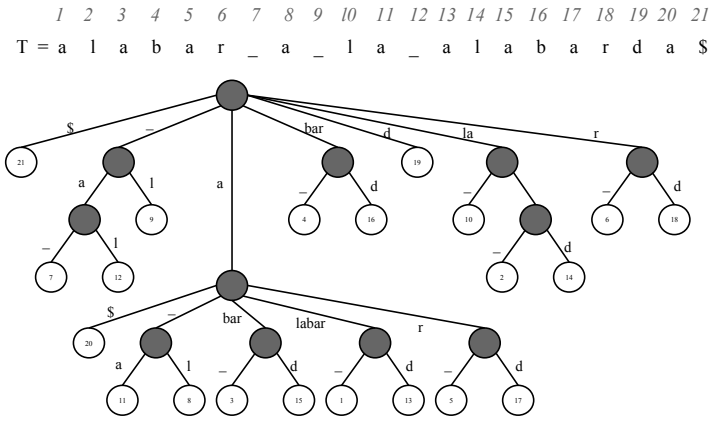
ExtractSuffix(Ψ, W, D, i, m)
  while $m > 0$ **AND** $i \neq 1$ **do**
    Report $D[\texttt{rank}_1(W, i)]$
    $i \leftarrow \Psi[i]$
    $m \leftarrow m-1$
  end while

do so we define a bitmap $B[1,n]$ such that $B[i] = 1$ iff $A[i] \bmod \delta = 0$ or $A[i] = n$. Sampling A at regular intervals of the suffix array guarantees that after applying Ψ δ times, we will always find a sampled value of A.

Given A_s, B, and Ψ, the procedure to obtain $A[i]$ is quite simple. We initially check if $B[i] = 1$. If so, $A[i]$ is explicitly stored in $A_s[\texttt{rank}_1(B,i)]$. Otherwise, we apply Ψ as many times as necessary until we reach a position j such that $B[j] = 1$. In this case we return $A_s[\texttt{rank}_1(B,1)] - d$, d being the number of times we have applied Ψ to obtain a sampled position. Note that $d \leq \delta$.

As an example, suppose we want to access $A[8]$ in the suffix array depicted in Figure 11.1, in which we have set $\delta = 5$ and those sampled values of A appear shadowed. As we can see, $A[8]$ is not sampled, thus, we apply $\Psi(8) = 14$. $A[14]$ is not sampled either, so we apply $\Psi(14) = 12$, which is sampled. $A[12] = 5$ and we have applied Ψ $d = 2$ times, which means that $A[12] = 5 - 2 = 3$.

Having the suffix array available is fundamental to carry out *locate* operations. Note that if $[i,j]$, with $i \leq j$, is an interval of the suffix array that contains all the occurrences of a pattern in the text, then $A[i,j]$ contains all the occurrences of that pattern in the input sequence.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A:	21	7	12	9	20	11	8	3	15	1	13	5	17	4	16	19	10	2	14	6	18
A^{-1}:	10	18	8	14	12	20	2	7	4	17	6	3	11	19	9	15	13	21	16	5	1
ψ:	10	7	11	17	1	3	4	14	15	18	19	20	21	12	13	5	6	8	9	2	16
T_{BWT}:	a	r	a	a	d	l	_	l	l	$	_	b	b	a	a	r	_	a	a	a	a
LCP:	0	0	2	1	0	1	2	1	4	1	6	1	2	0	3	0	0	2	5	0	1
W:	1	1	0	0	1	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	0
D:	$	_	a	b	d	l	r														

Figure 11.1: Suffix tree example with all the components necessary for CSAs and FM-Indexes.

11.1.1.3 Compressing the Inverse of the Suffix Array

Just as for the suffix array A, storing the vector A^{-1} is unfeasible in practice if we care about the space (it would require also $O(n \lg n)$ bits). Instead, we sample some positions of A^{-1} and then we use the data structures we already have to figure out the rest. Contrary to A, A^{-1} is sampled at regular intervals of A^{-1}. This means that we do not need any additional bitmap to mark those positions.

Supposing we have $A_s^{-1}[1, \lceil n/\beta \rceil]$ such that $A_s^{-1}[i/\beta+1] = A^{-1}[i]$ if $i \bmod \beta = 0$, $A_s^{-1}[1] = A^{-1}[1]$, and being $\beta > 0$ the sampling period (typically $\beta = O(\text{polylog}\, n)$), we can access to $A^{-1}[i]$ as follows. If $i \bmod \beta = 0$ then we just report $A_s^{-1}[i/\beta + 1]$. Otherwise, we access to $A_s^{-1}[\lfloor i/\beta \rfloor + 1]$ and we apply Ψ ($i \bmod \beta$) times, returning the last value returned by Ψ.

As an example, suppose we want to access $A^{-1}[19]$ in the suffix array depicted in Figure 11.1, in which $\beta = 5$ and sampled values of A^{-1} appear shadowed. $A^{-1}[19]$ is not sampled, thus we have to access the previously sampled position of A^{-1}, which is at position 15. This means that $A^{-1}[19] = \Psi^{(19-15)}(A^{-1}[15]) = \Psi^4(9) = 16$, where $\Psi^x(i) = \Psi^{x-1}(\Psi(i))$, $x > 0$, and $\Psi^1(i) = \Psi(i)$.

The inverse suffix array A^{-1} helps on $\texttt{extract}(T, i, j)$ queries. Note that $A^{-1}[i]$ reports the position where the suffix $T[i, n]$ is in the suffix array. This permits us to check in which region of the suffix array is that suffix, and thus to obtain the symbol at $T[i]$ by using the vector C and the dictionary of symbols.

11.1.2 FM-Indexes

Modern FM-Indexes [FMMN07] build all their functionality on `access` and `rank` queries on the BWT (Burrows Wheeler Transform) [BW94] of T, T_{BWT}. The T_{BWT} is a permutation of T such that $T_{BWT}[i] = T[SA[i] - 1]$, being $T_{BWT}[1] = T[n]$ and $1 < i \leq n$. That is, T_{BWT} is formed by sequentially scanning the suffix array A, annotating the symbol preceding each suffix.

Note the string T_{BWT} is a reordering of the symbols of T, therefore $\mathcal{H}_0(T_{BWT}) = \mathcal{H}_0(T)$. Thus, zero-order-compressed representations of T also obtain zero-order compression of T_{BWT}. However, some kinds of zero-order compressors, in particular wavelet trees or matrices with bitmaps compressed with Raman et al. (`RRR` from Section 2.6) obtain $n\mathcal{H}_k(T)$ bits of space for any $k < \log_\sigma n$ [MN08] when applied to T_{BWT} [MN08].

Supposing we are given the vector $C[1, \sigma]$ (same meaning as for CSAs, recall Section 11.1.1), the LF function is defined as $LF(i) = C[T_{BWT}[i]] + \texttt{rank}_{T_{BWT}[i]}(T_{BWT}, i)$. LF returns the position in T_{BWT} (or equivalently in the suffix array) of the suffix $T_{BWT}[i]T[A[i], n]$. If the core of CSAs is the Ψ function, which permits to virtually move forward in the text, that of FM-Indexes is the LF operation, which permits to virtually move backwards. Actually, $LF(\Psi(i)) = i$, which means that Ψ is the inverse function of LF.

Algorithm 16 count operation for a pattern $P[1,m]$ in an FM-Index.

count$(P[1,m], T_{BWT}, C[1,\sigma+1])$
  $sp \leftarrow C[P[m]]+1$
  $ep \leftarrow C[P[m]+1]+1$
  $i \leftarrow m-1$
  while $i > 0$ **do**
    $sp \leftarrow C[P[i]] + \texttt{rank}_{P[i]}(T_{BWT}, sp-1)+1$
    $ep \leftarrow C[P[i]] + \texttt{rank}_{P[i]}(T_{BWT}, ep)$
    if $sp < ep$ **then**
      return 0
    end if
    $i \leftarrow i-1$
  end while
  return $ep - sp + 1$

As an example, suppose we want to carry out $LF(13)$ in Figure 11.1. Since $T_{BWT}[13] = b$, and $C[b] = 13$, $LF(13) = C[b] + \texttt{rank}_b(T_{BWT}, 13) = 13 + 2 = 15$.

Algorithm 16 shows how to carry out *count* queries on an FM-Index. As we can see, the algorithm is linear in the pattern length, and only depends on the performance of the `rank` operation in T_{BWT}, which is essentially the time to carry out LF mappings. Being that time α, the total time for *count* is $O(\alpha \times m)$.

To support *locate* and `extract`, FM-Indexes basically follow the same strategy than CSAs: Sample A and A^{-1} and use the LF mapping to obtain those values which have not been sampled. The only particularity is that LF permits to virtually move backwards in the text, not forward as Ψ does, so we have to slightly adapt the algorithms.

11.2 Self-Indexes on Highly Repetitive Scenarios

The BWT is typically formed by a few long *runs* of equal symbols: the number of runs is at most $n\mathcal{H}_k(T)+\sigma^k$ for any k [MN05], and the number is much lower on repetitive sequences [MNSV10]. Thus, in a highly repetitive scenario, the runs of T_{BWT} are much longer than $\log_\sigma n$, and thus typical kth-order statistical compression of T_{BWT} fails to capture its most important regularities. Something similar happens with CSAs, in which long sequences of consecutive increasing numbers appear in Ψ [MNSV10].

Run-Length FM-Indexes (`RLFMIs`) and CSAs (`RLCSAs`) [MN05, MNSV10] aim at capturing these regularities. A Run-Length FM-Index stores in T'_{BWT} the first symbol of each run, marking their positions in a bitmap $R[1,n]$ (they also store a bitmap $R'[1,n]$ with a reordering of the bits in R). CSAs have also been adapted to exploit runs in a structure called Run-Length CSA [MNSV10] by using δ-codes (see

Section 2.5) to differentially encode the Ψ function (along with a sampling strategy to avoid decompressing the whole sequence when we access it).

11.3 Grammar and Block-Tree FM-Indexes

An alternative to run-length encodings to exploit the regularities of T_{BWT} consists of using a sequence representation for T_{BWT}. However, until the presentation of the Grammar Compressor with Counters (`GCC`, Chapter 6) and the Block-Tree (`BT`, Chapter 7), we were not aware of any practical data structure that was able to: (a) exploit the repetitions of T_{BWT}, and (b) efficiently solve `rank` queries on it.

Therefore, what we propose in this chapter is to use our `GCC` and `BT` to represent T_{BWT}. This permits us to obtain a *self-index* with different space time trade-offs on highly repetitive inputs. We dubbed our solutions `GFMI` and `BTFMI`, from Grammar compressed and Block-Tree based FM-Index respectively.

11.4 Experimental Results

To experimentally evaluate our proposals, we use datasets `para`, `influenza`, `escherichia`, `fiwikitags`, `einstein`, and `software`, all of them described in Section 5.6.

To evaluate if grammar and block-tree compression of T_{BWT} are better at capturing regularities than Run-Length based approaches, we compare the following self-index implementations:

- `FMI-GCC`, using the technique `GCC` presented in Chapter 6 to represent T_{BWT}.
- `FMI-AP.RP.WTRP`, using the variant `AP.RP.WTRP` to represent T_{BWT}.
- `FMI-BT`, using the `BT` technique presented in Chapter 7 to represent T_{BWT}.
- `FMI-WMH.BT`, which is the extension of `BT` that adapts better to the increase of the alphabet size presented in Chapter 7 to represent T_{BWT}.
- `FMI-WTH.RRR`, which uses `WTH.RRR` (Huffman-shaped wavelet tree with bitmaps compressed with `RRR`, recall Sections 2.7.2, and 2.6) to represent T_{BWT}.
- `FMI-WT.RRR`, which uses `WT.RRR` (wavelet tree, recall Section 2.7.1, with bitmaps compressed with `RRR`)to represent T_{BWT}.
- `RLFMI-WTH+DELTA`, a Run-Length FM-Index of Section 11.2 where bitmaps R and R' are compressed with `DELTA` (recall Section 2.6), while S'_{BWT} is represented with `WTH.RRR`.
- `RLCSA`, a Run-Length Compressed Suffix Array of Section 11.2, setting the sampling rate of its function Ψ to $\{32, 64, 128\}$.

We used an Intel(R) Xeon(R) E5620 at 2.40GHz with 96GB of RAM memory, running GNU/Linux, Ubuntu 10.04, with kernel 2.6.32-33-server.x86_64. All our implementations use a single thread and are coded in `C++`. The compiler is `g++` version 4.7, with `-O9` optimization. We implemented our solutions inside LIBCDS [Cla].

We evaluate the performance of the operation `count` in the indexes, for various pattern lengths, reporting the average time for 10,000 queries consisting of patterns picked at random from each dataset. Figures 11.2,11.3,11.4, and 11.5 show the results for pattern lengths of $m = 2, 4, 8, 16$ respectively.

As it can be seen, `FMI-GCC` obtains the least space on the smaller alphabets. The space of `RLCSA` is close, but still larger than that of `FMI-GCC`, in collections `fiwikitags` and `influenza`. For `para` and `escherichia` the differences are larger, our structure using 60%–80% of `RLCSA` space. Interestingly, grammar compression of T_{BWT} is stronger than `RLCSA` compression, especially when the sequence is not so repetitive. In exchange, `RLCSA` is about an order of magnitude faster. On the other hand, `FMI-BT` and `FMI-WMH.BT` do not seem to be very good options since they are not in the pareto-optimal set for any dataset.

`FMI-GCC` also uses half the space, or less, than `RLFMI-WTH+DELTA`, which also adapts to repetitiveness but not as well as grammar compression, and performs badly as soon as repetitiveness starts to decrease. Comparing `FMI-GCC` with the best statistical approach, `FMI-WTH.RRR`, the differences are even larger: our solution needs only 20%–40% of the space in the most repetitive collections, only getting closer in `escherichia`, which is not so repetitive.

On the larger alphabets, instead, `FMI-AP.RP.WTRP` outperforms `FMI-GCC` and uses about the same space as the `RLFMI-WTH+DELTA`, while being faster or equally fast. It is only 2–4 times slower than the statistical approaches, while using 10%–20% of their space. However, as expected, `RLCSA` outperforms every FM-index on larger alphabets. Yet, in some applications the FM-index cannot be replaced by a `RLCSA`, as specific properties of the BWT are used [Ohl13].

In the sequel we call `GFMI` to `FMI-GCC` or `FMI-AP.RP.WTRP`, whichever is better.

11.5 Discussion

In this chapter we have presented `GFMI` and `BTFMI`, two new FM-Indexes on highly repetitive sequences. The experimental evaluation showed that our `GFMI` generally obtains the best state of the art space performance, albeit it is from 2-4 times to an order of magnitude slower than typical solutions based on FM-Index and CSAs. However, `BTFMI` is generally overcomed by its competitors, remaining as a not recommendable solution (except in `fiwikitags`, where `BTFMI` manages to outstand since it is one of the most repetitive datasets). If we need an FM-Index based solution instead of an `RLCSA`, then the `GFMI` may be a good alternative.

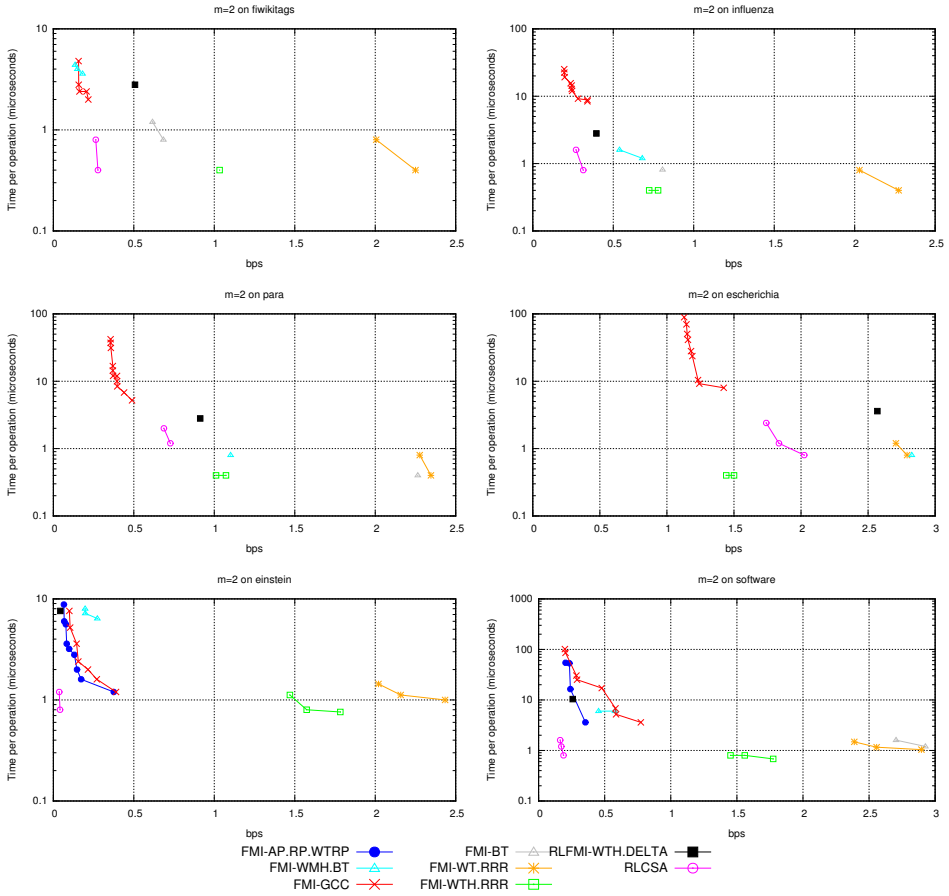


Figure 11.2: Space-time tradeoffs for operation `count` with $m = 2$.

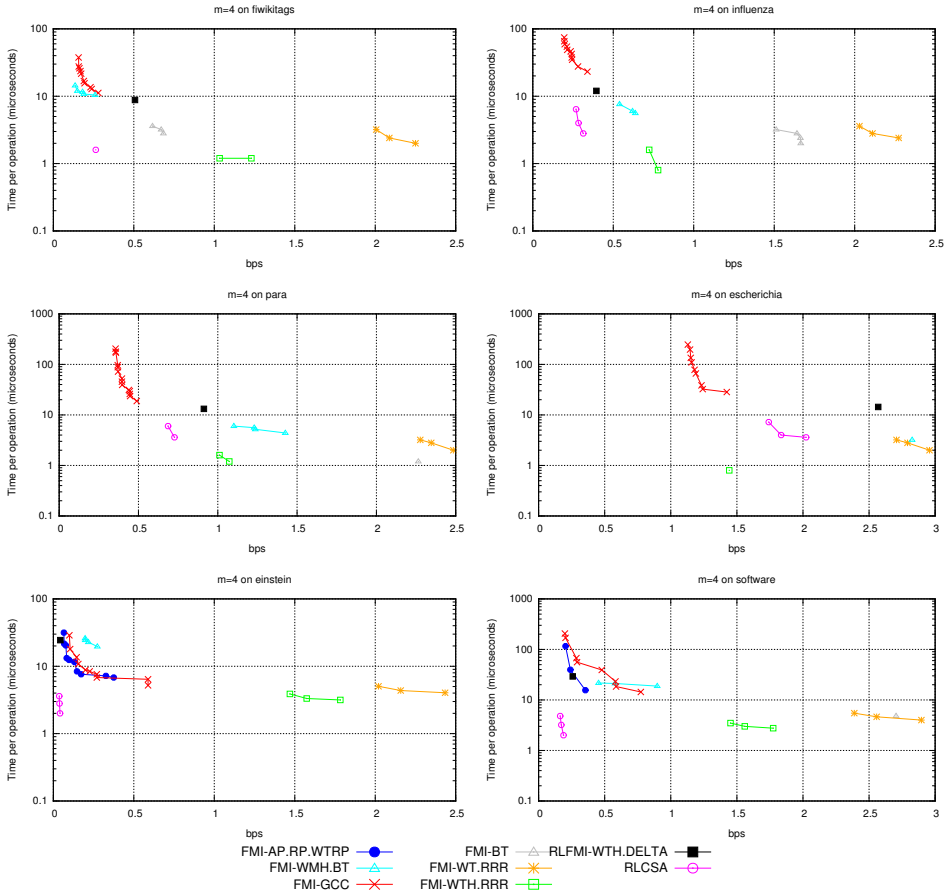


Figure 11.3: Space-time tradeoffs for operation `count` with $m = 4$.

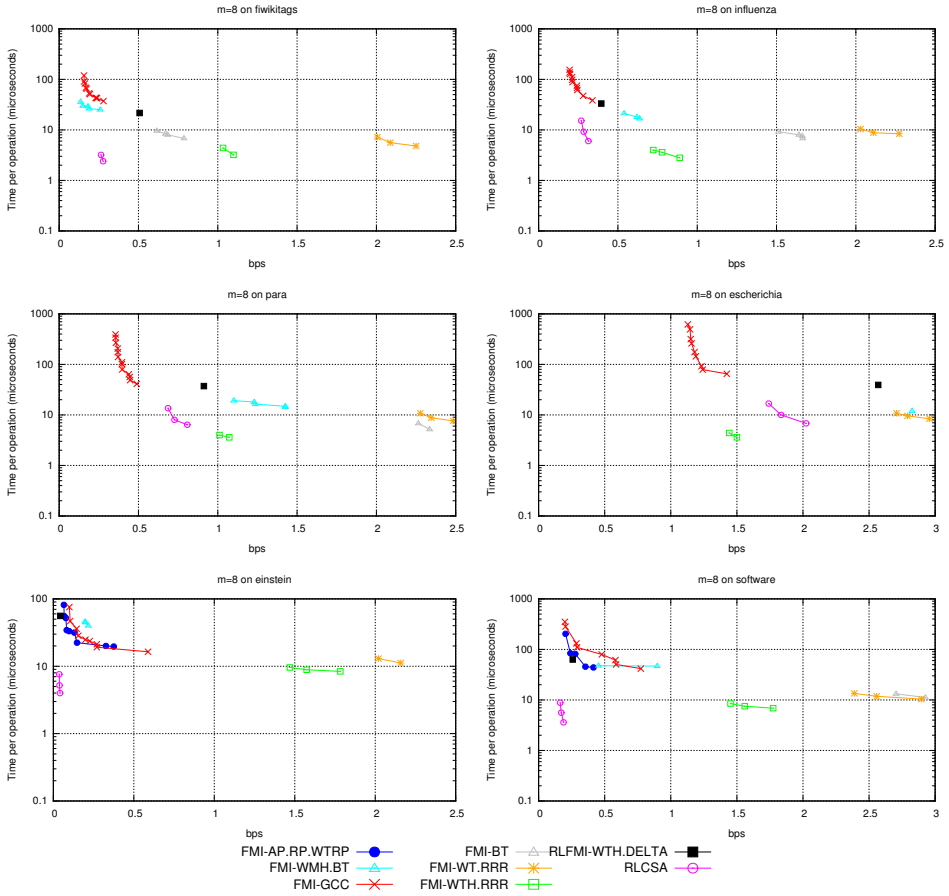


Figure 11.4: Space-time tradeoffs for operation `count` with $m = 8$.

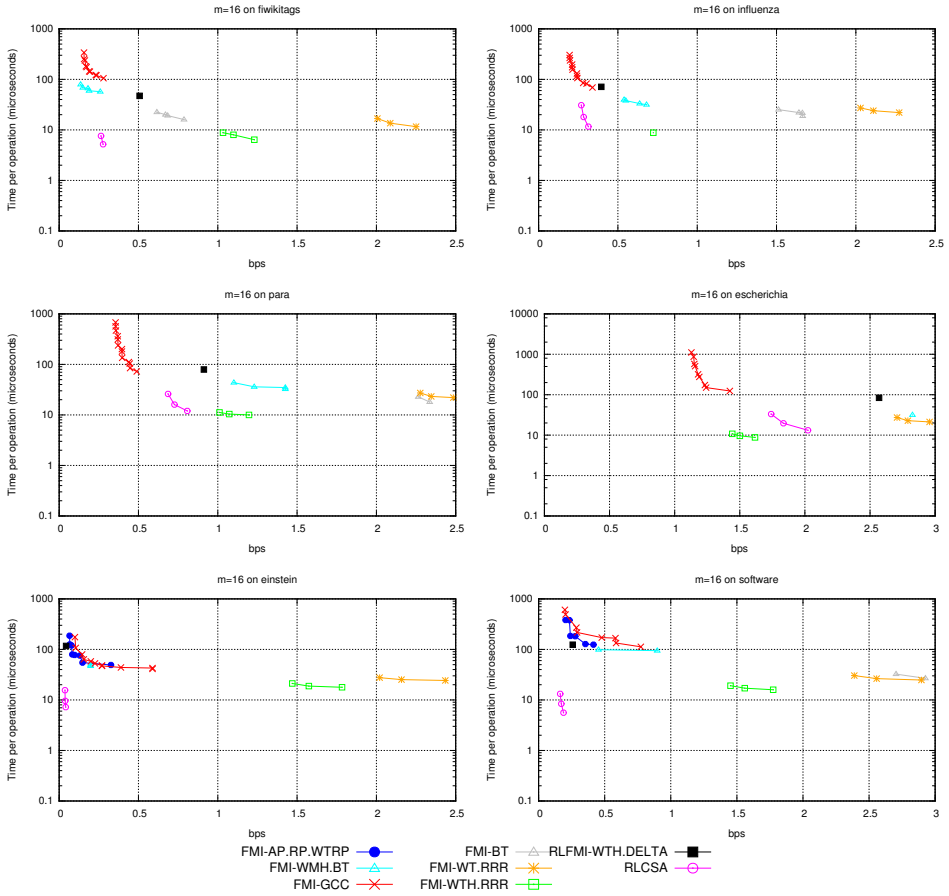


Figure 11.5: Space-time tradeoffs for operation `count` with $m = 16$.

Chapter 12

XPath on Repetitive XML

XML (*eXtensible Markup Language*) has become a standard in the storage, transmission, and manipulation of data along the past few years. Some reasons of its popularity may be the possibility of extending its mark-up, the fact of being easy to parse, its intensive use in web-services or simply its versatility to store semi-structured data.

Additionally, the advent of several systems that made possible to carry out complex queries on XML might have also helped. The XQuery language [XQu] defines a standard to query these kind of files, although implementing it can be challenging both in theory and practice. For that reason, in this chapter we focus only in a subset of XQuery language, XPath [XPa], which is sufficiently expressive for most applications.

In order to carry out XPath queries, beyond several theoretical proposals, only a few disk-oriented implementations (data + indices designed for secondary memory) exist: Qizx/DB [Mp07], MonetDB/XQuery [BGVK+06], and Tauro [Sig08]. Trying to overcome disk latencies, several in-memory systems like Saxon [Kay08], GALAX [FSC+03] or Qizx/Open [Mp07] were also proposed. Typically, in-memory systems overcome disk-oriented ones, although they are not free of exceptions. According to some previous works like Arroyuelo et al. [ACM+15], in-memory systems can reach memory peaks of 5-10 times the size of the input XML depending on the nature of the input file, which means having to rely on disk and the consequent performance downgrade.

A step-forward to avoid this situation was recently taken. It consisted of using compressed data structures to increase the chance of fitting the whole index in main memory. The proposed solution should not only be compact but also functional, being able to support at least an important set of XPath queries. Representatives of this research line are systems like XXS [BCPN14] and `SXSI` [ACM+15].

Those compressed representations are not aimed at highly repetitive scenarios, which also arise in XML collections. For instance, software repositories or versioning

storage systems have in common the need to store and manage highly repetitive collections of XML files. These collections usually store many versions of the same file with probably tiny differences between one version and the next. This is an ideal scenario for repetition-based data structures, although no XPath system has been adapted to this scenario.

In this chapter we show how to re-engineer the internal data structures of `SXSI` [ACM+15] to adapt it to highly repetitive scenarios. By doing so, we end up obtaining an XPath system that takes a small fraction of the space of its statistically compressed counterparts, providing the same functionality at the price of increasing the search times. This permits to have a highly efficient storage system that still permits to carry out search queries without decompressing the whole dataset.

This chapter is organized as follows: Section 12.1 briefly details the main components of the `SXSI` system [ACM+15] and explains which data structures we use to adapt it to highly repetitive inputs; Section 12.2 presents an brief experimental evaluation; and finally Section 12.3 gives our conclusions.

12.1 SXSI on Highly Repetitive Scenarios

As previously said, `SXSI` [ACM+15] is a recent system that represents XML datasets in compressed form and solves XPath queries on them. Its query processing strategy uses a tree automaton that traverses the XML data, using several queries on the content and structure to speed up navigation towards the points of interest. `SXSI` represents the XML data using three separate components:

1. a text index that represents and carries out pattern searches over the text nodes (any compressed full-text index [NM07] can be used);

2. a balanced parentheses representation of the XML topology that supports navigation using $2 + o(1)$ bits per node (various alternatives exist, many of them described in Section 2.9);

3. an `rsa`-capable representation of the sequence of the XML opening and closing tags, using some sequence representation.

When the XML collection is repetitive (e.g., versioned collections like Wikipedia, versioned software repositories, etc.), one can use the `RLCSA` (see Section 11.2) as the text index for (1), but now we also consider using our `GFMI` of Chapter 11. Components (2) and (3), which are usually less relevant in terms of space, may become dominant if they are represented without exploiting repetitiveness. For (2), we consider `GCT`, the grammar compressed tree topology presented in Chapter 8, and a classical representation (`FF`, see Sections 2.9, and 8.1). For (3), we will use our most compact repetition-aware sequence representation presented in Chapter 6 (`GCC`) as well as `BT` from Chapter 7, comparing them with the alternative proposed

in SXSI (MATRIX of Section 2.7, using one compressed bitmap per tag) and a WTH representation of Section 2.7.2.

12.2 Experimental Results

To show the performance of SXSI on highly repetitive scenarios, we use a repetitive data-centric XML collection of 200MB from a real software repository. Its sequence of XML tags, called software, is described in Section 5.6. As a proof of concept, we run two XPath queries that make intensive use of the sequence of tags and the tree topology: XQ1=//class[//methods], and XQ2=//class[methods].

We used an Intel(R) Xeon(R) E5620 at 2.40GHz with 96GB of RAM memory, running GNU/Linux, Ubuntu 10.04, with kernel 2.6.32-33-server.x86_64. All our implementations use a single thread and are coded in C++. The compiler is g++ version 4.7, with -O9 optimization. We implemented our solutions inside Libcds [Cla] and use Navarro's implementation of *RePair* (www.dcc.uchile.cl/gnavarro/software/repair.tgz).

Table 12.1 shows the space in bpe (bits per element) of components (2) and (3). An element is an opening or a closing tag, so there are two elements per XML tree node. The space of the RLCSA without sampling is always 0.18 bits per character (bpc) of the XML document, whereas our new GFMI uses 0.15 bpc if combined with AP.RP.WMRP. The table also shows the impact of each component in the total size of the index, considering this last space. On the rightmost columns, it shows the time to solve both queries.

The original SXSI (MATRIX+FF) is very fast but needs almost 14 bpe, which amounts to 98% of the index space in this repetitive scenario (in non-repetitive text-centric XML, this space is negligible). By replacing MATRIX by a WTH, the space drops significantly, to slightly over 4 bpe, yet times degrade by a factor of 3–6. By using our GCC for the tags, a new significant space reduction is obtained, to 2.65 bpe, and the times increase by a factor of 2, becoming 6–12 times slower than the original SXSI. If we use BT, the space increases considerably with regard to GCC while times are approximately halved. Finally, changing FF by GCT [NO14a], we can reach as low as 0.56 bpe, 24 times less than the original SXSI, and using around 60% of the total space. Once again, the price is the time, which becomes 50–90 times slower than the basic SXSI. The price of using the slower GCT is more noticeable on XQ2, which uses more operations on the tree.

While the time penalty is 1–2 orders of magnitude, we note that the gain in space can make the difference between running the index in memory or on disk; in the latter case we can expect queries to be up to 6 orders of magnitude slower.

dataset	tags	tree	%tags	%tree	%text	XQ1	XQ2
MATRIX+FF	12.4	1.27	88.95	9.11	1.08	16	35
WTH+FF	2.88	1.27	65.16	28.73	3.39	92	113
GCC+FF	0.37	1.27	19.37	66.49	7.85	184	226
BT +FF	1.7	1.27	52.47	39.20	4.63	98	131
GCC+GCT	0.37	0.19	44.58	22.89	18.07	807	3066
BT+GCT	1.7	0.19	78.70	8.80	6.94	690	2750

Table 12.1: Results on XML. Columns **tags** and **tree** are in bpe. Columns **XQ1** and **XQ2** show query time in microseconds.

12.3 Discussion

There exist XML collections formed by highly repetitive data. Applications managing these datasets, as well as systems to compute metrics from them, typically require to carry out complex XPath queries. In the previous experimental evaluation we have shown that by applying grammar or block-tree compression techniques to the internal data structures of the original `SXSI` [ACM+15], the space usage is drastically shrunk. This is of special interest on software repositories or versioning systems in which reducing storage costs is a must. Although the time penalty is rather high, it is still of interest in many scenarios in which this is not a limitation, for example periodically computed metrics.

Chapter 13

GCST: Grammar Compressed Suffix Tree

Suffix trees [Wei73, McC76, Ukk95] are a favorite data structure in stringology, with a large number of applications in bioinformatics [Apo85, Gus97, Ohl13], thanks to their versatility. By means of a small set of query and traversal primitives (see Tables 2.2 and 13.1), suffix trees yield efficient solutions to many complex problems on pattern matching, pattern discovery, string comparisons, and others. The main problem of suffix trees is their space usage, which can easily reach 20 bytes per text symbol. On DNA sequences, where each base can be represented in 2 bits, the suffix tree takes up to 80 times the text size!

A solution to the space problem could be to deploy the suffix trees on secondary memory [FG99, CF02, KR03, DKMS08, FGM12, KK14a, KK14b, GMC$^+$14]. Unfortunately, most of the complex tasks carried out on suffix trees need to traverse them across arbitrary access paths, in which case secondary memory representations perform poorly due to the low locality of reference. The fact that suffix trees use much space but need to fit in main memory to operate efficiently restricts their applicability to small sequence collections only; for example, handling just one human genome requires a machine with 60GB of RAM.

A number of engineered representations of suffix trees have been proposed to cope with their space problem [Kur99, AKO04], but these still take 6 to 10 bytes per symbol. Suffix arrays [MM93] reduce the space to about 4 bytes per symbol, but they lose a number of suffix tree functionalities that are essential in many complex tasks (e.g., suffix links).

The emergence of compressed suffix arrays (CSAs) (see Chapter 11), which managed to represent *both* the sequence and its suffix array within the space of the *compressed* sequence, paved the way to radical improvements in the area, by making it possible to build compressed suffix trees (CSTs) on top of CSAs. Sadakane [Sad07a]

introduced the first CST representation, which included a succinct representation of the tree topology and retained full suffix tree functionality. A recent, well engineered implementation by Gog [Gog11], requires about 10 bits per symbol (bps), that is, slightly more than one byte per symbol, on general DNA text (this includes the storage of the CSA, and thus of the sequence itself), and can perform all the operations in a few microseconds; the easy ones may need just a few nanoseconds. Fischer et al. [FMN09, Fis10] developed a new CST using even less space. An efficient variant by Ohlebusch et al. [OFG10] was shown to use about 8 bps [Gog11]. Their main idea was to avoid the explicit representation of the tree topology. Their operation times, as a consequence, are higher than Sadakane's, but still within microseconds. Russo et al. [RNO11] introduced an even smaller CST, using about 4 bps, yet raising operation times to milliseconds.

All these CSTs use space proportional to the *empirical entropy* of the text collection [Man01], which is a measure of statistical compressibility. In most DNA collections, however, the empirical entropy is also close to 2 bps, that is, DNA is essentially incompressible with statistical compressors. Still, CSTs operating within microseconds can be built on a human genome (of about 3 billion bases), for example, and maintained in a main memory of about 3–4 GB.

However, the goal of maintaining *one* human genome in main memory has quickly become outdated. The rapid improvements in sequencing technology have driven the growth of large genome repositories. Modern challenges are to handle repositories of thousand genomes (e.g., see the 1000-Genomes project, `http://www.1000genomes.org`). Further, one would like to efficiently perform complex bioinformatic analyses on those huge sequence collections, ideally maintaining a suffix tree on them. Even a CST using 1 byte per symbol is problematic when a thousand genomes must be maintained: we would need 3TB of main memory!

Fortunately, those fast-growing DNA collections are formed by the sequenced genomes of hundreds or thousands of individuals of the same species. This makes those collections highly repetitive.

There have been some indexes aimed at performing pattern matching (i.e., just simple string searches) on repetitive collections based on those techniques [CN10b, KPZ11, KN13, CN12, DJSS12, GGK+12]. However, they do not provide the versatile suffix tree functionality, and they do not seem to yield a way to obtain it. Instead, the so-called run-length CSA [MNSV10] (`RLCSA`), although based in principle on weaker compression techniques, yields a data structure that is useful to achieve CSTs for repetitive collections.

Building on the `RLCSA` (see Section 11.2) and on the CST of Fischer et al. [FMN09], Abeliuk et al. [AN12, ACN13] introduced the first CST for repetitive collections, by using grammar-compressed representations of some of their internal components. On the repetitive biological collections they tested, their CST used around 1–2 bps, well below the spaces achieved with the general-purpose CSTs. Their operation time was, however, in the order of milliseconds, which makes the structure far less

attractive.

Our proposal is called GCST, for "grammar-compressed suffix tree", and achieves low space on repetitive collections and much better times. The GCST operates in the order of microseconds, becoming much closer to the times of general-purpose CSTs [Sad07a, OFG10, Gog11], and actually outperforming the smallest members of that family [RNO11, CN10a, ACN13] (which are still significantly larger than the GCST on repetitive collections). On synthetic DNA collections with 99.9% similarity, our GCST uses 2 bps, whereas the previous CST for repetitive collections uses 1.5 bps, and their difference shrinks as the collections become more repetitive. In exchange for this higher space, the GCST is up to 3 orders of magnitude faster.

To achieve this result, we build on the CST of Sadakane [Sad07a], but use grammar compression on the tree topology, instead of just a succinct representation. More precisely, we use our GCT (Grammar Compressed Tree) of Chapter 8. A repetitive text collection turns out to have a suffix tree with repetitive topology, and having the tree represented in this form allows us to speed up many operations that are very slow to simulate without the explicit topology [FMN09, RNO11].

The GCST retains the full functionality of succinct tree representations [NS14] (see also Section 8.1), but is likely to use much less space when the tree has frequent repeated substructures. While we do not prove worst-case results on the GCST representation, our experiments show that it performs well in the scenario studied in this chapter.

The chapter is organized as follows: Section 13.1 presents the state of the art of compressed suffix trees; Section 13.2 describes our proposal; Section 13.3 provides a complete experimental evaluation of our proposal, comparing it with the state of the art techniques previously described; and finally Section 13.4 presents our conclusions and addresses the future work.

13.1 Current Compressed Suffix Trees

In Chapter 11 we already presented suffix arrays, compressed suffix arrays, and how we may adapt them when dealing with highly repetitive collections with *Run-Length-* or *grammar*-based data structures. Although these are of major interest, not less important is the concept of *longest common prefix (LCP)* array, since it is a key component of various suffix tree representations.

The $LCP[1,n]$ array stores in $LCP[i]$ the length of the longest common prefix between the suffixes $T[A[i],n]$ and $T[A[i-1],n]$ (with $LCP[1] = 0$). Sadakane [Sad07a] showed how to represent LCP using just $2n$ bits, by representing $PLCP[1,n]$ instead, where $PLCP[j] = LCP[A^{-1}[j]]$ (or $LCP[i] = PLCP[A[i]]$), that is, $PLCP$ is LCP represented in text order, not in suffix array order. The key property is that $PLCP[j+1] \geq PLCP[j] - 1$, which allows $PLCP$ be represented using a bitvector $H[1,2n]$, at the price of having to compute $A[i]$ in order to compute $LCP[i]$.

Operation	Description
$sDepth(v)$	$\|str(v)\|$
$letter(v, i)$	$str(v)[i]$
$child(v, a)$	u such that $a \in \Sigma$ is the first letter on edge (v, u)
$sLink(v)$	u such that $str(u) = \beta$ in case $str(v) = a\beta$ and $a \in \Sigma$
$LAQs(v, d)$	the highest ancestor u of v such that $\|str(u)\| \geq d$
$locate(v)$	i such that $str(v)$ starts at $T[i]$ (for a leaf v)

Table 13.1: Typical operations supported by a suffix tree (besides those of Table 2.2). By $str(v)$ we denote the string obtained by concatenating the labels on the edges between the root and v.

Fischer et al. [FMN09] proved that H was in addition compressible when the text was statistically compressible, but Cánovas and Navarro [CN10a] found out that the compression was not significant on standard texts. Instead, Abeliuk and Navarro [AN12] showed that the technique proposed to compress H [FMN09] worked very well on repetitive texts.

Having this in mind, Sadakane [Sad07a] showed that a functional compressed suffix tree (CST), that is, a data structure that solves the operations in Table 2.2 as well as those in Table 13.1, could be represented with three components:

1. A compressed suffix array (CSA).
2. A compressed LCP array.
3. A representation of the topology of the suffix tree.

Other elements, like the string labels, were computed from these components without representing them.

Concretely, Sadakane used an existing CSA, compressed the LCP array to $2n$ bits, and represented the tree topology using succinct trees, which take $2n$ to $4n$ bits since the suffix tree has $t = n$ to $2n$ nodes. A study of such succinct tree representations [ACNS10] shows that a Balanced Parentheses implementation like `FF` (see Sections 2.9 and 8.1) is well suited for the operations required on a suffix tree. Gog [Gog11] implemented Sadakane's CST, obtaining extremely fast operations.

Alternatively, Fischer et al. [FMN09] showed that one can operate without explicitly representing the tree topology, because each suffix tree node corresponds to a distinct suffix array interval. One can operate directly on those intervals, and all the tree operations can be simulated with three primitives on the intervals: $RMQ(i, j)$ finds the (leftmost) position of the smallest value in $LCP[i, j]$, and $PSV/NSV(i)$ finds the position in LCP preceding/following i with a value smaller than $LCP[i]$.

Cánovas and Navarro [CN10a] implemented this theoretical proposal, speeding up the operations RMQ and PSV/NSV by building the balanced tree described in Section 8.1 on top of the LCP array (instead of on array E) and using ideas similar to those used to navigate trees [NS14] (albeit the application is quite different). Ohlebusch et al. [OFG10] presented a fast alternative implementation that uses $3n$ bits of space.

Abeliuk and Navarro [AN12] proposed the first CST for repetitive text collections. They built on the representation of Cánovas and Navarro [CN10a], using the `RLCSA` and the compressed version of H to represent LCP, which became compressible on repetitive texts. The only obstacle was that the balanced tree used to speed up RMQ and PSV/NSV operations was insensitive to repetitiveness. They overcame this by using the fact that the differential LCP array ($LCP[i] - LCP[i-1]$) is grammar-compressible, particularly on repetitive text collections. They applied *RePair* compression (see Section 5.4) to the differential LCP array and used the grammar tree (whose nodes are the grammar nonterminals) instead of the incompressible balanced tree. That is, they stored the information needed to compute $PSV/NSV/RMQ$ in the nodes of the grammar tree. As a result, they obtain very low space usage on repetitive texts (from 0.6 to 4 bps, depending on the repetitiveness of the real-life collections used). A drawback is that the operations require milliseconds, instead of the microseconds required by most CSTs designed for standard text collections [ACN13].

13.2 Grammar Compressed Suffix Tree

We introduce a new CST which builds on the original proposal of Sadakane [Sad07a] but tailored to repetitive texts. We use the `RLCSA` as the suffix array, and the compressed representation of H [FMN09, AN12] for the LCP array. Unlike the previous CST of Abeliuk and Navarro, we do represent the suffix tree topology with our `GCT` (recall Chapter 8), to avoid the huge time performance penalty of omitting it. As a result, our `GCST` will use slightly more space than that of Abeliuk and Navarro [AN12], but it will be orders of magnitude faster.

13.3 Experimental Results

In order to evaluate the performance of our proposal, we use several synthetic and real datasets, all of them described in Section 5.6. Concretely, we used collections `DNA.1`, `DNA.01`, `DNA.001`, `DNA.0001`, `influenza`, `escherichia`, `para`, and `einstein`.

13.3.1 Space Usage

Figure 13.1 gives a space breakdown of our `GCST` representation for each dataset in bps. The breakdown has five parts: (1) the `RLCSA`, which is built with parameters

blockSize = 32 and *sample* = 128 to provide reasonable time performance; (2) the LCP representation; (3) the representation of the rules R of the `GCT`; (4) the representation of sequence C of the `GCT`; (5) the extra data we store for R and C associated with samples. For this last part, we tested various values of $y \in \{2^0, 2^1, 2^2, 2^4, 2^8\}$ and $z \in \{2^8, 2^{10}, 2^{12}, 2^{14}\}$ (same as in Section 8.3). Obviously, this is the only part of the space that changes with y and z. We used the balanced version of *RePair* (see Section 5.4), which consistently gave us better results.

In the synthetic DNA collections, the space decreases as repetitiveness increases. The fixed part of the structures (without the sampling data on R and C) goes from about 0.85 bps on the most repetitive collection to about 4.7 bps when the mutation rate reaches 1%. Note that, from this space, about 0.6 bps from the `RLCSA` are fixed and insensitive to repetitiveness; this is the space used by the `RLCSA` samples. Components LCP, R and C decrease monotonically with repetitiveness.

The space for the R and C samplings varies significantly with parameter z, but not so much for y, as previously analyzed in Section 8.3.

On the real data, the situation is more or less the same. Using reasonable values for z, the space is about 7 bps for `escherichia`, the least repetitive collection. However, it decreases to about 3.3 bps on `para` and to 2.3 on `influenza`, which is much more repetitive. On `einstein`, the most repetitive collection, this space is below 0.7 bps.

13.3.2 Space-Time Performance of Operations

We compare space and time performance of our `GCST` with previous CSTs, for a number of suffix tree operations. The CSTs considered are the following.

GCST Our new suffix tree representation. We used various combinations of parameters y and z, obtained a cloud of points, and chose the dominant ones. In most cases, this implies leaving y at sampling every nonterminal and using z to reduce the space. The `RLCSA` parameters are fixed to 32 for the sampling of Ψ and 128 for the text sampling (the one that affects the computation of suffix array entries). Note the total space of a `GCST` is that of the `GCT` of Chapter 8 plus that of the additional data structures necessary to build a fully-functional CST (LCP and `RLCSA`).

Sada Sadakane's CST [Sad07a] adapted to repetitiveness but without including our new grammar-compressed tree topology (the actual index is too large to be of interest in this comparison). That is, we use the `RLCSA` as the suffix array, the compressed-bitvector H for the LCP, and the plain representation that uses 2 bits per node [NS14, ACNS10] for the topology. This allows us to measure the effect of our grammar-compressed topology in time and space. We use sampling 32 for Ψ and 64, 128, and 256 for the text sampling. Again, note the total space of this approach of that of the Sada data structure of Chapter 8

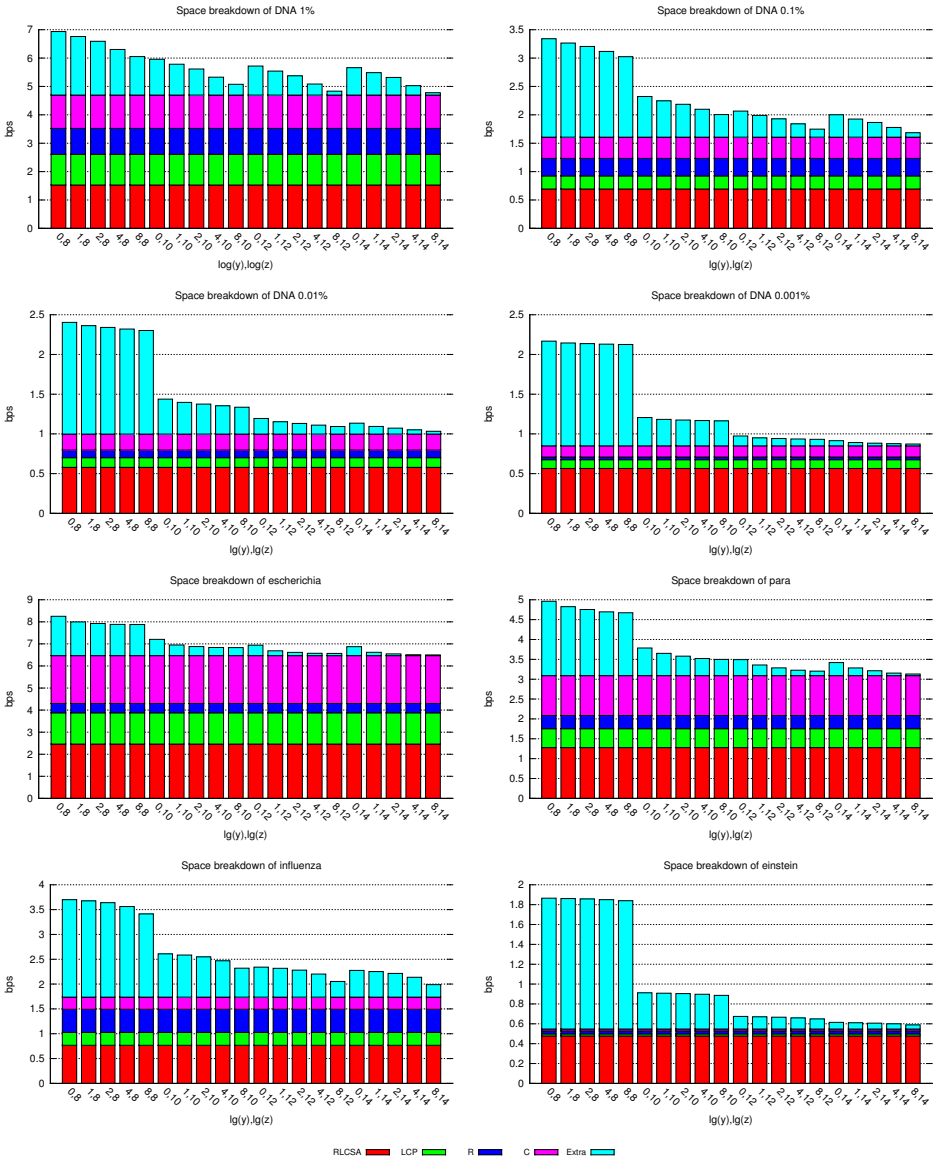


Figure 13.1: Space breakdown of our `GCST` representation for the different collections and combinations of parameters y (rule sampling) and z (sampling of C).

plus that of the additional data structures necessary to build a fully-functional CST (*LCP* and `RLCSA`).

SCT3 The fastest CST for general collections among those that use reasonable space [OFG10]. It is also the most compact CST implemented in the SDSL library [Gog11] (called `cst_sct3` in SDSL). For the CSA it uses an FM-index on Huffman-shaped wavelet trees [FMMN07], which makes it small and fast on DNA. It uses a non-compressed bitvector H to represent the *LCP*, and a structure of $3n$ bits to solve *PSV*/*NSV* operations. The tree topology is not represented. The bitvector samplings is set to 63, the sampling to extract text to 63, and the text sampling to 32, 64, and 128. For the rest, it was compiled with the default configuration of SDSL.

NPR-Repet The only previous CST designed for repetitive collections [AN12, ACN13]. We choose the best point between using balanced or unbalanced RePair in each case. They run over the `RLCSA`, with sampling 32 for Ψ and 64, 128, 256, and 512 for the text.

NPR The smallest CST for general collections that achieves times within microseconds [CN10a, ACN13]. Among their many variants, we use the so-called FMN-RRR, which uses the least space. To make it more space-competitive in this scenario, we change its suffix array to the `RLCSA`, with the same sampling choices of NPR-Repet.

FCST The smallest CST for general collections [RNO11]. The FCST is much slower than NPR; its times are in the range of the milliseconds, close to those of NPR-Repet. The FCST also uses an FM-index on wavelet trees [FMMN07] as its suffix array.

We exclude the faster and larger variants of NPR [CN10a, ACN13], as they represent *LCP* values directly and these become very large on repetitive collections ($\approx$ 27 bps only the *LCP*s!). Other larger variants implemented in SDSL are also disregarded in this comparison.

We note that not all the previous CSTs implement all the operations, so they may not appear in some plots. In addition, we were unable to build NPR-Repet on the most repetitive dataset, `DNA 0.001`, because its grammar-compression algorithm on the differential *LCP* array crashed.

We ran the experiments in an isolated Intel(R) Core(TM) i7-3820 running at 3.60GHz with 62GB of RAM memory. The operating system is GNU/Linux, Ubuntu 12.04, with kernel 3.2.0-68-generic.x86_64. All our implementations use a single thread and all of them but FCST are coded in `C++` (FCST is in `C`). The compiler is `gcc` version 4.6.3, with `-O9` optimization flag set (except SCT3, which uses its own set of optimization flags).

We use five different strategies based on previous work [NS14, ACN13] to extract queries, each defined to correctly measure the performance of a specific type of queries. These strategies are described as follows:

(a) We randomly pick a leaf and collect and report all those nodes in the path from that leaf to the root.

(b) We randomly pick a leaf and collect and report all those nodes in the path from that leaf to the root that have more than 3 children.

(c) We randomly pick a leaf, move to its parent, and then we move towards the root but using *sLink* operations.

(d) We randomly select couples of leaves.

(e) We randomly select leaves with *tDepth* (or *sDepth*) values larger than 10, and then we report that leaf and a value in the range $[1, tDepth \text{ (or } sDepth) - 1]$.

Queries for operations *fChild*, *tDepth*, *sDepth*, *nSibling*, and *parent* we use strategy (a); for *letter*, and *child* strategy (b); for *sLink*, strategy (c); for *LCA*, strategy (d); and for *tAncestor* and *LAQs*, strategy (e). We averaged each data point over 10,000 random queries.

Space

Let us use Figure 13.2 to discuss the space usage of the indexes. The general-purpose indexes are mostly insensitive to repetitiveness (except because in some of those we used the `RLCSA` as the suffix array). Even with the sparsest samplings used, Sada takes up to 10 bps on the least repetitive collections and then decreases to 7 bps on the most repetitive ones. SCT3 (which does not use the `RLCSA`) always uses about 5.5–7 bps. NPR (which uses an `RLCSA`) goes from 6.5 bps on the least repetitive collections and to 4.5 bps on the most repetitive ones. Finally, the FCST (which also does not use an `RLCSA`) is the only one that increases space with repetitiveness, rarely exceeding 4 bps but reaching 5.3 bps on `einstein`. The reason is that repetitive collections induce deeper suffix trees. Since the FCST samples nodes at regular intervals across *sLink* paths, a deeper suffix tree entails longer paths and thus more samples (up to some maximum guaranteed limit).

The repetitiveness-oriented CSTs use significantly less space, NPR-Repet being always smaller than `GCST`. The `GCST` becomes, in broad terms, more competitive with NPR-Repet as repetitiveness increases. While, on the least repetitive `DNA 1`, NPR-Repet can use as little as 2.8 bps, which is about 60% of the `GCST` space, the ratio raises to 80% already for `DNA 0.01`. On the real texts, instead, the ratio stays around 60%, but for the most repetitive `einstein` both indexes use basically the same space.

Note that, on the least repetitive collections, the repetitiveness-oriented CSTs are not interesting anymore: On `DNA 1` and `Escherischia`, the FCST is already smaller than the `GCST` (albeit much slower), and uses about the same space and time of NPR-Repet.

The comparison between `GCST` and Sada shows that compressing the parentheses reduces the space by 2–6 bps, the impact being larger on the more repetitive collections. On those, this difference dominates the total space of the structures, for example Sada is about 7 times larger than the `GCST` on `DNA 0.001` and on `einstein`.

The impact on the times is analyzed next. For the `GCST`, we will comment about the choice of parameter that reaches its "sweet point", which is roughly the left-to-right point where the time ceases to decrease abruptly and stabilizes. This is still a choice of good space usage.

Direct tree operations

Figures 13.2 to 13.6 show the time-space performance for operations *fChild* (requiring just an access to the parentheses), *tDepth* (requiring simple parenthesis operations), *nSibling*, *parent* and *tAncestor* (requiring the more complex *fwd* and *bwd* operations on the parentheses, for which a full description is given in Section 8.1). For *tAncestor* we test with a random depth between 1 and the tree node depth.

Direct tree operations are particularly fast when the topology is represented with parentheses. This is the case of Sada and the `GCST`. In the first case the operation times goes from one nanosecond (ns) to at most one microsecond (μs). The faster ones, running in at most 10 ns, are *fChild*, *tDepth*, and *parent*. Instead, *nSibling* and *tAncestor* are slower, requiring 0.5–1 μs.

The `GCST` is not so fast because it compresses the topology, but still it performs well. It solves *fChild*, *tDepth* and *parent* in 5–10 μs, *tAncestor* in 10–30 μs, and *nSibling* 20–50 μs. That is 1–3 orders of magnitude slower than a plain parentheses representation.

Instead, the operations are 2–3 orders of magnitude slower on NPR, which uses much more space than `GCST` but does not store the tree topology. NPR requires 100–700 μs for operations *fChild*, *nSibling*, and *parent*, except on `influenza` and `einstein`, where for unclear reasons the times drop to 10–80 μs.

Lacking an explicit topology, *tDepth* and *tAncestor* can only be solved via successive *parent* operations until reaching the root or the desired depth difference. This makes these two operations way slower on the other indexes. For *tDepth* the time of NPR reaches 0.7–5 *ms*, and for *tAncestor* it reaches 2–50 *ms* (and 50–300 μs on the two collections where it is faster).

If we consider NPR-Repet, which does not store the tree topology and in addition is optimized for repetitiveness (reaching less space than the `GCST`), the times jump one or two orders of magnitude further: *fChild*, *nSibling*, and *parent* require 0.6–10 *ms* (0.3 *ms* on `einstein`), *tDepth* takes 50–300 *ms* (10 *ms* on `einstein`), and *tAncestor* uses 7–500 *ms*. Therefore, the only previous index that is smaller than the

GCST on repetitive collections is 2–4 orders of magnitude slower than it. The choice of including the parentheses, even if highly compressed and slow to use, definitely pays off.

SCT3 does not represent the topology, but uses $3n$ bits to speed up the operations PSV/NSV on the LCP values. As a consequence, it uses more space than NPR, but it performs significantly faster. For *fChild*, *nSibling* and *parent*, it takes 0.2–2 μs, which is 1–2 orders of magnitude faster than GCST (but still way slower than Sada, which uses plain parentheses). However, it is also 1–2 orders of magnitude slower than GCST for *tDepth* and *tAncestor*, where it takes 10–40 μs and 0.2–2 *ms*, respectively (for unclear reasons, SCT3 is much slower on para).

Finally, the FCST takes 0.6–7 μs on operations *fChild*, *nSibling* and *parent*, and 2–50 *ms* on *tDepth* and *tAncestor*. This is also several orders of magnitude slower than the GCST.

Operation *LCA*

This is the most complex among the operations that only need the topology of the suffix tree. Figure 13.7 shows that the GCST uses 30–100 μs to solve it. NPR requires 0.3–1 *ms* in most cases, and 30–200 μs on influenza and einstein. On the other hand, NPR-Repet requires 0.7–10 *ms*.

The heaviest part of this operation is an *RMQ*. Both SCT3 and Sada have explicit structures to carry out this operation, thus they solve it fast, in 4–5 μs. The FCST is also particularly fast on this operation: 5–20 μs. The reason is that LCA is a core operation for the FCST, so it is solved most efficiently and is the base for the other operations.

Operation *sLink*

The suffix link operation is the first we study that is specific of suffix trees, and can be considered as the one that distinguishes suffix trees from other digital trees. Operation *sLink* requires several tree operations and interacting with the RLCSA. For the GCST and Sada, it requires mapping the node to its suffix array interval (which involves *fwd* and counting leaf nodes up to a position in P), then computing the native function Ψ of the RLCSA [MNSV10] (or the inverse of LF in the FM-index [FMMN07]) for both extremes of the interval, then maping them back to suffix tree leaves (which requires finding the lth leaf in the tree), and finally computing an LCA operation. The GCST requires 60–300 μs for operation *sLink*, whereas Sada needs only 2–5 μs, profiting from its faster tree operations.

On the structures that do not use explicit tree topologies, the node identifier is directly the suffix array interval, and thus all what is needed is to compute Ψ on both extremes of the interval and then an LCA operation on the resulting positions. In the case of NPR-Repet and NPR, the time for LCA dominates the others: 0.6–40 *ms* and 0.2–1 *ms* (40–100 μs on influenza and einstein), respectively. In the

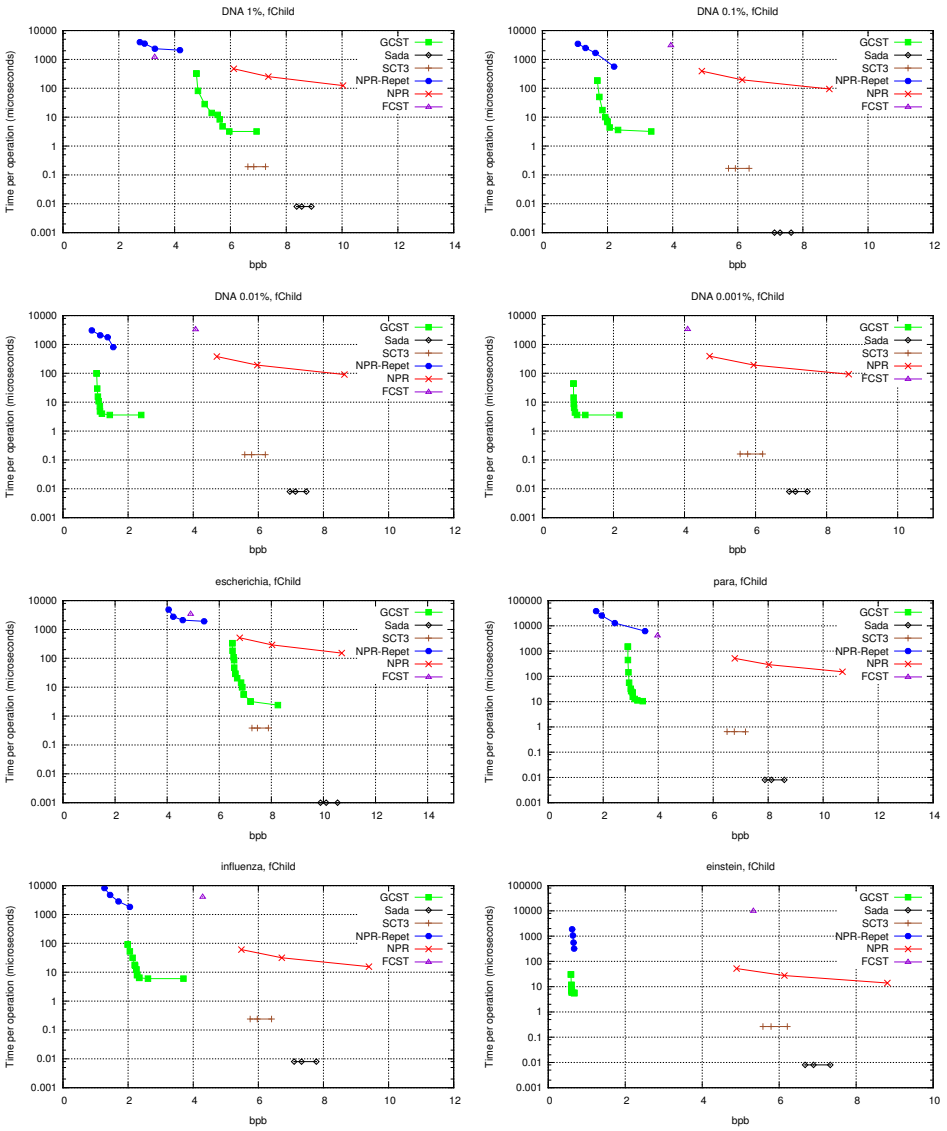


Figure 13.2: Space-time tradeoffs for operation *fChild*.

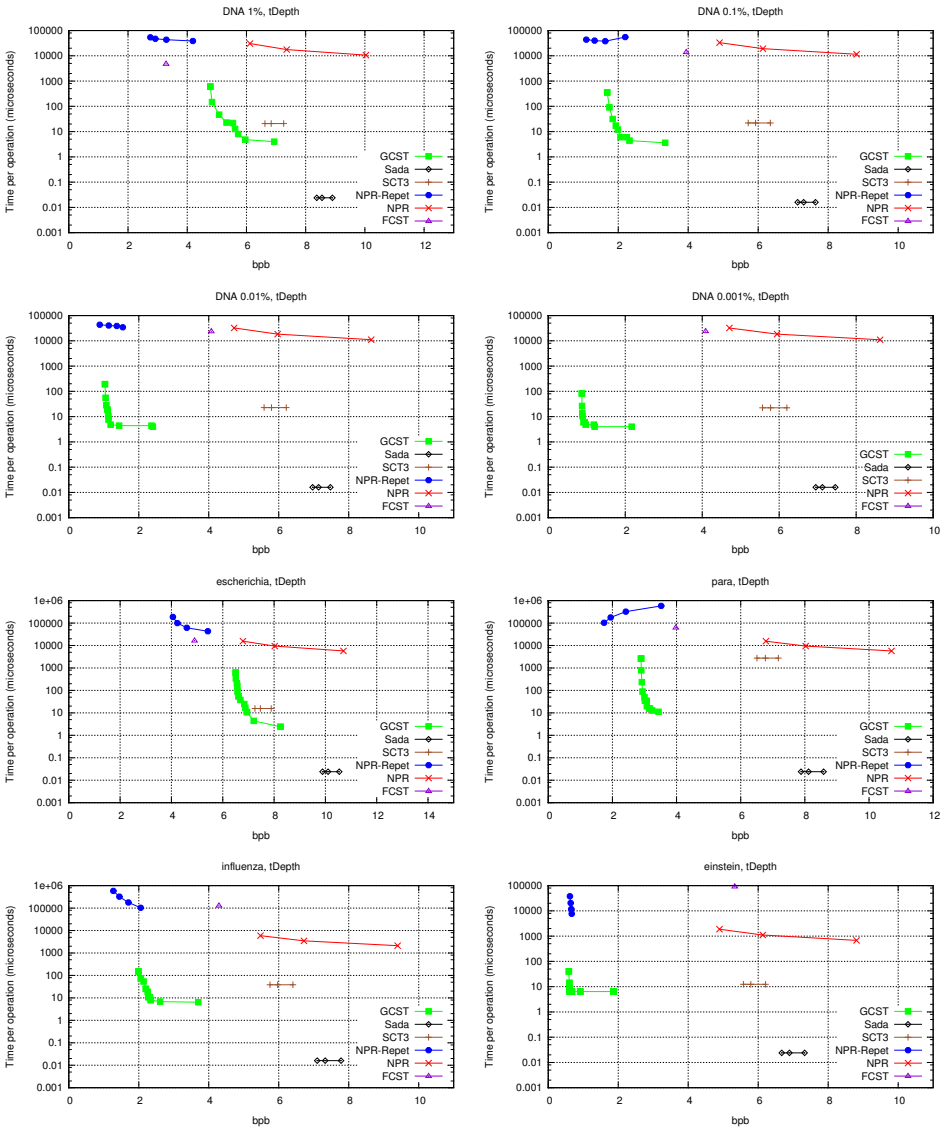


Figure 13.3: Space-time tradeoffs for operation *tDepth*.

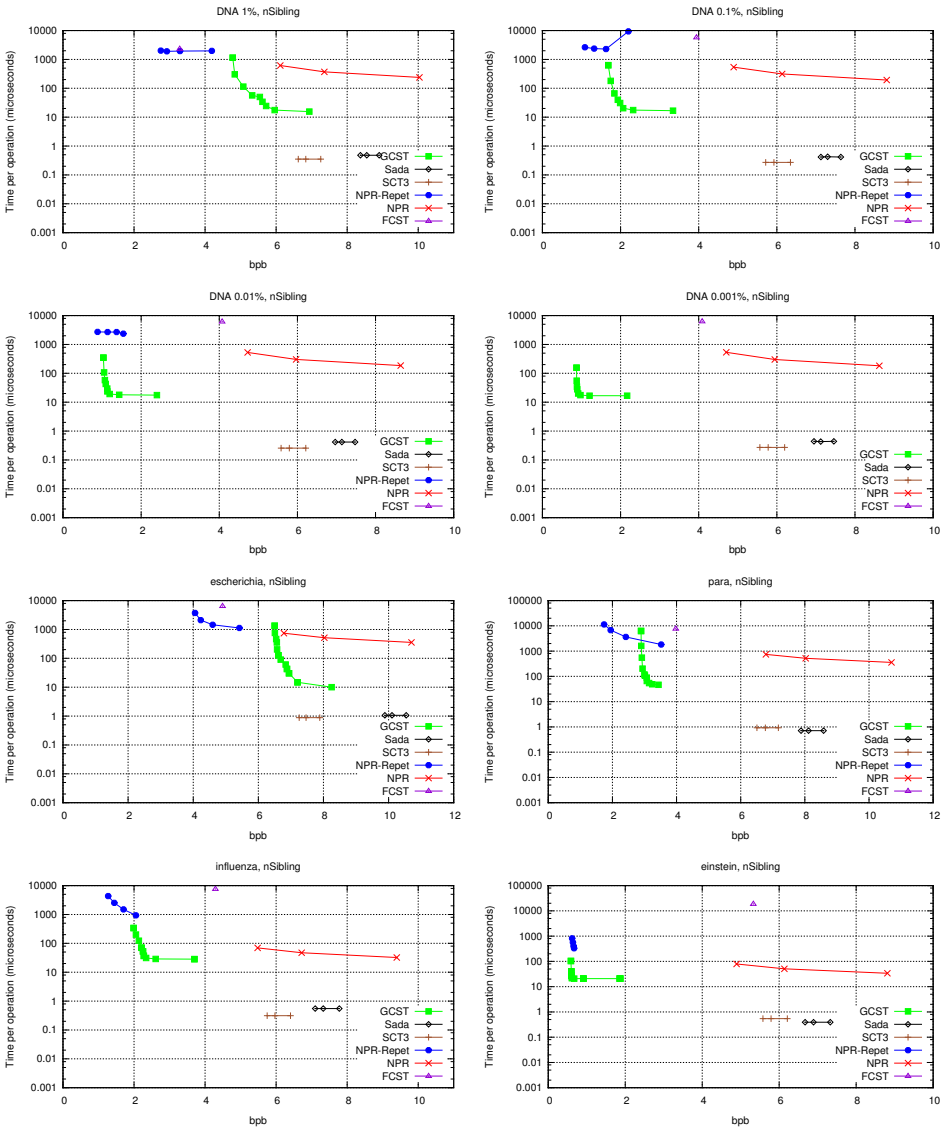


Figure 13.4: Space-time tradeoffs for operation *nSibling*.

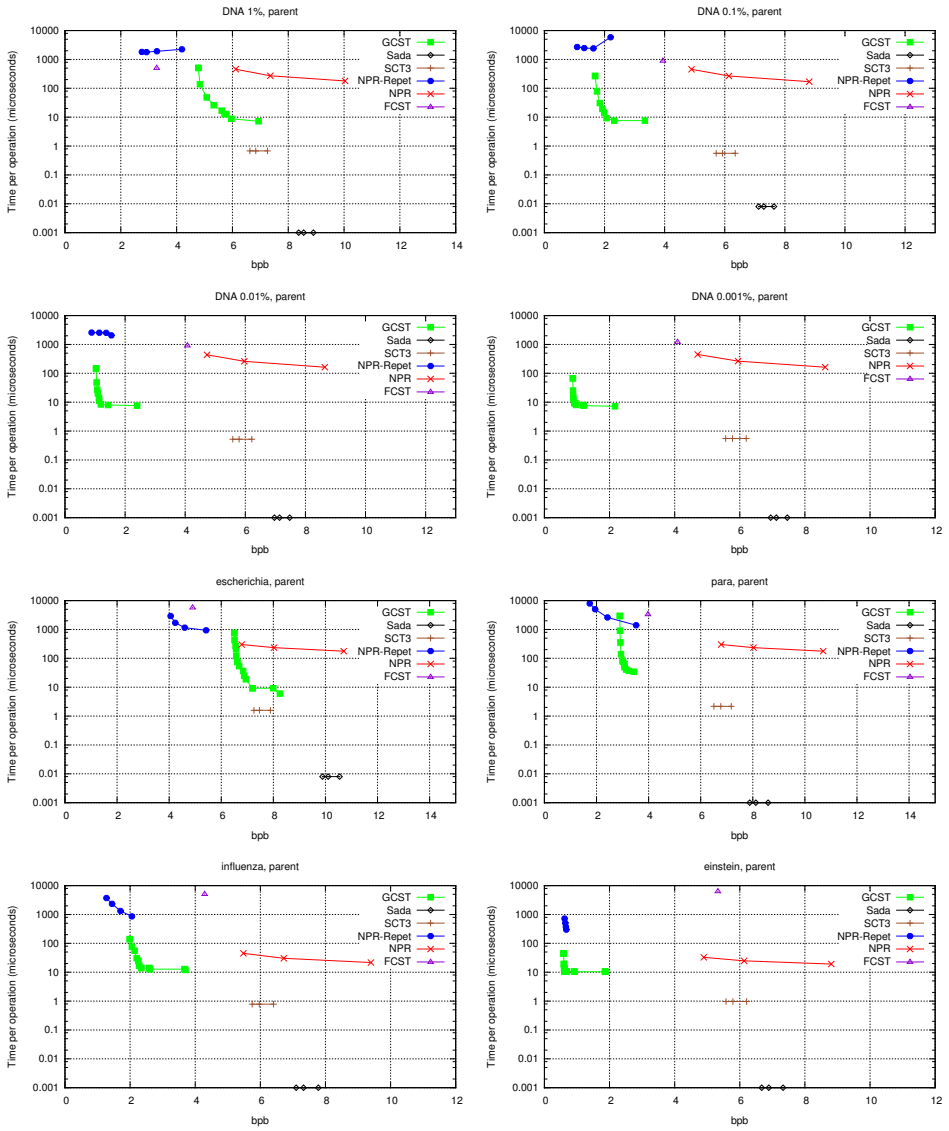


Figure 13.5: Space-time tradeoffs for operation *parent*.

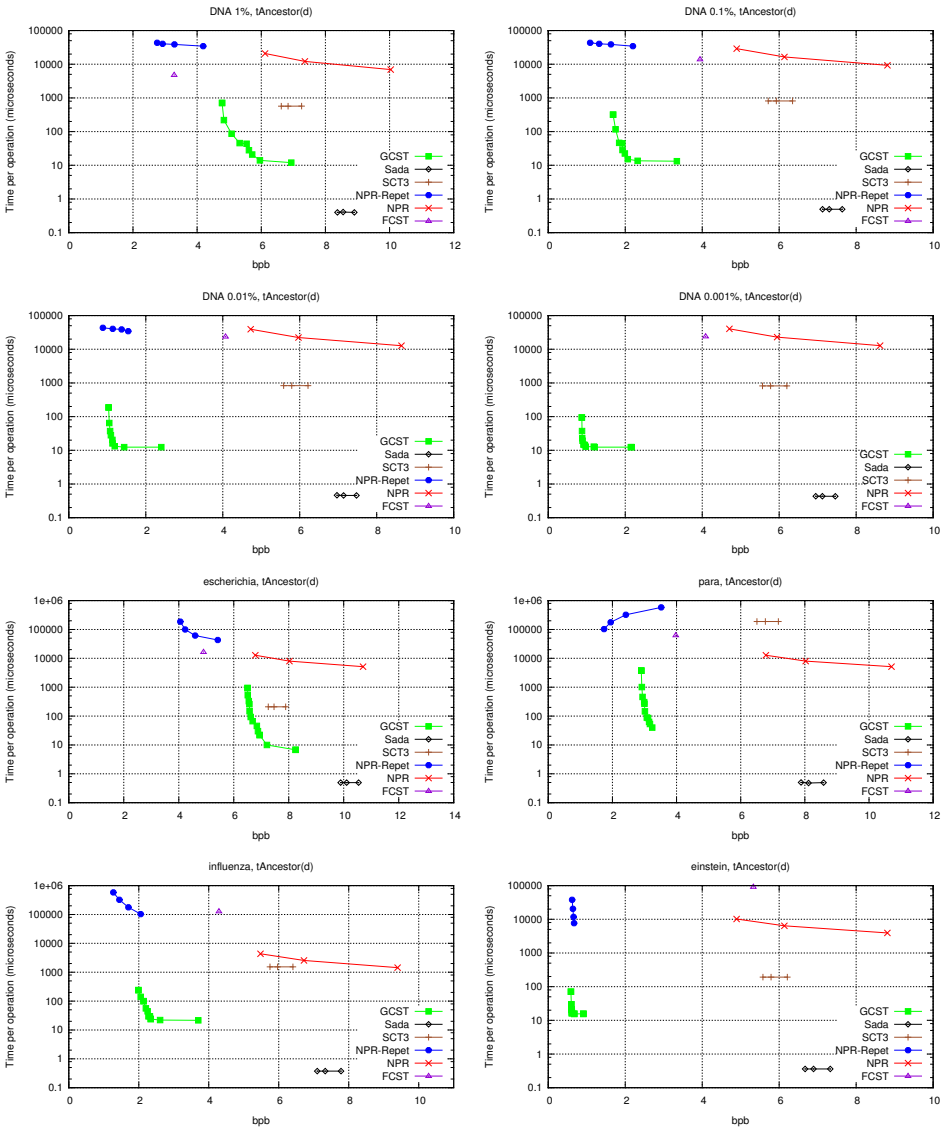


Figure 13.6: Space-time tradeoffs for operation *tAncestor*.

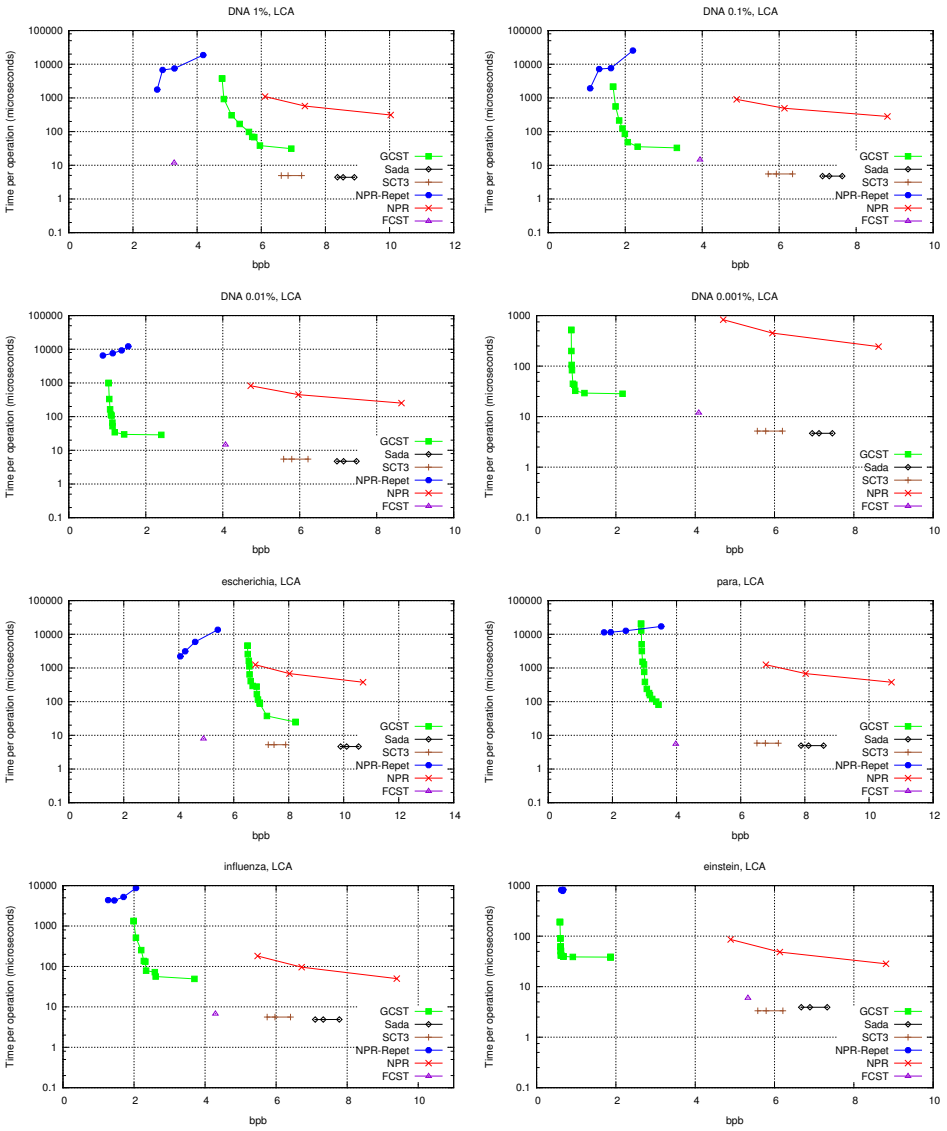


Figure 13.7: Space-time tradeoffs for operation *LCA*.

case of the FCST, operation *LCA* is fast but the other operations are not so much, driving the time to 0.2–1.5 *ms*. SCT3 takes 2–5 μs, being only slower than Sada.

Operation *sDepth*

This operation computes the string depth of a node, and is crucial for other suffix tree operations. In the `GCST` and Sada it requires mapping the second child of the node to the CSA (and thus it involves the corresponding *fwd* and leaf counting operation), whereas in NPR, NPR-Repet and SCT3 it requires a *RMQ* operation. Then, both kinds of structures must accesses the *LCP* data, which implies using the bitvector H plus the locating functionality of the `RLCSA` or FM-index. Therefore, this is the first operation where the text sampling of the suffix array plays a role in the time performance, actually dominating the overall time in various cases. The FCST, instead, implements *sDepth* as a core operation.

The operation takes 50–100 μs on the `GCST`, 60–1000 μs on NPR-Repet, and 60–100 μs on NPR (6–10 μs on `influenza` and `einstein`). Sada and SCT3 require 10–100 μs, the tradeoff being also dominated by the text sampling. The FCST takes 0.7–3 *ms*.

Operation *LAQs*

This finds the ancestor of the node with the given string depth (we test with a depth chosen at random between 1 and the node string depth). On `GCST` and Sada, which can compute *tAncestor* fast, this operation can be carried out via a binary search on *sDepth* using *tAncestor*. Thus it is computed in 250–700 μs on the `GCST`, and in 50–300 μs on Sada.

On the SCT3 and FCST, the operation must be computed via successive *parent* operation and measuring *sDepth* at each node. Therefore, it is more expensive: 0.3–3 *ms* on SCT3 and 75–300 *ms* on FCST.

Instead, this operation is almost native on NPR and NPR-Repet [ACN13], since they use on the *LCP* array a structure similar to the one we use on the excess of the parentheses. However, it still needs to compute some *sDepth* values on unsampled blocks of the *LCP* array, and this cost dominates. NPR takes 250–1000 μs (except 40–200 on `influenza` and `einstein`) and NPR-Repet takes 1–10 *ms*.

Operation *letter*

This is a simple operation exclusive of suffix trees. It gives the ith letter of the string represented by a node (we test $i = 4$). On the `GCST` and Sada, it requires mapping to the suffix array and computing Ψ^{i-1} on the `RLCSA` or LF^{-i} on the FM-index (this is usually faster than computing a suffix array cell). The `GCST` solves it in 4–10 μs and Sada in 0.75–1 μs.

The other CSTs use direct suffix array ranges, and thus do not need the mapping step. As a result, their time depends only on the CSA they use, and are faster than

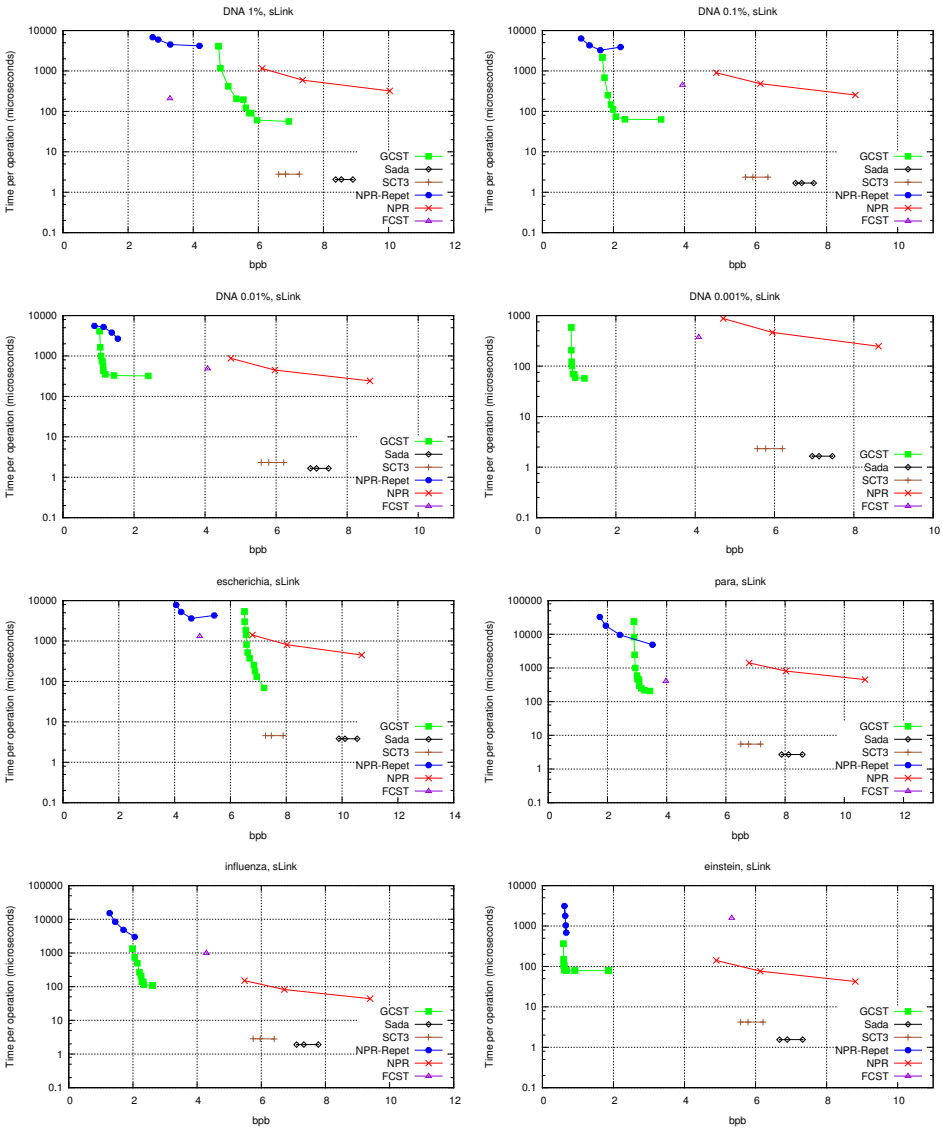


Figure 13.8: Space-time tradeoffs for operation *sLink*.

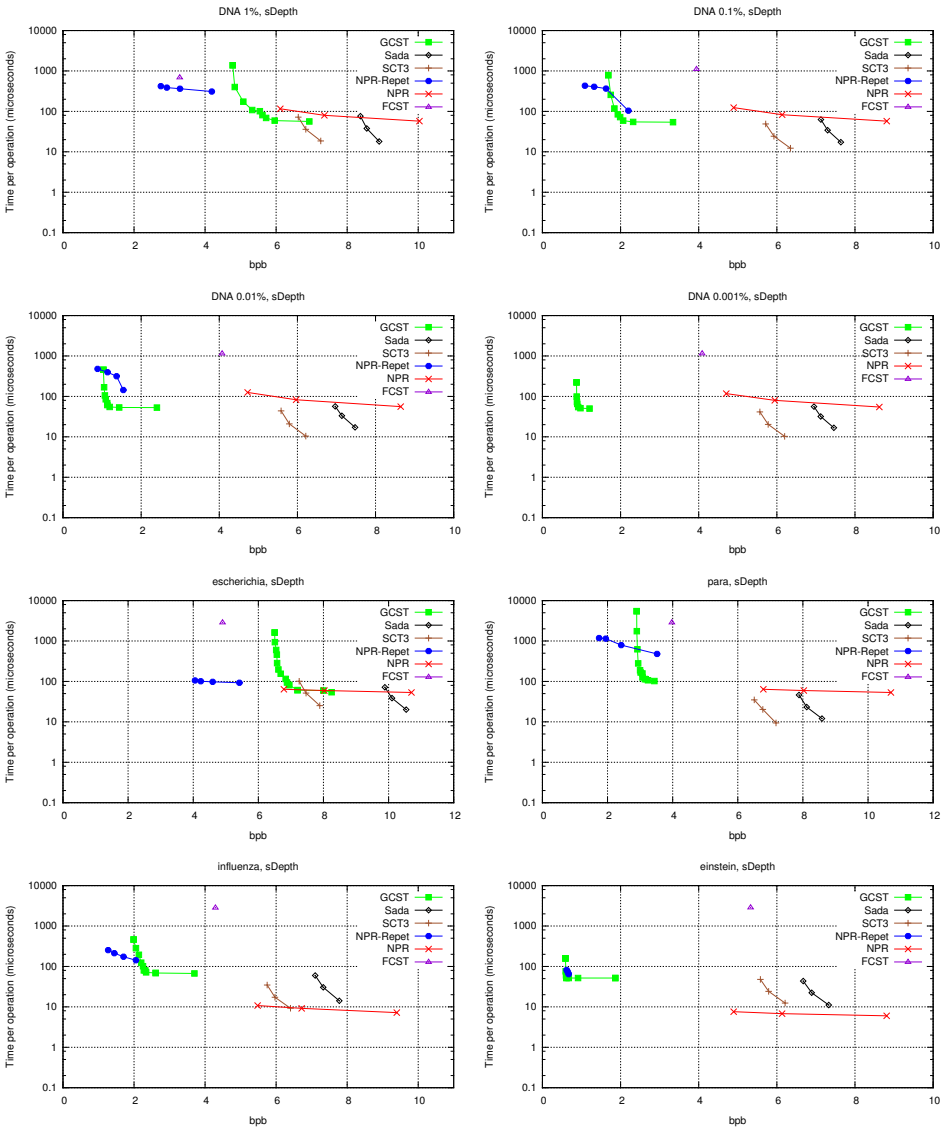


Figure 13.9: Space-time tradeoffs for operation *sDepth*.

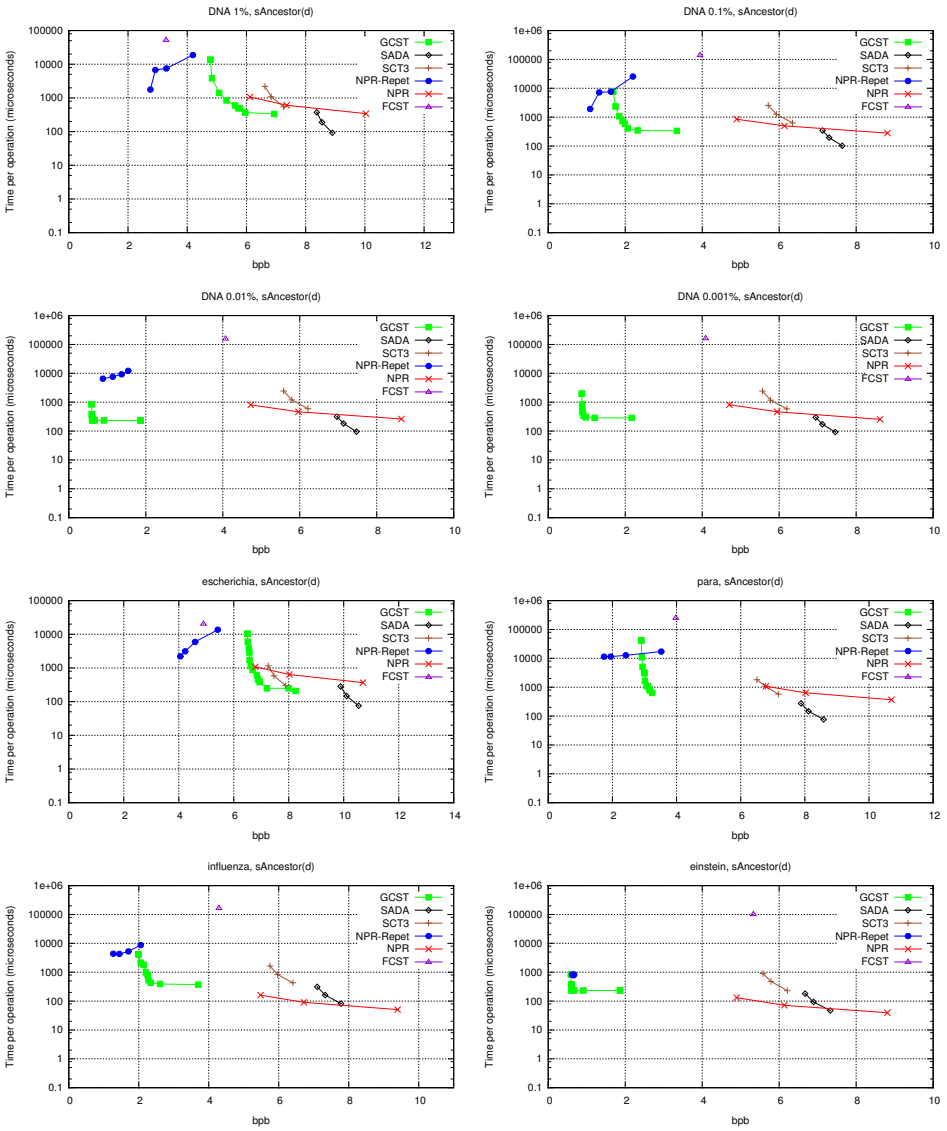


Figure 13.10: Space-time tradeoffs for operation *LAQs*.

the `GCST`, and even than Sada: NPR-Repet uses 2–4 μs, NPR uses 0.3–1 μs, and FCST takes 4 μs. The operation is not implemented in SCT3, but as it depends on the FM-index used, it should be close to 1–4 μs as well.

Operation *child*

Finally, the most complex operation is *child*, which descends to a child by an edge labeled with a given letter. It must first compute *sDepth* and then traverse linearly the children of the node, computing *letter* for each until finding the desired one.

The operation takes 0.3–1 *ms* on the `GCST`, 1–10 *ms* on NPR (but 0.3–1 *ms* on `influenza` and `einstein`), 2.5–6 *ms* on FCST, 2–30 *ms* on NPR-Repet, 70–1000 μs on Sada, and 30–200 μson SCT3 (with the exception of `DNA 0.001`, where it reaches almost 3 *ms*). Only SCT3, which has a fast implementation of *NSV* to find the successive children, and Sada, are faster than `GCST`.

Other Operations

We have left out other less important operations from the experiments: *root* is trivial in all implementations; *preorder* is similar to *tDepth* for the `GCST`, and not possible to implement in the other schemes, which do not maintain the tree topology; *pSibling* is similar to *nSibling*; *isLeaf* costs the same as *fChild* on the `GCST` and is instantaneous on the others (as they use suffix array intervals as suffix tree node identifiers, and thus leaves correspond to intervals of length 1); *ancestor* is similar to *nSibling* and is instantaneous on the others (as it involves checking containment of intervals); *subtree* is also similar to *nSibling* and cannot be implemented without the explicit topology; and *locate* depends exclusively on the performance of the underlying CSA (albeit the `GCST` requires also counting leaves). Essentially, the cost of *sDepth* is that of a *nSibling* plus a *locate* operation.

13.3.3 Discussion

Operation Times

Table 13.2 shows the ranges of the operation times of the `GCST` over all the collections tested, and how many orders of magnitude are those times lower or higher than the competitor structures[1].

The differences are particularly striking on the operations that directly refer to the tree topology: even when the `GCST` significantly compresses the topology, which entails a time cost of 1–4 orders of magnitude compared to less compressed representations (Sada), this is still 1–4 orders of magnitude faster than alternative schemes, which use the topology in implicit form (an exception is the *LCA* operation

[1] For the sake of generalization, we omitted the 10-times faster times of NPR on `influenza` and `einstein`, and a couple of excessively high times of SCT3.

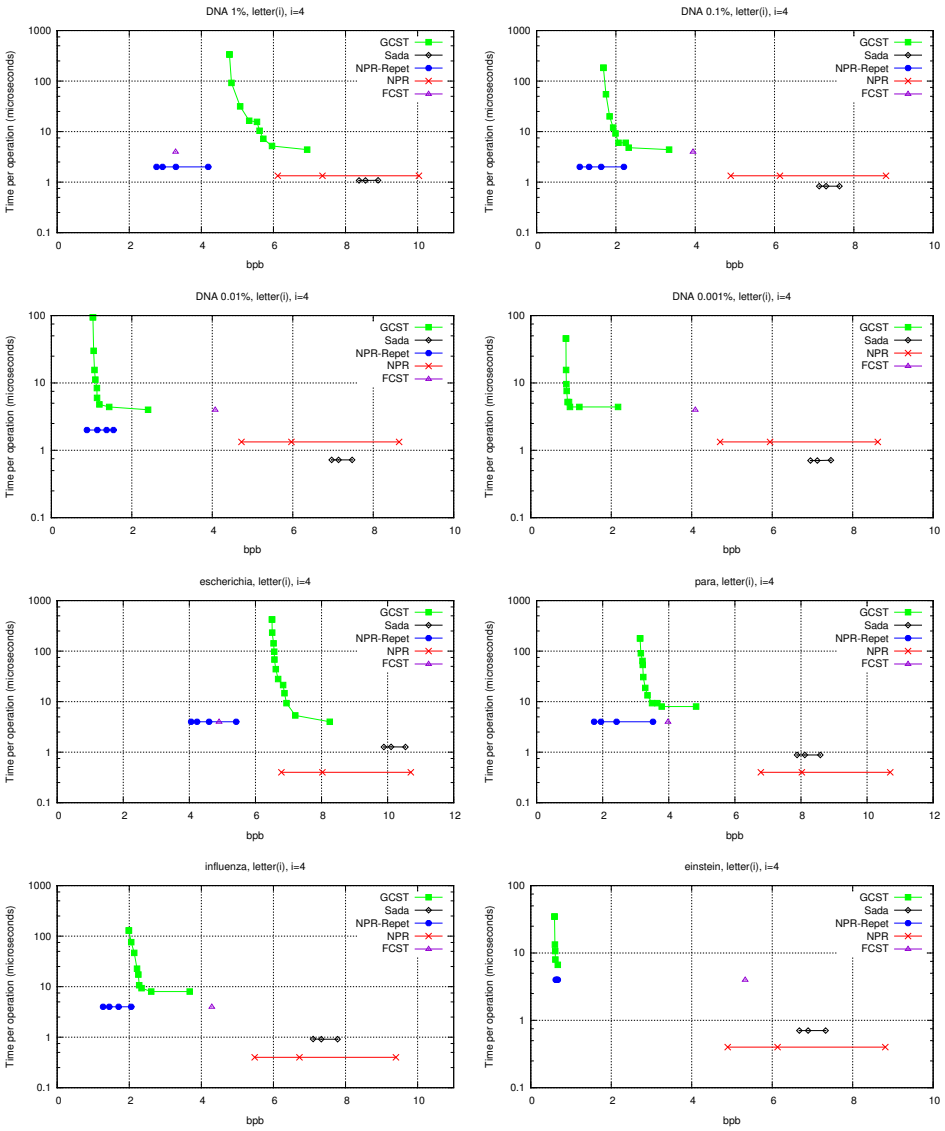


Figure 13.11: Space-time tradeoffs for operation *letter*.

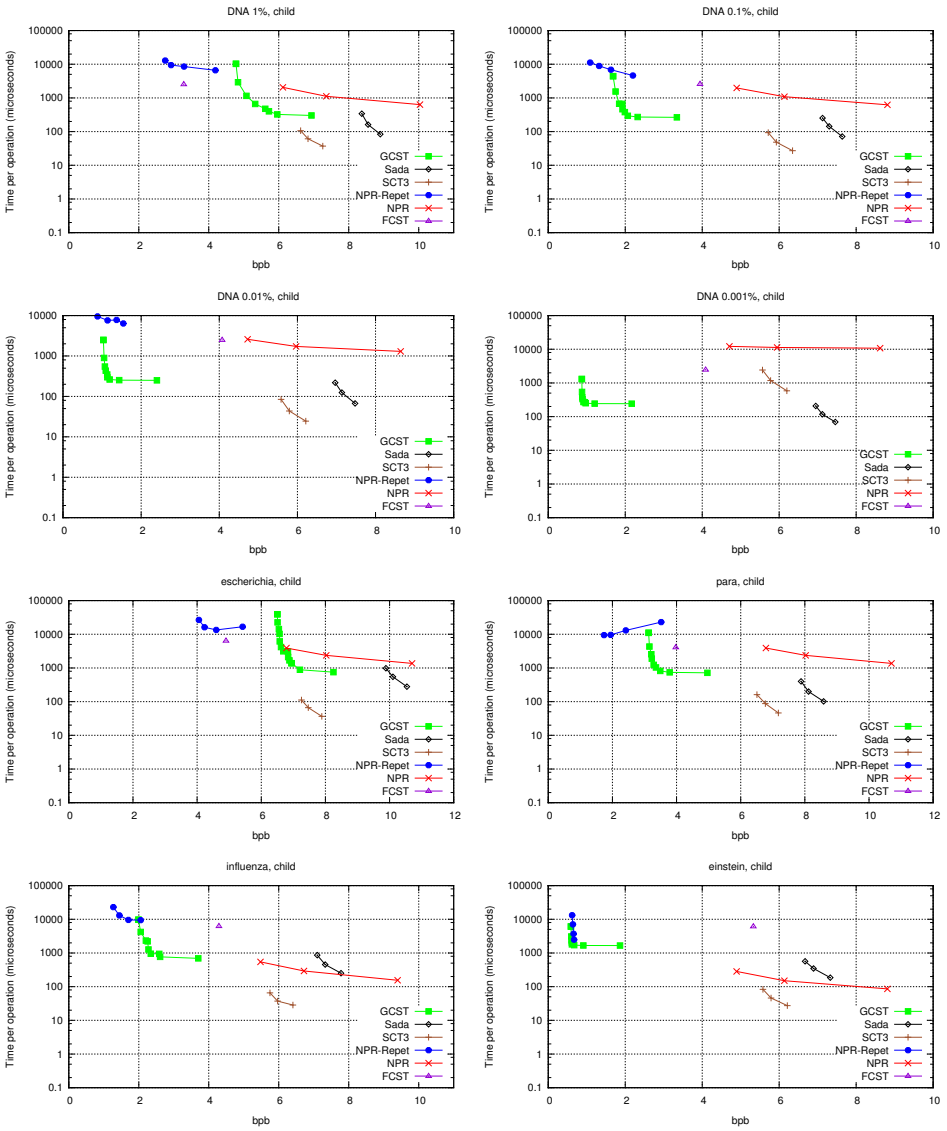


Figure 13.12: Space-time tradeoffs for operation *child*.

Operation	Time (μs)	NPR-Repet	FCST	NPR	SCT3	Sada
fChild	3–10	2–3	2–3	2	(1)	(2–3)
tDepth	5–10	2–4	3–4	2–4	0–1	(2)
nSibling	10–50	2	2	1	(1–2)	(1–2)
parent	10–40	2	2	1	(1)	(3–4)
tAncestor	10–30	3–4	3–4	2–3	1–2	(1–2)
LCA	30–100	1–2	(1)	1	(1)	(1)
sLink	60–300	1–2	0–1	0–1	(1–2)	(1–2)
sDepth	50–100	0–1	1	0	0	0
LAQs	250–700	0–1	2–3	0	0–1	0
letter	4–10	0	0	(1)		1
child	300–1000	1–2	1	0–1	(0–2)	(0–1)

Table 13.2: Operation time ranges for the `GCST` and orders of magnitude of difference with alternative CSTs (the other structure is slower by that order, unless the number is in parentheses, in which case it is faster by that order). The space increases left to right, in general terms.

on the FCST, which is very fast). On SCT3, which uses speedup structures that are alternatives to the topology, the differences in speed are up to 2 orders of magnitude in either direction, depending on the operation.

The difference decreases to 0–2 orders of magnitude on the operations that involve interaction with the CSA, as this usage is common to all the CSTs and encompasses a significant part of the total time. The general trend, within these lower gaps, is maintained: the `GCST` is faster than NPR-Repet, FCST and NPR, the comparison is mixed with SCT3, and Sada is faster. Exceptions are *tAncestor* on FCST, which is 2–3 orders of magnitude slower than `GCST`, and *letter*, which is faster on NPR than on `GCST`, and slower on Sada than on `GCST`.

The effect can also be seen on the absolute operation times. While the `GCST` uses 5–50 μs on direct tree operations (except on the *LCA*, which is by far the most complex one), the times raise to the range 50–1000 μs on the more complex operations that interact with the CSA. Incidentally, the *LCA* is the only operation where another structure within a competitive space range, the FCST, is faster than the `GCST` (other less important ones would be *isLeaf* and *ancestor*).

Times on a Complex Process

While the operation-wise comparison gives us a fine-grained picture of the performance differences, it may be difficult to determine how will the time differences look along a whole process formed by various operations of different kinds. To give a significant example of the differences between `GCST` and its fastest competitor, Sada, on a real-life problem, we choose a paradigmatic example of suffix tree functionality: find the maximal substrings of a new string $S[1,m]$ that are also substrings of T.

The algorithm is as follows: We descend by the suffix tree with the symbols of $S[1,i]$ until descending further is not possible. Then $S[1,i]$ is a maximal substring. Then we take the suffix link, corresponding to $S[2,i]$, and try to descend further. If this is still not possible, we keep traversing suffix links until we reach a node representing $S[j,i]$ from where it is possible to descend, until the node representing $S[j,i']$. Then $S[j,i']$ is the second maximal substring, and so on. The total process requires $O(m)$ operations *child* and *sLink*, which are among the most important ones on suffix trees.

The process, however, is complicated by the fact that the involved suffix tree nodes may not be explicit. Those *virtual* nodes are written as (v,ℓ), meaning the ℓth child along the unary path that descends from v in the suffix trie ($\ell = 0$ for explicit nodes). To take the suffix link of a virtual node, we can take the suffix link $v' = \mathit{sLink}(v)$ and descend up to ℓ times from v' (as there may be some intermediate explicit nodes below v' before reaching the suffix link of (v,ℓ)). This amortizes to $O(m)$ operations, but it makes repeated use of *child*, which is one of the slowest operation for all CSTs (around 1 *ms* in the `GCST` and Sada). Instead, we take advantage of the faster *LAQs* operation (around 300 μs in both CSTs) and proceed otherwise: we take the explicit descendant u of (v,ℓ), compute $u' = \mathit{sLink}(u)$, and finally the desired node is $\mathit{LAQs}(u', d)$, where $d = \mathit{sDepth}(u) - \mathit{sDepth}(v) - \ell$ (operation $\mathit{sDepth}(u)$ takes less than 100 μs in both CSTs, whereas $\mathit{sDepth}(v)$ is known from the previous operation). This of course takes also $O(m)$ operations, which require less than half the time of the classical alternative.

Overall, the operations *child*, *sLink*, *LAQs* and *sDepth* are involved. These in turn make use of the primitives *fwd*, counting leaves up to a position, computing Ψ, finding a given leaf, *LCA*, computing *LCP* values (and thus locating a suffix array position, which depends on the sampling, and accessing bitvector H), binary searching on *tAncestor* (which makes use of *bwd*), traverse the children of a node, and computing *letter* (which again applies *fwd* and Ψ). Therefore the test on the suffix tree operations is rather comprehensive.

We take `influenza` as our text collection. For the string S, we take other Influenza sequences[2]. We take one sequence of 3000 base pairs, so the process simulates finding zones of the collection that are highly similar to a new gene. The resulting maximal intervals have lengths around 100. To consider a longer string S, we also concatenate 2 MB of those sequences (removing separators), which is the

[2]From `www.cs.helsinki.fi/group/suds/rlcsa/data/influenza.gz`

approximate length of a genome in our collection. For GCST, we take the sweet point at $y = 1$ and $z = 2^{10}$, and use the RLCSA samples of 32 for Ψ and 64, 128, and 256 for the text sampling. The RLCSA sampling used for Sada is the same.

Figure 13.13 shows the results. The differences between GCST and Sada are more noticeable as the RLCSA sampling is denser, since the other operations take more relevance. However, for reasonable sampling values, the differences in time are below a factor of 3. More importantly, the GCST can reach the same time performance of Sada while using much less space. For example, for the short string S, the GCST speeds up to 300 μs per symbol while using around 3 bps, so the whole process takes less than a second. If allowed to use that time, however, Sada still cannot use less than 7 bps. The differences are higher on the longer S, where the GCST can process the whole genome in around 15 minutes using 3 bps.

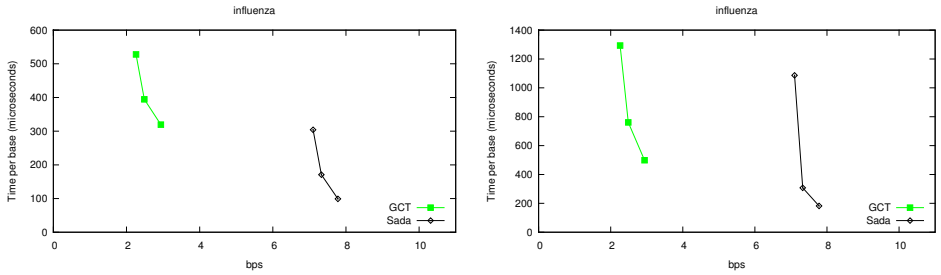


Figure 13.13: Space-time tradeoffs for finding the maximal substrings of $S[1, m]$ that appear in the collection. On the left, $m = 3000$, on the right, $m = 2$MB.

Evolution of Space Usage

The times obtained on larger CSTs are, of course, lower. For example, the large NPR structure [ACN13] reaches 1 μs in most operations (except 10 μs on LCA and 100 μs on *child*). However, as explained, it would be particularly large on repetitive collections. Other structures implemented in SDSL [Gog11] are larger than SCT3 and faster. In particular, the original structure by Sadakane [Sad07a], as implemented in SDSL, should use around 9–10 bps with a sampling sufficiently dense to solve all the operations in 1–10 μs (and the direct tree operations in nanoseconds, as shown in our experiments).

Those general-purpose suffix trees will maintain their bps value approximately stable as the collection grows, whereas those oriented to repetitiveness like NPR-Repet and GCST are likely to keep reducing their bps. Figure 13.14 shows how the space of the CSTs considered evolves as repetitiveness increases on the synthetic DNA collections (where repetitiveness can be precisely measured). As discussed,

the FCST is the only one that worsens with repetitiveness. With mutation rates of 1% the GCST uses less than 6 bps. Although NPR-Repet and FCST use less space, the GCST is orders of magnitude faster than them. When the mutation rate drops to 0.1%, the GCST becomes smaller and way faster than NPR and FCST, and the difference widens as the mutation rate drops. Only NPR-Repet stays more space-efficient than the GCST, but the difference decreases fast with repetitiveness (it would probably almost disappear at 0.001%). Still, the GCST is several orders of magnitude faster than NPR-Repet. SCT3 and Sada are faster than GCST for many operations, but already for a mutation rate of 0.1% they use more than 3 times the space of GCST. This raises to more than 5 times for the mutation rate 0.01%.

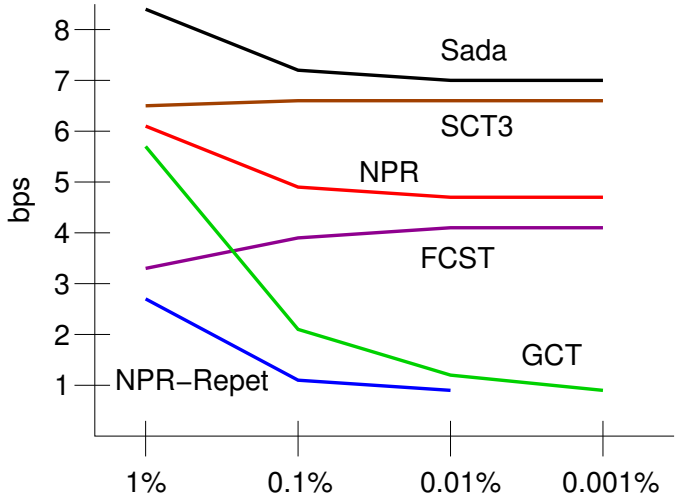


Figure 13.14: Approximate space figures for the different CSTs as repetitiveness increases on the synthetic collections. For the space of GCST we take the sweet point, whereas for the others we show their minimum space in the plots.

Construction

Finally, let us consider construction times. Figure 13.15 shows the cost to build the GCST separated by collections (with the average at the end) and by subprocess in the construction: from bottom to top, the construction of the RLCSA, the construction of the LCP (bitvector H), the generation of the parentheses topology, and finally its Repair-compression. This last step takes a significant portion of the total construction time, and renders the GCST 2–5 times slower to build than the classical CSTs (except the FCST, which builds more than 10 times slower). Analogously, the

RePair-compression of the differential *LCP* array is what makes NPR-Repet equally slow to build. Still, the construction of the GCST for a human genome should take less than 2.5 hours, which seems acceptable.

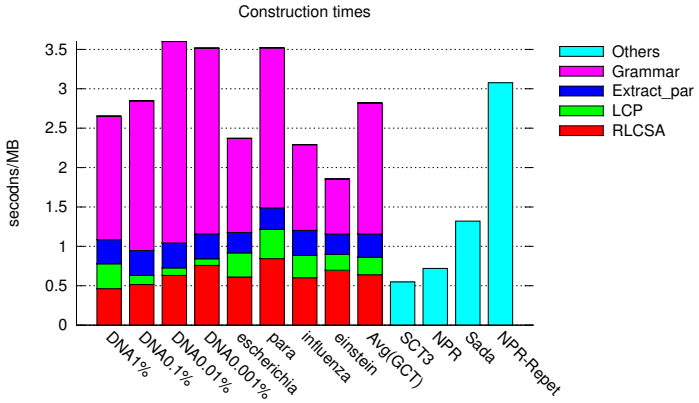


Figure 13.15: Construction times, in seconds per MB, for the different indexes. The GCST is separated into the different subprocesses. The time of FCST is over 30 seconds per MB.

13.4 Discussion

We have shown the GCST allows representing explicitly the topology of compressed suffix trees within very little space on repetitive sequence collections, using less than 2 bits per symbol (bps) for synthetic mutation rates under 0.1%, and within 2–3 bps on actual repetitive DNA sequence collections. Thanks to the explicit representation of the topology, the GCST is fast, solving the query and navigation operations in the range of the microseconds. This is generally several orders of magnitude faster than alternative representations that achieve competitive space [AN12] (and not so competitive [RNO11, CN10a]) by managing the topology in implicit form. From those alternatives, only one [AN12, ACN13] is actually smaller than the GCST (with the difference shrinking as repetitiveness increases), but its operation times are in the range of milliseconds. Only larger CSTs [Sad07a, OFG10, CN10a, Gog11], which on these collections would use 3–7 times the space of the GCST, operate within microseconds and can be faster than the GCST (sometimes orders of magnitude faster) for most operations.

Some important challenge remain open, though. We have been so successful in compressing the various components of the suffix tree, that the sampling of the

`RLCSA` [MNSV10], which is not compressed, starts to dominate. For example, on DNA with 0.001% of mutations, the whole `GCST` uses 0.9 bps, from where 0.6 bps owe to the `RLCSA` sampling. Finding ways to compress this sampling when the collection is repetitive is becoming a pressing issue. Some recent and promising results in this aspect point to new research directions [NPC+13].

Part IV

Thesis Summary

Chapter 14

Conclusions and Future Work

In this chapter we present our conclusions about the work carried out to complete this thesis, addressing also some future research lines that would be interesting to explore in more detail.

14.1 Conclusions

In this thesis we have focused mainly in two kinds of databases: statistically-compressible and highly repetitive databases. We have proposed compressed data structures that address problems of interest regarding each of these two areas, which may open the door to new developments and applications.

Regarding stastistically-compressible databases, we have focused on sequences with large alphabets, that is, in which the number of different symbols is large. This implies some challenges that were not previously faced, resulting in novel compressed data structures with applications such as grid representations, word-based self-indexes, Web graphs, and others.

Our first contribution addressed the problem of space efficiently representing prefix free codes in case of large alphabets. We experimentally evaluate the space and compression efficiency of optimal and suboptimal codes, showing that various combinations offer useful tradeoffs that can be applied in highly space-restricted environments.

Our second contribution consist of a new `rank` and `select` compressed data structure especially suited for sequences with large alphabets. It obtains zero-order compression and it outperforms all wavelet tree based data structures.

In the second part of the thesis, we have focused on highly repetitive databases, presenting several data structures that deal with typical problems that appear in

these kinds of contexts. Note that tackling these kinds of problems is of major interest since not many compressed data structures are available to deal with highly repetitive data. Classical statistical-compressed representations are known to be useless at capturing repetitiveness.

Our third contribution deals with solving `rank` and `select` queries on highly repetitive databases. To do so, we propose a new compressed data structure based on grammar compression that obtains the best space performance among all the state of the art, especially outperforming those specifically designed for these kinds of scenarios.

Our fourth contribution also deals with solving `rank` and `select` queries on highly repetitive databases. In this case, we obtain the first *LZ77* space bounded compressed data structure that solves this kind of queries in $O(1)$ time. In practice, this data structure almost obtains the same time performance than statistically-based compressed data structures, but using many times less space.

Our fifth contribution faces another interesting problem related to highly repetitive databases. We proposed the first grammar-compressed tree topology, which is of interest in case we have trees with many isomorphic subtrees. The results we obtained have no precedent, and have applications in many contexts like compressed suffix trees.

Additionally, we present several applications to point grids, inverted indexes, self-indexes, XPath query systems, and compressed suffix trees, in which these data structures apply. Concretely, we present new `count` and `report` query algorithms for the wavelet matrix that outperforms those based on wavelet trees; we present an experimental evaluation involving inverted indexes simulated with `rank` and `select` data structures and classical inverted indexes; we present the most compact self-index up to date by using our `rank` and `select` data structures for highly repetitive scenarios; we present the most space efficient XPath system; and we also present one of the most space and time efficient compressed suffix trees for highly repetitive collections based on our grammar compressed tree topologies proposal. In general, we obtain large space savings at the cost of worse time performance. This may be a convenient tradeoff in many scenarios.

14.2 Future work

In this section do not focus on the particular research lines derived from each of the proposals as that was already done at the end of each corresponding chapters. Instead, we will try to summarize the future research lines that would be interesting to continue in the following years. As claimed on the previous section and in the introduction, the field of highly repetitive databases is relatively new, and there are still more problems than solutions. As a result, many classical compressed representations are still being used instead of compressed data structures. For instance, file versioning systems still use the classical approach of storing the deltas

between versions, which is very space efficient but which does not offer practically anything but space-efficiency and document recovery. However, by using compressed data structures, we could manage to have all the versions available while offering richer functionalities beyond document recovery. This would be interesting, for instance, to extract metrics in software repositories by having access to all versions efficiently, to carry out complex queries on the compressed data.

To keep pushing on this direction, and in order to convince the community that this new approach is better than what they already have, new complete systems have to be built and tested against classical approaches. Besides, we still have to provide more space and time efficient compressed data structures, especially faster algorithms.

Another aspect that deserves more attention are the construction algorithms. It is fundamental for these algorithms to run fast in order to be attractive and applicable in any context, and most of them are not. But not only the construction speed is a limitation. For instance, the best implementation of RePair construction algorithm is only able to process relatively small datasets. Being able to handle larger datasets becomes fundamental to apply many of the proposals presented in this thesis to actual problems.

Another avenue that seems actually very interesting is to use the Block Tree construction algorithm to exploit repetitiveness in other domains like trees or graphs. This will have applications to the so demanded graph databases, among others.

Additionally, how to turn these data structures into dynamic approaches is actually a challenge. There has been little effort to study this kind of scenario, but providing compressed data structures able to handle some degree of dynamism would also be a tremendous step forward in this field.

Not less important is the fact that, for highly repetitive scenarios, we are not aware of lower bounds that tell us how far can we go in terms of compression performance. This is mainly because of the relatively recent nature of these kinds of databases.

Appendices

Appendix A

Publications and Other Research Results

This chapter summarizes the research publications as well as the research visits directly related with this thesis.

Publications

Journals

- F. Claude, G. Navarro, and A. Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.
- T. Gagie, G. Navarro, Y. Nekrich, and A. Ordóñez. Efficient and compact representations of prefix codes. *IEEE Transactions on Information Theory*, 61(9):4999–5011, 2015.
- G. Navarro, A. Ordóñez. Faster Compressed Suffix Trees for Repetitive Text Collections. *ACM Journal of Experimental Algorithmics*, (To appear).
- A. Ordóñez., G. Navarro, N. R. Brisaboa Grammar Compressed Sequences with Rank/Select Support. *Information Systems*, (Submitted).

International Conferences

- Belazzougui, D.; Gagie, T.; Gawrychowski, P.; Kärkkäinen, J.; Ordóñez, A.; Puglisi, S.J.; Tabei, Y.: Queries on LZ-Bounded Encodings. In *Proc. of the 2015 Data Compression Conference (DCC 2015)*, IEEE Computer Society, Salt Lake City, Utah (United States), 2015, pp. 83-92.

- Navarro, G.; Ordóñez, A.: Grammar Compressed Sequences with Rank/Select Support. In *Proc. of the* 21*th International Symposium on String Processing and Information Retrieval (SPIRE 2014)* - LNCS 8799, Springer , Ouro Preto (Brazil), 2014, pp. 31-44.
- Navarro, G.; Ordóñez, A.: Faster Compressed Suffix Trees for Repetitive Text Collections. In *Proc.* 13*th International Symposium on Experimental Algorithms (SEA 2014)* - LNCS 8504, Springer, Copenhagen (Denmark), 2014, pp. 424-435.
- Navarro, G.; Ordóñez, A.: Compressing Huffman Models on large Alphabets, In *Proc. of the 2013 Data Compression Conference (DCC 2013)*, IEEE Computer Society, Snowbird, Utah (United States), 2013, pp. 381-390.
- Brisaboa, N. R.; Navarro, G.; Ordóñez, A.: Smaller Self-Indexes for Natural Language. In *Proc. of the* 19*th International Symposium on String Processing and Information Retrieval (SPIRE 2012)* - LNCS 7608, Springer, Cartagena de Indias (Colombia), 2012, pp. 372-378.

International research visits

- *March, 2012 - June, 2012.* Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile, with Gonzalo Navarro.
- *June, 2013 - July, 2013.* Department of Computer Science, University of Oxford, Oxford, UK, with Sebastian Maneth and Michael Benedikt.
- *November, 2013.* Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile, with Gonzalo Navarro.
- *March, 2014.* Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile, with Gonzalo Navarro.
- *September, 2014 - October 2014.* Department of Computer Science, University of Helsinki, Helsinki, Finland, with Travis Gagie, Veli Mäkinen, and Simon J. Puglisi.
- *March, 2015.* Departamento de Ciencias de la Computación, Universidad de Chile, Santiago, Chile, with Gonzalo Navarro.

Appendix B

Resumen del Trabajo Realizado

La cantidad de información almacenada a lo largo de las últimas décadas ha crecido exponencialmente, y parece que las cosas van a cambiar poco en un futuro reciente. Las bases de datos están siendo inundadas con cantidades ingentes de datos provenientes de fuentes muy diversas y con propiedades igual de diversas. Por ejemplo, existen bases de datos textuales que contienen el resultado de procesos de *crawling* Web o digitalización de documentos que han crecido más rápido que la capacidad de muchas organizaciones para almacenarlas. El problema es aún más grande en aplicaciones tales como los sistemas de control de versiones o los repositorios de software en los que queremos acceder el histórico o las versiones de un documento, con el tremendo impacto en términos de espacio que esto implica ya que debemos almacenar todas las versiones de cada documento. También tenemos bases de datos que contienen secuencias de ADN en las cuales almacenamos los genomas de muchos individuos de la misma especie. Esto supone todo un reto ya que este tipo de bases de datos han crecido al mismo ritmo que el coste para obtenerlas ha decrecido, es decir, más rápido que la capacidad del hardware. Almacenar y hacer consultas sobre la estructura de una red social o un grafo Web es también un reto debido a la cantidad de nodos y conexiones presentes en este tipo de redes. Los Sistemas de Información Geográfica son otro ejemplo de aplicaciones en los que la cantidad de datos generados es masiva.

Sin embargo, no siempre el incremento en la cantidad de datos a almacenar es el problema ya que la otra cara de la moneda es la capacidad de almacenamiento. Con el uso masivo de dispositivos móviles (*smartphones, tablets* o sensores), es común tener que procesar grandes cantidades de datos en entornos con capacidades, tanto de procesamiento como de almacenamiento, muy restringidas. Este tipo de dispositivos deben ser capaces de procesar muchísimos datos en un entorno altamente limitado y

sin la posibilidad de recurrir a mecanismos de almacenamiento adicionales. Además, en entornos móviles transmitir datos en formato comprimido es fundamental para ahorrar ancho de banda y batería.

Afortunadamente, la mayoría de los datos que se generan y manejan en cualquiera de estos escenarios es de naturaleza no aleatoria. Generalmente, dichos datos se pueden modelar a través de modelos estadísticos o al menos son, en algún sentido, más o menos predecibles. Y predecible en teoría de la información es sinónimo de compresible. Desde la publicación de la tesis de Shannon, muchas son las representaciones comprimidas que han sido diseñadas para representar datos con diferentes orígenes. Encontrar las representaciones comprimidas que permitan representar un conjunto de datos utilizando el menor espacio posible es un problema de primer orden. Y no solo por los evidentes motivos económicos que suponen necesitar menos dispositivos de almacenamiento. La compresión de datos es también fundamental para aumentar la velocidad de transmisión en redes, lo que implica una mejora de rendimiento en entornos *cluster*. También es fundamental para ahorrar energía en dispositivos móviles mediante la reducción del tamaño de los paquetes que se envían a través de redes sin hilos. Sin embargo, el principal problema de las representaciones comprimidas clásicas es que simplemente ofrecen compresión. Si quisiésemos hacer búsquedas dentro de los datos comprimidos tendríamos que descomprimir los datos y luego hacer la búsqueda sobre el resultado de la descompresión. Esto significa que en caso de necesitar funcionalidades más avanzadas, los beneficios de las representaciones comprimidas pueden llegar a desaparecer.

Con el objetivo de lidiar con este problema, la aparición de las estructuras de datos compactas ha sido un auténtico alivio. Una estructura de datos comprimida no se preocupa únicamente por el espacio sino que también se centra en las funcionalidades. Usar la mínima cantidad de espacio posible sigue siendo uno de los objetivos, aunque las estructuras de datos comprimidas van más allá permitiendo realizar operaciones directamente sobre los datos comprimidos. Esto significa que, por ejemplo, si queremos realizar una búsqueda dentro de una estructura de datos comprimida, lo podemos hacer directamente sin descomprimir antes los datos. O si queremos acceder a una porción de dichos datos, lo podemos hacer sin descomprimir toda la representación. Es decir, la estructuras de datos comprimidas permiten realizar operaciones sobre los datos comprimidos, lo cual supone un paso adelante significativo con respecto a las representaciones comprimidas por los siguientes motivos: (a) seguimos teniendo los mismos beneficios que aportan las representaciones comprimidas, (b) ahorramos mucho tiempo si evitamos tener que descomprimir toda la representación sólo para acceder a una porción de la misma, (c) la eficiencia de las búsquedas aumenta ya que no tenemos que buscar en la secuencia descomprimida, que generalmente es mucho más larga, (d) obtenemos estructuras más rápidas ya que podemos operar directamente en memoria RAM o *cache*, y (e) permiten ahorrar energía.

El problema de las estructuras de datos comprimidas es que son mucho más recientes que la representaciones comprimidas clásicas, lo que significa que, en realidad, tenemos muchísimos más problemas abiertos que soluciones disponibles. Necesitamos estructuras de datos comprimidas más eficientes, tanto en términos espaciales como de tiempo, para problemas la búsqueda indexada de texto, para tratar problemas de biología computacional, para problemas relativos a la recuperación de información, para mejorar el rendimiento en motores de búsqueda, para representar información jerárquica, para mejorar las búsquedas dentro de ficheros XML, para mejorar las comunicaciones en redes, para mejorar el rendimiento en *clusters* de computadores, para proporcionar sistemas de búsqueda en sistemas con capacidades limitadas como dispositivos móviles y sensores, para mejorar sistemas de información geográfica, y para un largo etcétera de aplicaciones.

En esta tesis nos hemos centrado en algunas de estos problemas pero teniendo muy en cuenta la naturaleza de los datos que estamos procesando, distinguiendo entre conjuntos de datos estadísticamente compresibles y conjuntos de datos altamente repetitivos.

Los compresores estadísticos aprovechan la distribución de los símbolos en un conjunto de datos para asignar códigos más cortos a los símbolos más frecuentes. Es un área de investigación muy robusta ya que ha estado activa desde hace décadas y porque se conocen multitud de cotas inferiores que nos permiten saber lo lejos que podemos llegar en términos de compresión. Sin embargo, y a pesar de su madurez, aún existe muchos problemas que no han sido resueltos o tan siquiera considerados. Hay que tener en cuenta que la compresión estadística fue inicialmente pensada para secuencias provenientes de una fuente de información infinita en la que el número de símbolos diferentes (el alfabeto) era finito. Y no sólo finito, sino que en muchos casos considerado relativamente pequeño. Sin embargo, con la introducción de las estructuras de datos comprimidas, las fuentes de información pasan a ser finitas (un documento a comprimir, por ejemplo), y el alfabeto no puede ser considerado pequeño, siendo un problema en muchos casos precisamente por esto. Por ejemplo, si consideramos un texto como una secuencia de palabras y no de letras, el tamaño del alfabeto aumenta dramáticamente. Si modelamos una grilla $n \times n$ de puntos como una secuencia de coordenadas, el tamaño del alfabeto es tan grande como la grilla. Las listas de adyacencia en sistemas de recuperación de información o grafos Web suelen presentar problemas similares. En cualquiera de estos escenarios, simplemente representar el alfabeto es un problema, y cómo lidiar con ello no ha sido considerado aún. La primera parte de la tesis se ha centrado en la propuesta de nuevas estructuras de datos comprimidas que se centran en bases de datos estadísticamente compresibles en las que el tamaño del alfabeto es grande.

Por otro lado, un conjunto de datos altamente repetitivo es aquel en el que tenemos muchas copias o quasi-copias del mismo documento. Esto ocurre, por ejemplo, en repositorios software o en sistemas de control de versiones en los que se esperan muchas versiones del mismo documento pero con sólo ligeras variaciones

entre versiones sucesivas del mismo. Las bases de datos de ADN son otro claro ejemplo ya que en ellas se almacenan muchos genomas de individuos de la misma especie, sabiéndose que dos individuos de la misma especie comparten la mayoría de su material genético, lo cual resulta en bases de datos altamente repetitivas. El problema es que los compresores estadísticos no ofrecen un buen rendimiento en conjuntos de datos repetitivos ya que no son capaces de capturar la repetitividad. Muchas son las representaciones comprimidas para conjuntos de datos altamente repetitivos, pero muy pocas la estructuras de datos compactas y aún menos las cotas inferiores que se conocen. Una explicación obvia es que las bases de datos altamente repetitivas solo han sido posibles debido al incremento exponencial en la capacidades de almacenamiento experimentado recientemente. Es decir, es una área de investigación muy joven y el número de propuestas es bastante limitado. En general, aquellas estructuras que ofrecen un buen rendimiento en teoría no suelen ser prácticas, mientras que aquellas que son prácticas suelen ser bastante ineficientes comparadas con la entropía de los datos. Esto es un impedimento tal que muchas aplicaciones que manejan bases de datos altamente repetitivas aún siguen usando representaciones comprimidas clásicas en lugar de estructuras de datos comprimidas. Esto es un problema grande ya que, por ejemplo, usando representaciones comprimidas clásicas, los sistemas de control de versiones pueden únicamente acceder a una versión determinada de un documento, mientras que poder hacer búsquedas dentro de cada versión de un documento sería mucho más útil, pero a su vez supondría un reto desde el punto de vista tecnológico. La segunda parte de esta tesis la hemos dedicado a la propuesta de estructuras de datos comprimidas y eficientes para conjuntos de datos altamente repetitivos. Estas propuestas, aparte de todos los beneficios descritos hasta el momento, podrían abrir la puerta a nuevas funcionalidades para diversas aplicaciones que no serían posibles mediante el uso de representaciones comprimidas clásicas.

B.1 Estructura de la Tesis y Contribuciones

En este punto describimos cómo hemos estructurado la tesis, así como las contribuciones específicas relacionadas con cada una de la partes. En general, y tal como se ha descrito en el apartado anterior, esta tesis consta principalmente de dos grandes bloques. El primero está completamente dedicado a estructuras de datos comprimidas para bases de datos estadísticamente compresibles que necesitan manejar alfabetos grandes, mientras que el segundo gran bloque se centra en los conjuntos de datos altamente repetitivos. Al inicio de cada bloque hemos incluido un capítulo en el que se explican los conceptos necesarios así como las estructuras conocidas en el estado del arte necesarios para entender las contribuciones que presentamos a lo largo de la tesis. Adicionalmente, en la tercera parte de la tesis hemos presentamos un conjunto de aplicaciones en las que las estructuras de datos presentadas son de gran utilidad.

Más concretamente, podemos desgranar esta tesis en una serie de contribuciones que se detallan a continuación:

1. La primera contribución de esta tesis, presentada en el capítulo 3, trata sobre cómo representar de manera eficiente código libres de prefijo tanto óptimos como sub-óptimos. Presentamos varias implementaciones de ideas previas y las evaluamos experimentalmente para mostrar que dichas estructuras no son únicamente relevantes desde un punto de vista teórico sino que son muy interesantes desde el punto de vista práctico si se implementan adecuadamente. La idea principal sobre la que se sustenta esta contribución es un trabajo previo del doctorando [BNO12] en el que se exploraba el uso de *alphabetic codes* para representar modelos grandes de manera eficiente en términos de espacio. En otro trabajo previo del doctorando [NO13], se demostró que una representación basada en permutaciones de códigos Huffman era en realidad más eficiente. Después de esto se extendió este último trabajo con una publicación previa de Gagie et al. [GNN10] que se centraba en la representación de códigos libres de prefijo sub-óptimos. El resultado ha sido finalmente publicado en la revista JCR *IEEE Transaction on Information Theory* [GNNO15].

   Mi contribución concreta se centra en la implementación, el proceso de ingeniería así como en la evaluación de las representaciones para códigos óptimos y sub-óptimos.

2. Nuestra segunda contribución, presentada en el capítulo 4 se denomina *compressed wavelet matrix* y es una representación alternativa para secuencias con alfabetos largos que retiene todas las propiedades de los *wavelet trees* comprimidos pero siendo significativamente más rápida. Esta contribución se basa en un trabajo previo [CN12] en el que se propuso la *wavelet matrix* sin comprimir. Sin embargo, resulta que los códigos ya conocidos para obtener compresión de orden cero en *wavelet trees* no funcionan cuando los aplicamos sobre *wavelet matrices* debido al reordenamiento de bits que se originan. Por lo tanto, para obtener compresión de orden cero en *wavelet matrices* hemos desarrollado una manera alternativa de asignar códigos a los símbolos basándonos en la desigualdad de Kraft que es óptima y compatible con las *wavelet matrices*. Con este nuevo algoritmo para generar los códigos, en teoría obtenemos compresión de orden cero mientras que en la práctica obtenemos una estructura de datos que es dominante tanto en términos de espacio como de tiempo sobre las demás implementaciones de *wavelet trees* sobre alfabetos grandes.

   Mi contribución han sido el nuevo algoritmo para generar códigos óptimos para las *wavelet matrices* comprimidas, la demostración de que dicho esquema es óptimo y correcto, la implementación y así como la evaluación experimental. El resultado de este trabajo ha sido publicado en la revista JCR *Information Systems* [CNO15].

3. Nuestra tercera contribución, presentada en el capítulo 6, es una estructura de datos comprimida para representar secuencias altamente repetitivas. Aplicaciones recientes necesitan representar este tipo de secuencias pero las estructuras de datos comprimidas basadas en compresión estadística se han mostrado ineficientes a la hora de capturar la repetitividad y, por tanto, a la hora de ahorrar una cantidad de espacio significativa. Nuestras propuesta consisten en dos estructuras de datos comprimidas basadas en gramáticas para representar secuencias en entornos altamente repetitivos. La primera de ellas, a la que llamamos GCC de *Grammar Compression with Counters* se centra en secuencias con alfabetos pequeños y es capaz de obtener en teoría el mismo espacio que estructuras de datos que obtienen espacio óptimo pero con la diferencia de que nuestra propuesta es también de orden práctico. Nuestra segunda propuesta consiste en la combinación del GCC con técnicas de partición del alfabeto que obtienen muy buenos resultados en la práctica cuando el alfabeto de la secuencia a representar es grande.

   Respecto a esta contribución, he sido el autor principal, incluída la idea, implementación, proceso de ingeniería y la evaluación experimental. El trabajo resultado ha sido publicado en *Proc. of the* 21^{th} *International Symposium on String Processing and Information Retrieval (SPIRE)* [NO14b] y la versión de revista ha sido enviada a la revista JCR *Information Systems* [ONB15].

4. Nuestra cuarta contribución, presentada en el capítulo 7, es la primera representación de secuencias que obtiene espacio proporcional al de un compresor *LZ77* y es capaz de resolver consultas `access`, `rank` y `select` en tiempo $O\left(\log_r \frac{n \lg \sigma}{z \lg n}\right)$ usando $O\left(\sigma z r \lg n \log_r \frac{n \lg \sigma}{z \lg n}\right)$ bits de espacio, siendo z el número de frase en un parseado *LZ77* de una secuencia $S[1, n]$ con alfabeto $\Sigma = [1, \sigma]$. Alternativamente, mostramos cómo podemos obtener tiempo $O(1)$ para `access`, `rank` y `select` si usamos $O(\sigma z n^{\epsilon})$ space ($\epsilon < 1$). Hemos denominado a esta estructura de datos como *Block Tree* y, más allá de sus propiedades teóricas también mostramos que funciona muy bien en la práctica cuando la aplicamos sobre secuencias con alfabetos pequeños.

   En este caso he estado involucrado en la concepción teórica de la estructura aunque centrándome en cómo hacer que fuese eficiente y competitiva en la práctica. Esto incluye todo el proceso de implementación, ingeniería y evaluación experimental. El trabajo resultado ha sido publicado en *Proc. of the 2015 Data Compression Conference (DCC)* [BGG$^+$15].

5. Nuestra quinta contribución, presentada en el capítulo 8, está relacionada con Los árboles altamente repetitivos. Resulta que en contextos de alta repetitividad en los que utilizamos estructuras en forma de árbol para almacenar información, y dependiendo de la aplicación, aparecen muchos isomorfismos en la topología del árbol (subárboles repetidos). Estos isomorfismos podrían ser

aprovechados por técnicas de compresión basadas en gramáticas para ahorrar espacio, aunque ninguna que sea completamente funcional y aproveches dicha repetitividad es conocida. Nuestra contribución, a la que denominamos GCT de *Grammar Compressed Tree*, es la primera implementación de una topología de árboles comprimida con gramáticas y que mantiene toda la funcionalidad de representaciones clásicas. Presentamos los algoritmos para llevar a cabo las operaciones más comunes así como una completa evaluación experimental en la que mostramos el rendimiento de esta estructura en la práctica.

Una idea similar a esta ha sido independientemente propuesta de manera muy superficial por Bille et al. [BLR$^+$11], y reciente y más detalladamente presentada por los mismos autores [BLR$^+$15]. Mientras que su objetivo es únicamente teórico, el nuestro se centra más en aspectos prácticos.

En este caso he sido el autor principal de este desarrollo, desde la idea original hasta la implementación, aspectos de ingeniería y la evaluación experimental. Los resultados derivados de esta investigación fueron inicialmente publicados en *Proc. of the 13th International Symposium on Experimental Algorithms (SEA 2014)* [NO14a]. En ese congreso, dicho trabajo fue seleccionado entre los mejores e invitado, y posteriormente aceptado [NO15], a la edición especial de la revista *ACM Journal of Experimental Algorithmics* dedicada a los mejores trabajos de esa conferencia.

6. Finalmente, en la tercera parte de esta tesis hemos explorado varias aplicaciones reales en las que las estructuras de datos comprimidas que hemos presentado son de gran utilidad. Dichas aplicaciones son las siguientes:

   (a) Hemos presentado nuevos algoritmos para realizar *orthogonal range queries* utilizando *wavelet matrices* y evaluado dichos algoritmos comparándolos con las versiones para *wavelet trees*, mostrando que nuestras propuestas mejoran sus resultados.

   (b) Hemos presentado una evaluación experimental de índices invertidos simulados mediante estructuras de datos que soportan consultas `rank` y `select`. Los índices invertidos clásicos generalmente obtienen mejores resultados, aunque nuestras propuestas son más versátiles, pudiendo realizar operaciones que no son posibles en caso de utilizar enfoques clásicos.

   (c) Hemos presentado un nuevo *FM-Index* para secuencias altamente repetitivas que usan algunas de las estructuras de datos presentadas a lo largo de la tesis. Hemos comparando nuestras propuestas con el estado del arte mostrando además que generalmente somos capaces de obtener las representaciones más compactas aunque siendo más lentos.

   (d) Hemos presentado un sistema de consultas *XPath* para secuencias altamente repetitivas que consiste en la re-ingeniería de un sistema *XPath*

para secuencias estadísticamente compresibles. Hemos construido un prototipo de dicho sistema y hemos realizado una evaluación experimental que sugiere que mediante el uso de estructuras de datos comprimidas para secuencias altamente repetitivas podemos reducir significativamente el coste espacial de sistemas clásicos cuando los aplicamos a secuencias en las que la repetitividad es alta.

(e) Hemos presentado el *Grammar Compressed Suffix Tree (GCST)*, que es uno de los árboles de sufijos más eficientes para secuencias altamente repetitivas. Además, hemos presentado una completa evaluación experimental en la que comparamos nuestra estructura con aquellas más eficientes en el estado del arte, mostrando que nuestra propuesta obtiene rendimientos espaciales y temporales muy competitivos.

B.2 Trabajo Futuro

En esta sección trataremos de resumir las líneas de trabajo futuro que se derivan de esta tesis en términos generales, más que centrarnos en las mejoras concretas o líneas de investigación que se derivan de cada una de las contribuciones propuestas.

Como ya ha sido mencionado con anterioridad, el área de las bases de datos con contenido altamente repetitivo es relativamente joven, con lo que actualmente nos encontramos con más problemas abiertos que soluciones reales. Como resultado, aún se siguen usando más representaciones comprimidas clásicas que estructuras de datos comprimidas para este tipo de problemas. Por ejemplo, los sistemas de control de versiones aún siguen utilizando el clásico enfoque de almacenar las diferencias entre versiones sucesivas de documentos, lo cual es muy eficiente en términos de espacio pero en términos de funcionalidad es bastante limitado: simplemente permiten recuperar una versión determinada. Sin embargo, y tal y como hemos demostrado a lo largo de la tesis, el simple hecho de usar estructuras de datos comprimidas nos permitiría ofrecer funcionalidades mucho más avanzadas usando muy poco espacio de almacenamiento. Esto sería interesante, por ejemplo, para extraer métricas en procesos de ingeniería del software de repositorios de código fuente si tuviésemos acceso eficiente a cada una de las versiones.

Con el objetivo de seguir empujando en esta dirección y para convencer a la comunidad de que este nuevo enfoque es más adecuado que el del uso de estructuras clásicas, construir sistemas completos bajo este enfoque para compararlo con los esquemas clásicos sería fundamental. Además, también sería muy interesante y necesario seguir trabajando para obtener estructuras de datos comprimidas más eficientes tanto en términos de espacio como de tiempo, en especial algoritmos y estructuras más rápidas.

Otro aspecto que merece más atención son los algoritmos de construcción. Es fundamental que estos algoritmos sean rápidos para que resulten atractivos y aplicables a cualquier contexto y, desafortunadamente, la mayoría de ellos no lo

son. Pero no solo la velocidad de construcción es un factor limitante. Muchos de ellos solo funcionan para entradas de un determinado tamaño, tal y como hace RePair, el mejor algoritmo que conocemos para construir gramáticas. Ser capaces de manejar cualquier entrada de cualquier tamaño es fundamental para que las propuestas que hemos presentado a lo largo de esta tesis sean de utilidad en problema reales.

Otra línea que suena realmente prometedora consiste en aplicar el algoritmo de construcción de los *Block Trees* para explotar la repetitividad estructuras más complejas tales como árboles o grafos. Esto tendría aplicaciones tan interesantes como los bases de datos de grafos, tan de moda hoy en día.

Adicionalmente, conseguir cierto grado de dinamismo en las estructuras de datos comprimidas sería un gran paso adelante y una limitación menos que superar para que este tipo de estructuras resulten más interesantes en un contexto general.

No menos importante es el hecho de que para contextos altamente repetitivos no disponemos de cotas inferiores para el espacio de las estructuras de datos comprimidas, con lo cual aún no sabemos cuán lejos podemos llega. Esto es, muy probablemente, consecuencia de la reciente naturaleza de este tipo de bases de datos.

Bibliography

[ACM+15] D. Arroyuelo, F. Claude, S. Maneth, V. Mäkinen, G. Navarro, K. Nguyễn, J. Sirén, and N. Välimäki. Fast in-memory XPath search using compressed indexes. *Software Practice and Experience*, 45(3):399–434, 2015.

[ACN13] A. Abeliuk, R. Cánovas, and G. Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013.

[ACNS10] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97, 2010.

[AF92] P. G. Anick and R. A Flynn. Versioning a full-text information retrieval system. In *Proc. of the 15th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 98–111, 1992.

[AGO10] D. Arroyuelo, S. González, and M. Oyarzún. Compressed self-indices supporting conjunctive queries on document collections. In *Proc. of the 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 43–54, 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.

[AM01] M. Adler and B. M. Maggs. Protocols for asymmetric communication channels. *Journal of Computer and System Sciences*, 63(4):573–596, 2001.

[AN12] A. Abeliuk and G. Navarro. Compressed suffix trees for repetitive texts. In *Proc. of the 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 30–41, 2012.

[Apo85] A. Apostolico. *The myriad virtues of subword trees*, pages 85–96. Combinatorial Algorithms on Words. NATO ISI Series. Springer-Verlag, 1985.

[BAHM12] J. Barbay, L.C. Aleardi, M. He, and J.I. Munro. Succinct representation of labeled graphs. *Algorithmica*, 62(1-2):224–257, 2012.

[BCG+14] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.

[BCN10] J. Barbay, F. Claude, and G. Navarro. Compact rich-functional binary relation representations. In *Proc. of the 9th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 6034, pages 170–183, 2010.

[BCPN14] N. Brisaboa, A. Cerdeira-Pena, and G. Navarro. Xxs: Efficient XPath evaluation on compressed xml documents. *ACM Transactions on Information Systems*, 32(3):13, 2014.

[BDM+05] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[BEF+06] A. Z. Broder, N. Eiron, M. Fontoura, M. Herscovici, R. Lempel, J. McPherson, R. Qi, and E. Shekita. Indexing shared content in information retrieval systems. In *Proc. of the Advances in Database Technology (EDBT)*, pages 313–330, 2006.

[Ben79] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[Ben80] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[BFLN12] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Implicit indexing of natural language text by reorganizing bytecodes. *Information Retrieval*, 15(6):527–557, 2012.

[BFNP07] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.

[BGG+15] D. Belazzougui, T. Gagie, P. Gawrychowski, J. Kärkkäinen, A. Ordóñez, S.J. Puglisi, and Y. Tabei. Queries on LZ-bounded encodings. In *Proc. of the 25th Data Compression Conference (DCC 2015)*, pages 83–92, Salt Lake City, Utah, 2015.

[BGMR07] J. Barbay, A. Golynski, I. Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theoretical Computer Science*, 387(3):284–297, 2007.

[BGOS11] T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. In *Proc 18th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7024, pages 197–208, 2011.

[BGVK+06] P. Boncz, T. Grust, M. Van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490. ACM, 2006.

[BHMM09] P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *Proc. of the 11th International Symposium on Algorithms and Data Structures (WADS)*, LNCS 5664, pages 98–109, 2009.

[BHMR11] J. Barbay, M. He, I. Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4):article 52, 2011.

[BLN13] N. Brisaboa, S. Ladra, and G. Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing and Management*, 49(1):392–404, 2013.

[BLNS10] N. Brisaboa, M. Luaces, G. Navarro, and D. Seco. A fun application of compact data structures to indexing geographic data. In *Proc. of the 5th International Conference on Fun with Algorithms (FUN)*, pages 77–88, 2010.

[BLR+11] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. S. Rao, and O. Weimann. Random access to grammar-compressed strings. In *Proc. of the 22th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 373–389. SIAM, 2011.

[BLR+15] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. S. Rao, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

[BN09] J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 111–122, 2009.

[BN13] J. Barbay and G. Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013.

[BN15] D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):article 31, 2015.

[BNO12] N. Brisaboa, G. Navarro, and A. Ordóñez. Smaller self-indexes for natural language. In *Proc. of the 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 372–378, 2012.

[Bow10] A. Bowe. *Multiary Wavelet Trees in Practice*. Honours thesis, RMIT University, Australia, 2010.

[BPT15] D. Belazzougui, S. J. Puglisi, and Y. Tabei. Access, rank, and select in grammar-compressed strings. In *Proc. of the 23rd Annual European Symposium on Algorithms (ESA)*, pages 142–154, 2015.

[BSTW86] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4), 1986.

[Bur93] M. Buro. On the maximum length of Huffman codes. *Information Processing Letters*, 45(5):219–223, 1993.

[BW94] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, 1994.

[BYRN11] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 2nd edition, 2011.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. INRIA, 2007.

[CF02] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[CFMPN11] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Indexes for highly repetitive document collections. In *Proc. of the 20th CIKM*, pages 463–468, 2011.

[Cha88] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[CHSV08] Y.-F. Chien, W.-K. Hon, R. Shah, and J. Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *Proc. of the 18th Data Compression Conference (DCC)*, pages 252–261, 2008.

[CKP85] Y. Choueka, S. T. Klein, and Y. Perl. Efficient variants of Huffman codes in high level languages. In *Proc. of the 8th Annual International ACM Conference on Research and development in Information Retrieval (SIGIR)*, pages 122–130. ACM, 1985.

[Cla] F. Claude. Libcds. `https://github.com/fclaude/libcds`. Downloaded: 2011.

[Cla96] D. Clark. *Compact Pat Trees.* PhD thesis, Univ. of Waterloo, Canada, 1996.

[CLL+05] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[CM07] J. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 137–148, 2007.

[CN08] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. of the 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[CN10a] R. Cánovas and G. Navarro. Practical compressed suffix trees. In *Proc. of the 9th International Symposium on Experimental Algorithms (SEA)*, LNCS 6049, pages 94–105, 2010.

[CN10b] F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2010.

[CN12] F. Claude and G. Navarro. The wavelet matrix. In *Proc. of the 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 167–179, 2012.

[CNO15] F. Claude, G. Navarro, and A. Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

[CNS11] F. Claude, P. Nicholson, and D. Seco. Space efficient wavelet tree construction. In *Proc. of the 18th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 185–196, 2011.

[DJSS12] H.-H. Do, J. Jansson, K. Sadakane, and W.-K. Sung. Fast relative Lempel-Ziv self-index for similar sequences. In *Proc. of the Joint International Conference on Frontiers in Algorithmics and Algorithmic Aspects in Information and Management (FAW-AAIM)*, pages 291–302, 2012.

[DKMS08] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, 12:article 3.4, 2008.

[DRR06] O. Delpratt, N. Rahman, and R. Raman. Engineering the LOUDS succinct tree representation. In *Proc. of the 5th International Workshop on Experimental Algorithms Experimental Algorithms (WEA)*, volume 4007, page 134. Springer, 2006.

[DRS12] P. Davoodi, R. Raman, and S. Rao Satti. Succinct representations of binary trees for range minimum queries. In *Proc. of the 18th Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS 7434, pages 396–407, 2012.

[FBN+12] A. Fariña, N. Brisaboa, G. Navarro, F. Claude, A. Places, and E. Rodríguez. Word-based self-indexes for natural language text. *ACM Transactions on Information Systems*, 30(1):article 1, 2012.

[FG99] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[FGM09] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849–866, 2009.

[FGM12] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[FGMS05] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005.

[Fis10] J. Fischer. Wee LCP. *Information Processing Letters*, 110:317–320, 2010.

[FKS84] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $\mathcal{O}(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[FLMM09] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):article 4, 2009.

[FM05] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[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.

[FOP14] A. Fariña, A. Ordóñez, and J. R. Paramá. Indexing and self-indexing sequences of IEEE 754 double precision numbers. *Information Processing & Management*, 50(6):857 – 875, 2014.

[FSC+03] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The GALAX experience. In *Proc. of the 29th international conference on Very Large Databases (VLDB)*, volume 29, pages 1077–1080, 2003.

[FW93] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[Gag06a] T. Gagie. Compressing probability distributions. *Information Processing Letters*, 97(4):133–137, 2006.

[Gag06b] T. Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.

[Gag08] T. Gagie. Dynamic asymmetric communication. *Information Processing Letters*, 108(6):352–355, 2008.

[GGK+12] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S.J. Puglisi. A faster grammar-based self-index. In *Proc. of the 6th International Conference on Language and Automata Theory and Applications (LATA)*, LNCS 7183, pages 240–251, 2012.

[GGMN05] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. of the 4th International Workshop on Experimental Algorithms (WEA)*, pages 27–38, 2005.

[GGP11] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Faster approximate pattern matching in compressed repetitive texts. In *Proc. of the 22nd International Symposium Algorithms and Computation (ISAAC)*, pages 653–662, 2011.

[GGV03] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[GHP14] T. Gagie, C. Hoobin, and S. J. Puglisi. Block graphs in practice. In *Proc. of the 2nd International Conference on Algorithms for Big Data (ICABD)*, pages 30–36, 2014.

[GKNP13] T. Gagie, J. Kärkkäinen, G. Navarro, and S.J. Puglisi. Colored range queries and document retrieval. *Theoretical Computer Science*, 483:36–50, 2013.

[GM59] E. N. Gilbert and E. F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38:933–967, 1959.

[GMC+14] S. Gog, A. Moffat, J. S. Culpepper, A. Turpin, and A. Wirth. Large-scale pattern search using reduced-space on-disk suffix arrays. *IEEE Transactions on Knowledge and Data Engineering*, 26(8):1918–1931, 2014.

[GMR06] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[GN09] T. Gagie and Y. Nekrich. Worst-case optimal adaptive prefix coding. In *Proc. of the 9th Symposium on Algorithms and Data Structures (WADS)*, pages 315–326, 2009.

[GNN10] T. Gagie, G. Navarro, and Y. Nekrich. Fast and compact prefix codes. In *Proc. of the 36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 5901, pages 419–427, 2010.

[GNNO15] T. Gagie, G. Navarro, Y. Nekrich, and A. Ordóñez. Efficient and compact representations of prefix codes. *IEEE Transactions on Information Theory*, 61(9):4999–5011, 2015.

[GNP12] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012.

[Gog11] S. Gog. *Compressed Suffix Trees: Design, Construction, and Applications*. PhD thesis, University of Ulm, Germany, 2011.

[Gol07] A. Golynski. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science*, 387(3):348–359, 2007.

[GOR10] R. Grossi, A. Orlandi, and R. Raman. Optimal trade-offs for succinct string indexes. In *Proc. of the 37th International Colloquium on Algorithms, Languages and Programming (ICALP)*, pages 678–689, 2010.

[GP14] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software, Practice, and Experience.*, 44(11):1287–1314, 2014.

[GPT09] T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 1–6, 2009.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[GV06] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2006.

[GVX11] R. Grossi, J. Vitter, and B. Xu. Wavelet trees: From theory to practice. In *Proc. of the 1st International Conference on Data Compression, Communications and Processing (CCP)*, pages 210–221, 2011.

[Has95] R. Hashemian. Memory efficient and high-speed search Huffman coding. *IEEE Transactions on Communications*, 43(10):2576–2581, 1995.

[HM10] M. He and I. Munro. Succinct representations of dynamic strings. In *Proc. of the 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 334–346, 2010.

[Hor77] Y. Horibe. An improved bound for weight-balanced tree. *Information and Control*, 34(2):148–151, 1977.

[HT71] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal of Applied Mathematics*, 21(4):514–532, 1971.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. volume 40, pages 1090–1101, 1952.

[HYS09] J. He, H. Yan, and T. Suel. Compact full-text indexing of versioned document collections. In *Proc. of the 18th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 415–424, 2009.

[HZS10] J. He, J. Zeng, and T. Suel. Improved index compression techniques for versioned document collections. In *Proc. of the 19th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1239–1248, 2010.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. of the 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[JSS12] J. Jansson, K. Sadakane, and W-K. Sung. Ultra-succinct representation of ordered trees with applications. *Journal of Computer and System Sciences*, 78(2):619–631, 2012.

[JW04] L.B. Jorde and S.P. Wooding. Genetic variation, classification and 'race'. *Nature Genetics*, 36(11s):S28–33, 2004.

[Kay08] M. Kay. Ten reasons why Saxon XQuery is fast. *IEEE Data Engineering Bulletin*, 31(4):65–74, 2008.

[KK14a] J. Kärkkäinen and D. Kempa. Engineering a lightweight external memory suffix array construction algorithm. In *Proc. of the 2nd International Conference on Algorithms for Big Data (ICABD)*, pages 53–60. CEUR, 2014.

[KK14b] J. Kärkkäinen and D. Kempa. LCP array construction in external memory. In *Proc. of the 13th International Symposium on Experimental Algorithms (SEA)*, LNCS 8504, pages 412–423. Springer, 2014.

[KN76] G. O. H. Katona and T. O. H. Nemetz. Huffman codes and self-information. *IEEE Transactions on Information Theory*, 22(3):337–340, 1976.

[KN09] M. Karpinski and Y. Nekrich. A fast algorithm for adaptive prefix coding. *Algorithmica*, 55(1):29–41, 2009.

[KN13] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[Knu73] D. E. Knuth. *The Art of Computer Programming. Vol. 3: Sorting and Searching.* Addison-Wesley, 1973.

[Kol65] A. Kolmogorov. Three approaches to the quantitative definition of information'. *Problems of information transmission*, 1(1):1–7, 1965.

[Kol68] A. Kolmogorov. Logical basis for information theory and probability theory. *Information Theory, IEEE Transactions on*, 14(5):662–664, 1968.

[KP11] J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proc. of the 18th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 174–184, 2011.

[KPZ11] S. Kuruppu, S.J. Puglisi, and J. Zobel. Optimized relative Lempel-Ziv compression of genomes. In *Proc. of the 34th Australasian Computer Science Conference (ACSC)*, pages 91–98, 2011.

[KR87] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[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.

[Kur99] S. Kurtz. Reducing the space requirements of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.

[LH90] L. L. Larmore and D. S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, 37(3):464–473, 1990.

[LM00] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. of the IEEE*, 88(11):1722–1732, 2000.

[LM06] M. Liddell and A. Moffat. Decoding prefix codes. *Software, Practice and Experience*, 36(15):1687–1710, 2006.

[LMM13] M. Lohrey, S. Maneth, and R. Mennicke. XML tree structure compression using repair. *Information Systems*, 38(8):1150–1167, 2013.

[Lue78] G. S. Lueker. A data structure for orthogonal range queries. In *Proc. of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 28–34. IEEE, 1978.

[LW80] D. T. Lee and C. K. Wong. Quintary trees: a file structure for multidimensional datbase sytems. *ACM Transactions on Database Systems*, 5(3):339–353, 1980.

[Mak12] C. Makris. Wavelet trees: a survey. *Computer Science and Information Systems*, 9(2):585–625, 2012.

[Man01] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[MB04] S. Maneth and G. Busatto. Tree transducers and tree compressions. In *Proc. of the 7th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, LNCS 2987, pages 363–377, 2004.

[McC76] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 32(2):262–272, 1976.

[MK95] A. Moffat and J. Katajainen. In-place calculation of minimum-redundancy codes. In *Proc. of the 4th International Workshop Algorithms and Data Structures (WADS)*, pages 393–402, 1995.

[ML01] R. L. Milidiú and E. S. Laber. Bounding the inefficiency of length-restricted prefix codes. *Algorithmica*, 31(4):513–529, 2001.

[MLMD03] R. L. Milidiú, E. S. Laber, L. O. Moreno, and J. C Duarte. A fast decoding method for prefix codes. In *Proc. of the 13th Data Compression Conference (DCC)*, page 438, 2003.

[MM93] U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):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.

[MN06] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proc. of the 7th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 703–714, 2006.

[MN07a] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. of the 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.

[MNKS13] S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto. Esp-index: A compressed index based on edit-sensitive parsing. *Journal of Discrete Algorithms*, 18:100–112, 2013.

[MNSV10] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[MNZB00] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[Mof89] A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, 1989.

[Mp07] XML Mind products. Qizx XML query engine. `http://www.xmlmind.com/qizx`, 2007.

[MR01] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[MRRR03] J. Munro, R. Raman, V. Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *Proc. of the 30th International Colloquium on Automata, Languages, and Programming (ICALP)*, LNCS 2719, pages 345–356, 2003.

[MT97] A. Moffat and A. Turpin. On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, 1997.

[Mun96] I. Munro. Tables. In *Proc. of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[Nak91] N. Nakatsu. Bounds on the redundancy of binary alphabetical codes. *IEEE Transactions on Information Theory*, 37(4):1225–1229, 1991.

[Nav04] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.

[Nav09] G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics*, 13:article 2, 2009.

[Nav12] G. Navarro. Indexing highly repetitive collections. In *Proc. of the 23rd International Workshop on Combinatorial Algorithms (IWOCA)*, LNCS 7643, pages 274–279, 2012.

[Nav14] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[NMWM94] C. G. Nevill-Manning, I. H. Witten, and D. L. Maulsby. Compression by induction of hierarchical grammars. In *Proc. of the Data Compression Conference (DCC)*, pages 244–253, 1994.

[NN12] G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1066–1078, 2012.

[NN14] G. Navarro and Y. Nekrich. Optimal dynamic sequence representations. *SIAM Journal on Computing*, 43(5):1781–1806, 2014.

[NNR13] G. Navarro, Y. Nekrich, and L. Russo. Space-efficient data-analysis queries on grids. *Theoretical Computer Science*, 482:60–72, 2013.

[NO13] G. Navarro and A. Ordóñez. Compressing Huffman models on large alphabets. In *Proc. of the 23rd Data Compression Conference (DCC)*, pages 381–390, 2013.

[NO14a] G. Navarro and A. Ordóñez. Faster compressed suffix trees for repetitive text collections. In *Proc. of the 13th International Symposium on Experimental Algorithms (SEA)*, LNCS 8504, pages 424–435, 2014.

[NO14b] G. Navarro and A. Ordóñez. Grammar compressed sequences with rank/select support. In *Proc. of the 21st International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 8799, pages 31–44, 2014.

[NO15] G. Navarro and A. Ordóñez. Faster compressed suffix trees for repetitive collections. *ACM Journal of Experimental Algorithmics*, 2015. To appear.

[NP10] G. Navarro and S. J. Puglisi. Dual-sorted inverted lists. In *Proc. of the 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 310–322, 2010.

[NP12] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proc. of the 11th International Symposium on Experimental Algorithms (SEA)*, LNCS 7276, pages 295–306, 2012.

[NPC+13] J. C. Na, H. Park, M. Crochemore, J. Holub, C. S. Iliopoulos, L. Mouchard, and K. Park. Suffix tree of alignment: An efficient index for similar data. In *Proc. of the International Workshop on Combinatorial Algorithms (IWOCA)*, LNCS 8288, pages 337–348. Springer, 2013.

[NPV14] G. Navarro, S. J. Puglisi, and D. Valenzuela. General document retrieval in compact space. *ACM Journal of Experimental Algorithmics*, 19(2):article 3, 2014. 46 pages.

[NS14] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

[OFG10] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *Proc. of the 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 6393, pages 322–333, 2010.

[Ohl13] E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013.

[ONB15] A. Ordóñez, G. Navarro, and N. Brisaboa. Grammar compressed sequences with rank/select support. In *Submitted to Information Systems Jouranl*, 2015.

[OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

[Pag01] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[PT08] S. Puglisi and A. Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *Proc. of the 19th International Symposium on Algorithms and Computation (ISAAC)*, pages 124–135, 2008.

[PV10] M. Pătraşcu and E. Viola. Cell-probe lower bounds for succinct partial sums. In *Proc. of the 21st annual ACM-SIAM symposium on Discrete Algorithms (SODA)*, pages 117–122, 2010.

[RNO11] L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):article 53, 2011.

[RRR07] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):article 43, 2007.

[Ryt03] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1):211–222, 2003.

[Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algor.*, 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.

[Sak05] H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3:416–430, 2005.

[Sal07] D. Salomon. *Data Compression.* Springer, 2007.

[SC07] T. Strohman and B. Croft. Efficient document retrieval in main memory. In *Proc. of the 30th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 175–182, 2007.

[She92] D. Sheinwald. On binary alphabetic codes. In *Proc. of the 2nd Data Compression Conference (DCC)*, pages 112–121, 1992.

[Sie88] A. Siemiński. Fast decoding of the Huffman codes. *Information Processing Letters*, 26(5):237–241, 1988.

[Sig08] Signum. Tauro. `http://tauro.signum.sns.it/`, 2008.

[SK64] E. S. Schwarz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.

[SOG10] T. Schnattinger, E. Ohlebusch, and S. Gog. Bidirectional search in a string with wavelet trees. In *Proc. of the 21st Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 6129, pages 40–50, 2010.

[ST07] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[SW49] C. E. Shannon and W. Weaver. A mathematical theory of communication, 1949.

[SWYZ02] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of the 25th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 222–229, 2002.

[Tis11] G. Tischler. On wavelet tree construction. In *Proc. of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 208–218, 2011.

[TK04] S.A. Tishkoff and K.K. Kidd. Implications of biogeography of human populations for 'race' and medicine. *Nature Genetics*, 36(11s):S21–27, 2004.

[TM00] A. Turpin and A. Moffat. Housekeeping for prefix coding. *IEEE Transactions on Communications*, 48(4):622–628, 2000.

[TTS13] Y. Tabei, Y. Takabatake, and H. Sakamoto. A succinct grammar compression. In *Proc. of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7922, pages 235–246, 2013.

[TTS14] Y. Takabatake, Y. Tabei, and H. Sakamoto. Improved esp-index: A practical self-index for highly repetitive texts. In *Proc. of the 13th International Symposium on Experimental Algorithms (SEA)*, pages 338–350. 2014.

[Ukk95] E. Ukkonen. Constructing suffix trees on-line in linear time. *Algorithmica*, 14(3):249–260, 1995.

[vEBKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Vig08] S. Vigna. Broadword implementation of rank/select queries. In *Proc. of the 7th International Workshop on Experimental Algorithms (WEA)*, LNCS 5038, pages 154–168, 2008.

[VM07] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 205–215, 2007.

[VY13] E. Verbin and W. Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7922, pages 247–258, 2013.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[WMB99] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.

[WZ99] H. E. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[XPa] XPath. W3C recommendation of XML path language (XPath). `www.w3.org/TR/xpath/`. Accessed: 10/2015.

[XQu] XQuery. W3C recommendation of XML query language (XQuery). `www.w3.org/TR/xquery/`. Accessed: 10/2015.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[ZPYL08] Y. Zhang, Z. Pei, J. Yang, and Y. Liang. Canonical Huffman code based full-text index. *Progress in Natural Science*, 18(3):325–330, 2008.