Space-Efficient Data Structures for Information Retrieval

by

Francisco Claude

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The amount of data that people and companies store has grown exponentially over the last few years. Storing this information alone is not enough, because in order to make it useful we need to be able to efficiently search inside it. Furthermore, it is highly valuable to keep the historic data of each document stored, allowing to not only access and search inside the newest version, but also over the whole history of the documents.

Grammar-based compression has proven to be very effective for repetitive data, which is the case for versioned documents. In this thesis we present several results on representing textual information and searching in it. In particular, we present text indexes for grammar-based compressed text that support searching for a pattern and extracting substrings of the input text. These are the first general indexes for grammar-based compressed text that support searching in sublinear time.

In order to build our indexes, we present new results on representing binary relations in a space-efficient manner, and construction algorithms that use little space to achieve their goal. These two results have a wide range of applications. In particular, the representations for binary relations can be used as a building block for several structures in computer science, such as graphs, inverted indexes, etc.

Finally, we present a new index, that uses on grammar-based compression, to solve the document listing problem. This problem deals with representing a collection of texts and searching for the documents that contain a given pattern. In spite of being similar to the classical text indexing problem, this problem has proven to be a challenge when we do not want to pay time proportional to the number of occurrences, but time proportional to the size of the result. Our proposal is designed particularly for versioned text, allowing the storage of a collection of documents with all their historic versions in little space. This is currently the smallest structure for such a purpose in practice.

Acknowledgements

First of all, I would like to thank my supervisor, J. Ian Munro, for his patience and support. I have learned a lot from him, and I am really grateful for all the advice I received from him throughout this process. I would also like to thank my master's supervisor, Gonzalo Navarro, with whom I have had the pleasure of continuing working intensively, as many of the results in this thesis reflect.

There are many people that directly contributed to the results in this thesis. I collaborated with Ian and Gonzalo, Jérémy Barbay, Patrick K. Nicholson and Diego Seco. My thesis committee, Gordon Agnew, Charles Clarke, Gad Landau, and Alejandro López-Ortiz, gave me many suggestions for improvement and useful comments. I also have to give special thanks to Moshe Lewenstein and Venkatesh Raman for correcting early drafts of my thesis; its submission would not have been the same without their help.

Thanks to all my friends that made these years so amazing and kept things interesting: Diana Berrios, Rodrigo Cánovas, Alessandro Cosentino, Nadia Eger, Antonio Fariña, Arash Farzan, Lucía Fraguela, Robert Fraser, Daniel Guajardo, Francisco Gutiérrez, Divam Jain, Shahin Kamali, Natalia Klein, Roberto Konow, Susana Ladra, Daniela Maftuleac, Miguel A. Martínez-Prieto, Guillermo Medel, Patrick K. Nicholson, Catalina Orb, Kimiisa Oshikoji, Philippe Pavez, Elena Ruiz, Alejandro Salinger, Ricardo Salmon, Barbara Sawicka, Diego Seco, Cherry Zhang, and Gelin Zhou.

During my stay in Waterloo I also got to discover new activities that are now part of my life. I will certainly miss everybody at the University of Waterloo Goju-Ryu Karate Club, ran by Sensei William Durocher and Sempai Nancy Soontiens. And I will also miss playing for the Hopeless Experts soccer team.

My family was a big support from afar. I would like to thank my parents, Ingrid Faust and Francisco Claude, my sister Carolina, and both my grandmothers, Nicole Bourdel and Sonja Kühne. I also have great memories of our long Skype conversations with my family in Germany, Elisabeth Hernández, Werner Faust, and my two cousins, Werner and Anton Faust.

I also got so much help from the staff at the university, especially, from Wendy Rush and Helen Jardine. They made my life at the university so much easier and fun.

Finally, I also have to thank Google for their support with the Ph.D. Fellowship in Search and Information Retrieval. I had great times with my research mentor Stefan Büttcher. It was also an amazing experience to work for a couple of months there. I was very lucky to have Henry A. Rowley and Sanjiv Kumar as my supervisors; I learnt a lot from them.

Contents

List of Figures

List of Tables

1 Introduction

In the last few decades we have seen an explosion on the amount of information that people and companies store. This has brought interesting new challenges, mainly related on how to organize this information to make it useful. Being useful depends on the context, it might mean we want to be able to find pieces of information related to a given topic, or extract summaries, or somehow generate new information in an automatic way from existing sources. All of these tasks are expensive, and require a good underlying representation of the data in order to be feasible in practice.

In this work we focus mostly on structures that support searching in the field of information retrieval. The structures presented in this thesis work in main memory and aim at being space efficient. We aim at providing building blocks for more complex processes that allow us to represent in as little memory as possible the information we require, allowing the algorithms run in main memory and hopefully not have to resort to secondary storage, which can be up to 10^6 times slower.

Our structures not only help avoid disk accesses, but they also pose an interesting option for distributed systems that handle data. The argument is very simple: If the data representation takes less memory, fewer machines are required to maintain the system running. This helps in terms of cost, energy, and administration overhead among others.

The next sections introduce some basic concepts required to follow the rest of the document. We close this chapter with Section 1.4, presenting in more detail the topics covered, and mentioning why they are interesting when put together.

1.1 Basic Succinct Data Structures

In this section we present the basic ideas of succinct data structures which will be used as building blocks for more complex structures. We present succinct representations that achieve space close to the information theoretic lower bound up to lower order terms.

1.1.1 The model

All the structures covered and used across this thesis work in the word-RAM model. This model assumes that we have a constant number of registers we can use to operate and address the RAM, and each register has size $\Theta(\lg n)$ bits[1], where n is the size of the RAM in bits. The memory is split into *words*, that are contiguous blocks of $\Theta(\lg n)$ bits, which we can load and store in constant time.

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

The operations supported in constant time in the model are the basic arithmetic operations $(+,-,\times,/)$, bit shifts $(\ll,\gg)$, obtaining the most significant bit or counting the number of bits set to 1 in the word. In case these two last operations are not natively supported, it is possible to compute them in constant time using tables that require $o(n)$ bits of space [85].

When we talk about space required by an object we can either count it in bits or words. The latter is equivalent to counting how many values of $O(\lg n)$ bits are required to represent the object.

1.1.2 Representing an object

Consider an object $\mathcal{O}(\ell_1,\ell_2,\ldots,\ell_k)$, where $\mathcal{L} = (\ell_1,\ldots,\ell_k)$ are parameters describing the object. The information theoretic lower bound is $\lg(\#\mathcal{O}(\mathcal{L}))$, where $\#\mathcal{O}(\mathcal{L})$ corresponds to the number of objects that fit the description $\mathcal{L}$.

Two basic examples of the information theoretic lower bound for a bitmap (i.e., a string or sequence composed of 1s and 0s) are given below:

- Consider $\mathcal{L} = (n)$, just the length of the bitmap, then we have 2^n different bitmaps and the information theoretical lower bound is then $n = \lceil \lg(2^n) \rceil$ bits.
- If we take $\mathcal{L} = (n,m)$, where n is the length of the bitmap and m corresponds to the number of bits set to 1, there are $\binom{n}{m}$ possible bitmaps of length n with m 1s. The lower bound for this characterization is roughly $m \lg \frac{n}{m} + (n-m) \lg \frac{n}{n-m}$.

The second example corresponds to the 0-th order empirical entropy of the bitmap. This is a common measure of the complexity for representing a sequence S drawn over an alphabet Σ, the 0-order entropy H_0 is defined as:

$$H_0(S) = \sum_{c \in \Sigma} \Pr(c|S) \lg \frac{1}{\Pr(c|S)}$$

$\Pr(c|S)$ corresponds to the probability of encountering symbol c in sequence S. We assume $0 \lg 0 = 0$. We say empirical entropy in the case of a fixed sequence when we estimate $P(c)$ as the frequency of c divided by the length of the sequence. In general we will talk about entropy without making this distinction for finite fixed sequences.

The notion of H_0 can be extended to better capture the structure of the sequence if we take into account fixed-length contexts in which every symbol appears. This measure is called k-th order entropy, H_k, and is formalized as:

$$H_k(S) = \sum_{s \in \Sigma^k} |S \perp s| H_0(S \perp s)$$

where $S \perp s$ corresponds to the string composed of all symbols appearing just after an occurrence of s in S.

1.1.3 Sequences

A sequence S of length n, over an alphabet Σ, corresponds to an n-tuple of elements from Σ. We usually use σ to denote the size of Σ, and assume $\Sigma = [\sigma] = \{1, \ldots, \sigma\}$. A lower bound on the expected space required to represent S is $n \lg \sigma$ bits, this occurs if all characters are equally likely.

We are not only interested in representing sequences in little space, but the representations should also to support the following operations in little space:

- $S.\texttt{access}(i \in [n])$: retrieve the symbol at position i in S.
- $S.\texttt{rank}(c \in \Sigma, i \in [n])$: count the number of times the symbol c appears in S up to position i (inclusive).
- $S.\texttt{select}(c \in \Sigma, j \in [n])$: find the position in S where symbol c appears for the j-th time, or return $n+1$ when it appears fewer than j times in S.

We first comment on the solution for the most basic case where $\Sigma = \{0, 1\}$, then discuss the more general case.

Binary Sequences

We usually refer to the case of binary sequences as bitmaps or bitstrings. The operations defined above apply to this case and they can be solved in $O(1)$ time within $n + O(n \lg \lg n / \lg n) = n + o(n)$ bits of space [67, 24, 85]. Furthermore, Raman et al. [101] proposed two compressed representations that are useful when the number m of 1s in S is small. One is the "fully indexable dictionary" (FID). It takes $m \lg \frac{n}{m} + O(m + n \lg \lg n / \lg n)$ bits and supports all the operations in constant time. A weaker one is the "indexable dictionary" (ID), that takes $m \lg \frac{n}{m} + O(m + \lg \lg n)$ bits and supports, in constant time, queries $S.\texttt{access}(i)$, $S.\texttt{rank}(1, i)$ if $S[i] = 1$, and $S.\texttt{select}(1, j)$.

The case of the FID has been further improved, reducing the lower order term. First by Golynski [53], whose representation achieves $\frac{n \lg(t \lg n)}{t \lg n} + O(\frac{n \lg^2(t \lg n)}{t^2 \lg^2 n})$ bits of space, and achieve time $O(t)$ for the three operations. Later, Pătraşcu [99] improved the lower order term to $\frac{n}{(\lg nt)^t} + \tilde{O}(n^{3/4})$ bits of space while supporting queries in $O(t)$ time. Recently, Grossi et al. [60] improved it further to $nH_0(B) + O(m^{1+\delta} + m(\frac{n}{m^s})^\epsilon)$, where $0 < \delta < 1/2, 0 < \epsilon \leq 1$ and s is an integer greater than 0. Their query time is $O(s\delta^{-1} + \epsilon^{-1})$ for the three operations.

Sadakane and Okanohara proposed several practical implementations of FIDs [98]. The tradeoff achieved by one of their solutions, together with the other ones discussed in this section, are summarized in Table 1.1. We ignore the access operation, since it can be supported in at most the same time as rank.

Proposal	Space in bits	rank	select
Jacobson [67], Clark & Munro [24, 85]	$n + O(\frac{n \lg\lg n}{\lg n})$	$O(1)$	$O(1)$
Raman et al. [101]	$nH_0(B) + O(\frac{n \lg\lg n}{\lg n})$	$O(1)$	$O(1)$
Golysnki [53]	$nH_0(B) + \frac{n \lg(t \lg n)}{t \lg n} + O(\frac{n \lg^2(t \lg n)}{t^2 \lg^2 n})$	$O(t)$	$O(t)$
Pătraşcu [99]	$nH_0(B) + \frac{n}{(\lg nt)^t} + \tilde{O}(n^{3/4})$	$O(t)$	$O(t)$
Okanohara and Sadakane [98]	$m \lg \frac{n}{m} + 1.92m + o(m)$	$O(\lg \frac{n}{m} + \frac{\lg^4 m}{\lg n})$	$O(\frac{\lg^4 m}{\lg n})$
Grossi et al. [60]	$nH_0(B) + O(m^{1+\delta} + m(\frac{n}{m^s})^\epsilon)$	$O(s\delta^{-1} + \epsilon^{-1})$	$O(s\delta^{-1} + \epsilon^{-1})$

Table 1.1: Summary of bitmap representations for a bitmap B of length n with m ones.

General Sequences

For larger alphabets the problems get harder and several solutions have been proposed. The most commonly used is the wavelet tree [59]. The wavelet tree takes $n \lg \sigma + o(n) \lg \sigma$ bits and supports all three operations in $O(\lg \sigma)$ time.

It is also possible to achieve compression by representing the internal bitmaps of the wavelet tree using a compressed representation, or altering the shape of the tree to save bits [91]. We review wavelet trees in more detail in Chapter 2 (Section 2.1).

In the case where $\sigma = O(\text{polylog}\, n)$ it is possible to achieve 0-order compression while answering queries in constant time [47]. When this solution is combined with the wavelet tree, obtaining a higher fan-out in the tree, the query time becomes $O(\lg \sigma / \lg\lg n)$ [47]. This structure is known as *generalized wavelet trees*.

Another proposal, by Golynski et al., achieves competitive space with respect to wavelet trees for large alphabets [54], $n \lg \sigma + o(\lg \sigma) n$ bits, while offering two tradeoff that achieve $o(\lg \sigma)$ time for all operations.

Finally, a representation by Barbay et al. [8, 7] achieves $nH_0(S)(1 + o(1))$ bits, and supports select queries in $O(1)$ time, and rank and access queries in $O(\lg\lg \sigma)$ time.

Table 1.2 shows a summary of the time and space achieved by each one of the solutions commented.

In this work, we assume the alphabet Σ to be contiguous (i.e., $[\sigma]$). If the alphabet is not contiguous, we can use a bitmap $C[1, \sigma]$ marking the symbols of Σ that are used, and we make them to a contiguous range $\Sigma' = [\sigma']$. We use $C.\texttt{select}(1, i)$ to find the i-th alphabet symbol and $C.\texttt{rank}(1, x)$ to find the rank of symbol x in a contiguous list of those used. By using Raman et al.'s representation, C requires at most $\sigma \lg \frac{\sigma}{\sigma'} + o(\sigma)$ bits, while supporting *rank* and *select* on C in constant time. With this we can assume that the alphabet used is contiguous in $[\sigma]$, adding little space on top of the sequence.

Proposal	Space in bits	access	rank	select
Wavelet Trees [59]	$nH_0(S)+o(n)\lg\sigma$	$O(\lg\sigma)$	$O(\lg\sigma)$	$O(\lg\sigma)$
Generalized Wavelet Trees [47]	$nH_0(S)+o(n)\lg\sigma$	$O(\frac{\lg\sigma}{\lg\lg n})$	$O(\frac{\lg\sigma}{\lg\lg n})$	$O(\frac{\lg\sigma}{\lg\lg n})$
Golynski et al. [54]	$n\lg\sigma+o(\lg\sigma)n$	$O(\lg\lg\sigma)$	$O(\lg\lg\sigma)$	$O(1)$
Golynski et al. [54]	$n\lg\sigma+o(\lg\sigma)n$	$O(1)$	$O(\lg\lg\sigma\lg\lg\lg\sigma)$	$O(\lg\lg\sigma)$
Barbay et al. [8, 7]	$nH_0(S)(1+o(1))$	$O(\lg\lg\sigma)$	$O(\lg\lg\sigma)$	$O(1)$

Table 1.2: Tradeoffs for representing a string S of length n over an alphabet of size σ.

1.1.4 Trees

The representation of trees, in fact, motivated most of the work on `rank` and `select` queries over bitmaps. There are $\frac{1}{n+1}\binom{2n}{n}$ binary trees with n nodes [58]. Since this value is about $\frac{4^n}{n^{3/2}\sqrt{\pi}}$, their representation takes at least $2n-\frac{3}{2}\lg n-2$ bits. This result remains the same for non-isomorphic ordinal trees. These are trees where the order of the children of a node is fixed. For the remainder of this thesis, we consider only ordinal trees with n nodes, without adding the restriction of being binary.

The most common solutions for representing trees achieve $2n+o(n)$ bits, some well known examples are LOUDS [67], BP [67] and DFUDS [15], all of them follow a similar approach: store a bitmap of length $2n$ and then index the bitmap to support operations rank, select and some other operations.

Given a tree T with n nodes, where every node has a unique integer id freely assigned from $[2n]$ by the representation, we consider the following queries:

- T.`parent`(i): returns the id of the parent node of the node i.
- T.`child`(i,k): returns the k-th child of node i.
- T.`child – rank`(i): returns the number of siblings before node i.
- T.`degree`(i): produces the number of children of node i.
- T.`depth`(i): computes the depth of node i, this is, the length of the path from the root to node i.
- T.`lca`(i,j): retrieves the lowest common ancestor of nodes i and j.
- T.`level_ancestor`(i,k): obtains the k-th ancestor of node i.
- T.`subtree_size`(i): returns the size of the subtree rooted at i.
- T.`height`(i): computes the height of node i, this is, the length of the longest path from node i to a leaf descendant from i.

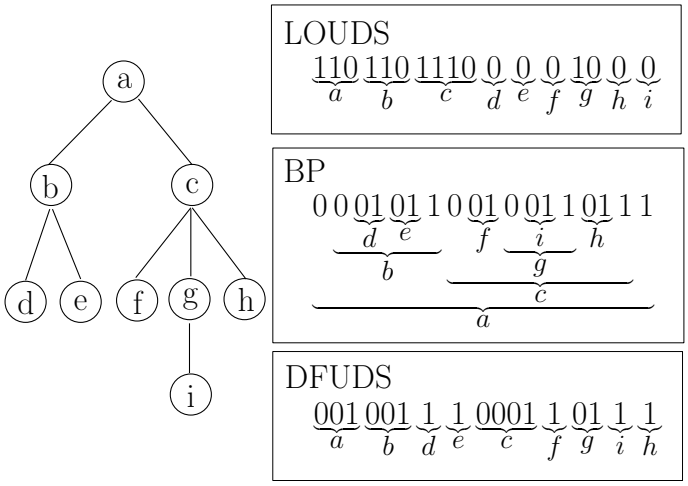

Figure 1.1: Example of a tree and its succinct representations.

We next present the five most common tree representations with a brief description of the main idea used for organizing the data, and the operations supported in constant time.

LOUDS. Level Order Unary Degree Sequence was proposed by Jacobson [67]. It represents each node by writing the degree of each node in unary. Figure 1.1 shows an example of a tree and its representation using LOUDS.

This representation supports `parent`, `child`, `child-rank` and `degree` in constant time, the space requirement is $2n + O(\frac{n \lg\lg n}{\lg n})$ bits.

BP. The Balanced Parenthesis representation works by doing a depth-first traversal [67]. Every time we visit a node we write an opening bracket (0) and every time we finish traversing a subtree we write a closing bracket (1). Figure 1.1 shows an example of the BP representation.

This representation supports all the operations considered in the list in $O(1)$ time [25], except for `child`(i,k), that takes $O(k)$ time. The space is also $2n + O(\frac{n \lg\lg n}{\sqrt{\lg n}})$ bits [87].

FF. The Fully-Functional representation [107] is based on BP but achieves $2n + O(n/\text{polylog}(n))$ bits of space, obtaining a better lower order term while supporting all operations in constant time. Their original work [107] also shows how to obtain dynamic representations.

DFUDS. The Depth First Unary Degree Sequence [15] combines both approaches, LOUDS and BP, extending the set of operations supported. In this work we are only concerned with the ones

listed before. The construction is by doing the same traversal as BP but writing one opening bracket per child every time we visit a node. Figure 1.1 shows an example, 0 represent opening brackets and 1 closing brackets. This representation offers the same tradeoff as BP.

TC. The last approach considered in this work is the one based on Tree Covering, where the tree is split into smaller pieces in such a way that it allows a succinct representation [63, 42]. The space requirement is $2n + o(n)$ and it also supports all operations listed in constant time.

1.1.5 Permutations

Another structure we consider among basic building blocks for general data structures are permutations. Given a permutation Π over $[n]$, we would like to answer efficiently the following queries:

- $\pi(i)$: obtain the value of $\Pi[i]$.
- $\pi^{-1}(j)$: find i such that $j = \Pi[i]$.
- $\pi^k(i)$: apply π k times, similarly we define $\pi^{-k}(j)$.
- $\mathrm{range}_\Pi(i_1, i_2, j_1, j_2)$: find elements i such that $i_1 \leq i \leq i_2$ and $j_1 \leq \pi(i) \leq j_2$.

Interesting solutions have been proposed for the succinct scenario. The first one by Munro et al. [86] achieves $(1 + 1/t)n \lg n + o(n)$ bits while supporting π in constant time and π^{-1} in $O(t)$. By adding $n(1 + o(1))$ extra bits, they support π^k and π^{-k} in $O(t)$ time.

Another solution by Mäkinen and Navarro [80] achieves $n \lg n + o(n \lg n)$ bits and supports all operations in $O(\lg n)$ time per element retrieved, including range.

Barbay and Navarro [12] proposed an adaptive representation where they decompose the permutation into subsequences where the values are increasing. They call ρ the number of sequences into which the permutation can be decomposed, and achieve $2n \lg \rho(1+o(1)) + \rho \lg n/\rho + O(\rho)$ bits of space and $O(1 + \lg \rho)$ time for π and π^{-1}. Combining this with the result of Munro et al. [86], they achieve the same time bound for $\pi^{\pm k}$.

The solution by Barbay and Navarro was later improved by Barbay et al. [7]. The adaptive representation for binary relations we present in Chapter 3 is based on this structure. We show an alternative way of deriving the same structure that allows to show how to support range operations.

1.1.6 Binary Relations

Consider a binary relation $\mathcal{R}$ over two sets $[n_1]$ and $[n_2], (n_2 \leq n_1)$, with $t = |\mathcal{R}|$ elements in the relation. We can see $\mathcal{R}$ as a matrix of n_1 rows and n_2 columns with t ones in it. A one in position (i, j) indicates that i relates to j through $\mathcal{R}$.

Barbay et al. [9, 10] considered the following operations:

	Implementation 1	Implementation 2
`rowrank`	$O(\lg\lg n_2 \lg\lg\lg n_2)$	$O(\lg\lg n_2)$
`rowselect`	$O(\lg\lg n_2)$	$O(1)$
`rowcount`	$O(1)$	$O(1)$
`colrank`	$O(\lg\lg n_2)$	$O(\lg\lg n_2 \lg\lg\lg n_2)$
`colselect`	$O(1)$	$O(\lg\lg n_2)$
`colcount`	$O(1)$	$O(1)$
`rel_access`	$O(\lg\lg n_2)$	$O(\lg\lg n_2)$

Table 1.3: Implementations supported by the representation of Barbay et al. [9, 10]. The space requirement is $t(\lg n_2 + o(\lg n_2))$ bits.

- $\mathcal{R}$.`rel_access`(i, j): returns true iff $(i, j) \in \mathcal{R}$.
- $\mathcal{R}$.`rowrank`(i, j): number of ones in row i up to position j (included).
- $\mathcal{R}$.`rowselect`(i, p): p-th one in row i, or ∞ if $\mathcal{R}$.`rowrank`$(i, n_2) < p$.
- $\mathcal{R}$.`rowcount`(i): number of ones in row i.
- $\mathcal{R}$.`colrank`(i, j): number of ones in column j up to position i (included).
- $\mathcal{R}$.`colselect`(p, j): p-th one in column j, or ∞ if $\mathcal{R}$.`colrank`$(n_1, j) < p$.
- $\mathcal{R}$.`colcount`(j): number of ones in column j.
- $\mathcal{R}$.`relrange`(i_1, i_2, j_1, j_2): retrieves the elements $(i, j) \in \mathcal{R}$ such that $i_1 \le i \le i_2$ and $j_1 \le j \le j_2$.

The structure supporting these operations requires $t(\lg n_2 + o(\lg n_2))$ bits of space and supports two different implementations, as shown in Table 1.3.

The representation by Barbay et al. uses rank, select and access operation on sequences to obtain the query complexities shown in Table 1.3. Another important operation that is hard to emulate under their representation is `relrange`. This is essentially the range searching problem in an $n_2 \times n_1$ grid, and corresponds to finding all elements within a given rectangle inside the grid.

Some work for `relrange` has been done for the case $n_1 = n_2$, where Mäkinen and Navarro achieved $n \lg n(1 + o(1))$ bits of space and $O(\lg n)$ time per element reported. Later, Bose et al. achieved $n \lg n(1 + o(1))$ bits of space and $O(\lg n / \lg\lg n)$ time per element retrieved. Their representations are not space optimal, since the information theoretical lower bound for representing $\mathcal{R}$ is $t \lg \frac{n_1 n_2}{t}(1 + o(1))$ bits. Obtaining such bound and offering full functionality is a challenging problem, especially when we consider a wider set of queries as the ones defined in Chapter 3.

1.2 Text Indexes

Text indexes deal with the problem of indexing a single text and answering various queries of interest over it. The most common ones are retrieving a substring of the text between two given positions, called `extract`; searching the positions where a given pattern appears, known as `locate` or `find`; and counting the number of times a pattern appears in the text, `count`.

The operation `extract` might appear to be trivial, but when we move to *self-indexing*, where the index and text are integrated, that operation can be far from trivial.

We discuss four different though related indexes: suffix trees, suffix arrays, compressed suffix arrays and the FM-index. We refer to the text as T, its length as n, and $\Sigma = [\sigma]$ the alphabet, where \$ represents the smallest symbol. We refer to the pattern text as P and its length as p. When answering a `locate` query, we call *occ* the number of occurrences of P in T.

1.2.1 Suffix Trees

A suffix tree is a compressed trie containing all suffixes of T. Since adding one node per symbol could lead to a $\Theta(n^2)$ space requirement, *skips* are incorporated [115, 56, 25]. The skips are values associated to each edge in the tree and represent how many elements should be skipped in before the next node. For example, in Figure 1.2, after following the *a* from the root, all suffixes that continue with a *b* also continue with *bra*, so we add the extra information 3 as the skip.

As all nodes of degree 1 are eliminated by the skips, and there are n leaves, the space requirement of suffix trees is $O(n)$ words with the straight- forward representation. Count takes $O(p)$ time and locate requires an $O(occ)$ extra time for reporting the positions. Using succinct trees, we can represent the tree in $O(n \lg \sigma)$ bits.

1.2.2 Suffix Arrays

The suffix array [82, 56] is an index built on top of text. It consists of one array A of length n. The field $A[i]$ stores the starting position of the i-th lexicographically smaller suffix of T. Figure 1.2 (bottom) shows an example of a suffix array. The suffix array corresponds to listing the positions of the suffix tree in order. The operation `count` takes $O(p \lg n)$ time and `locate` takes $O(p \lg n + occ)$ time, by straight-forward implementations.

The suffix array allows us to emulate many of the operations in the suffix tree, since we can emulate the traversal by paying $O(\lg n)$ per step. This can be improved for searching, where adding the *longest common prefix (LCP)* array allows us to reduce the complexity of searching for a pattern to $O(\lg n + p)$ time [82].

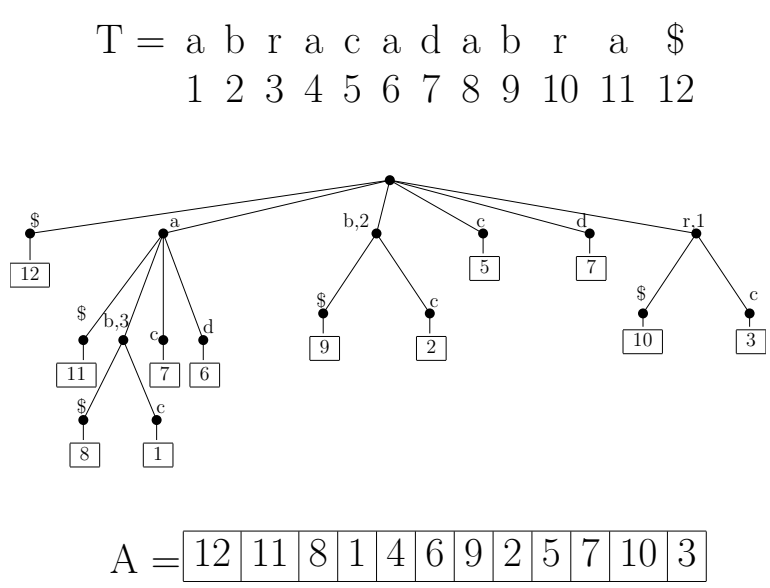

Figure 1.2: Example of a suffix tree and suffix array, corresponding to the array storing the values at the leaves of the suffix tree. Unary paths are compressed to a pair (c,s), where c corresponds to the first symbol of the path, and s to the length of the unary path, also known as skip value.

1.2.3 Compressed Suffix Array (CSA)

Given that the natural representation for a suffix array takes $n \lg n$ bits for a string requiring only $n \lg \sigma$ bits, suffix arrays became a natural target for succinct representation. While variety of compressed representations for the suffix array have been proposed [92]. In this section we focus in the approach of Sadakane [104, 92].

This index works by representing a permutation Ψ efficiently. The permutation can be described in terms of the suffix array as follows:

$$A[\Psi[i]] = \begin{cases} A[i]+1 & \text{if } A[i] < n \\ 1 & \text{otherwise} \end{cases}$$

In simple terms, Ψ at position i points to the place where the suffix starting at position $i+1$ begins.

The Ψ array has interesting properties, for example, it contains σ strictly increasing runs, one for every symbol in the alphabet. This is easy to see, since removing the first symbol in two strings does not change the ordering when they match in their first position. This property also allows for binary searching positions inside ranges corresponding to a given symbol in the suffix array. This is the property exploited in backward search, a method proposed by Sadakane [104] that allows to support pattern matching operations over Ψ.

Solution	Space (bits)	extract	count	locate
Ferragina et al. for $\sigma = O(\text{polylog}\, n)$	$nH_k(T)+o(n)$	$O(p)$	$O(\ell)$	$O(\lg^{1+\epsilon} n)$
Ferragina et al.	$nH_k(T) + o(n\lg\sigma)$	$O(p\frac{\lg\sigma}{\lg\lg n})$	$O(\ell\frac{\lg\sigma}{\lg\lg n})$	$O(\lg^{1+\epsilon} n\frac{\lg\sigma}{\lg\lg n})$
Mäkinen and Navarro	$nH_k(T) + o(n\lg\sigma) + O(\sigma^{k+1}\lg n)$	$O((\ell+\lg^{1+\epsilon} n)\lg\sigma)$	$O(p\lg\sigma)$	$O(\lg^{1+\epsilon} n\lg\sigma)$

Table 1.4: Different tradeoffs achieved for the FM-Index. The time for locate is per occurrence retrieved, and count has to be performed in advance to reporting.

The space requirement of this index is $\frac{1}{\epsilon}nH_0(T)+O(n\lg\lg\sigma)+\sigma\lg\sigma$ bits. The time for count is $O(p\lg^{\epsilon} n)$ and each occurrence reported adds $O(\lg^{\epsilon} n)$ time to locate. The operation extract takes $O(\ell+\lg^{\epsilon} n)$ time.

1.2.4 FM-Index

The FM-Index [45, 46, 92] also uses backward search. The main difference is that the structure makes use of the Burrows-Wheeler transform and represents this permutation of the text [20]. This permutation allows one to capture the regularities of the text in an easy way, and one can support backward search by supporting rank and select operations. Examples of this are the FM-Index combined with the solution of Ferragina et al. [47] that achieves the tradeoff shown in Table 1.4. A much simpler, but slightly less efficient, is the one proposed by Mäkinen and Navarro [80], that combines a wavelet tree [59] with the solution of Raman et al. for bitmaps [101] and achieves the third tradeoff shown in Table 1.4.

There are also dictionary-based self-indexes. The LZ-Index [89, 5] and the SLP-Index [33, 27] are examples of this approach. In particular, the SLP-Index has proved to be very effective in repetitive collections (i.e., the collection contains many copies of small mutations of a base text.) We study this more deeply in Chapter 4.

1.3 Document Retrieval

Given a set of documents $\mathcal{D} = \{T_1, T_2, \ldots, T_m\}$, and a query $\mathcal{Q} = \{P_1, P_2, \ldots, P_q\}$ where $p_i = |P_i|$, we want to obtain the documents in $\mathcal{D}$ containing the patterns in $\mathcal{Q}$. When the query contains only one pattern, we usually refer to it as P and its length as p. We refer to this problem as the document retrieval problem.

Despite its similarity to the pattern matching problem, the document retrieval problem in the general setting has not received as much attention. This is primarily because, in practice, the restrictions imposed by inverted indexes [21] are acceptable for natural language.

The first linear space solution to the document retrieval was proposed by Muthukrishnan in 2002 [88]. He considered several variations:

- Document listing: obtain all documents containing a given query word w.
- Document mining: retrieve all documents where a given query word w appears at least k times.
- Repeats: retrieve all documents containing a given query word w that appears at least twice and within distance at most k inside the document. The distance is measured as the difference between the position of each occurrence.

The work by Muthukrishnan gave a linear space solution (in words) for the document listing problem with complexity $O(p + output)$, where *output* is the size of the output. The same result can be achieved to document mining and repeats if the value of k is fixed. For variable k the space of document mining raises to $O(n \lg K)$ words, where K is the maximum possible value for k. In the case of repeats, the space increases to $O(n \lg n)$ words. For both cases the query time remains optimal.

This result was later improved by Sadakane [106], the space was brought down to $\frac{1}{\epsilon} nH_0(\mathcal{D}) + O(n) + O(m \lg \frac{n}{m})$ bits, compared to the $O(n \lg n)$ bits solution of Muthukrishnan. The query time becomes $O(p + output \lg^{\epsilon} n)$.

Sadakane also considers the problem of reporting the documents and their TF-IDF score. This can be done within twice the space of the previous solution. The time for computing the ranking function adds an $O(output(\lg \lg output))$ term to the total complexity per query.

Sadakane's solution, inspired by the one proposed by Muthukrishnan, makes use of a Range Minimum Query (RMQ) data structure. He achieves constant time queries within $4n + o(n)$ bits, where the structure allows to retrieve the position of the minimum element, but not the element itself. This structure was later improved by Fisher [49], reducing the leading constant from 4 to 2, and improving the construction time of the structure.

The first practical implementation of a structure for text retrieval is due to Välimäki and Mäkinen [112]. They propose a solution that replaces the term $O(m \lg \frac{n}{m})$ by $n \lg m(1 + o(1))$ bits. Under the assumption that σ and m are in $O(\text{polylog}\, n)$, this achieves an interesting tradeoff, since retrieving the documents containing a given pattern P takes $O(p + output)$. They achieve this result by reducing many operations of Muthukrishnan's solution to `rank` and `select` queries over sequences, and use a Wavelet Tree to answer them [59, 47]. In the general case their query time becomes $O(m \lg \sigma + output \lg m)$. This can be improved with newer structures for the `rank` and `select` problem [54, 8, 7].

Välimäki and Mäkinen's implementation shows that these indexes are an interesting alternative to inverted indexes, in particular, when the number of occurrences of a pattern in the collection is large compared to the number of documents reported.

The most general proposal so far was presented by Hon et al. [64]. Their solution, using augmented suffix trees, supports retrieving the top-k documents under any scoring function in

$O(p + k \lg k)$ time using $O(n)$ words of space. They show how to make this structure succinct by replacing the suffix tree by the CSA of Sadakane [104], and sampling the augmented nodes of the suffix tree. The resulting structure supports retrieving the top-k documents in $O(p + k \lg^{4+\epsilon} n)$ time, within $2|CSA| + m \lg n/m + o(n)$. They also include succinct solutions for static rank functions, document mining and document listing problems, offering better tradeoffs than the general solution for succinctly computable scoring functions.

Navarro and Nekrich [93] showed how to index a collection in $O(n)$ words and support top-k queries. In their case they consider retrieving the k documents that contain more occurrences of a given pattern (and also other scoring functions). The query time is optimal, that is, $O(p + k)$, and they can further reduce their space to $O(n(\lg \sigma + \lg m + \lg \lg n))$ bits.

In the last couple of years we have seen many practical solutions for top-k queries [76, 95, 39]. We will not discuss these results, since we will focus mostly on the document listing problem in Chapter 5. Navarro and Valenzuela have a practical result on document listing [94], we compare against it in Chapter 5.

1.4 Roadmap

The topics covered in this document are:

- Chapter 1 covers the basic succinct data structures that appear recurrently as building blocks for more complex solutions. Then, we present related work on compressed text indexes. Later, we cover the document retrieval problem, which is related to text indexes, but addresses a slightly different problem.
- Chapter 2 presents new results for building wavelet trees efficiently. This is joint work with Patrick Nicholson and Diego Seco. The main result shown in this chapter shows how to build a wavelet tree almost in-place in roughly $O(n \lg n \lg \sigma)$ time, and in $O(n \lg \sigma)$ time with only n extra bits of space. Wavelet trees are used in almost every chapter in this thesis, so these results allow for better construction algorithms for the structures presented in this document.
- Chapter 3 presents several representations for binary relations. This is joint work with Jeremy Barbay, J. Ian Munro and Gonzalo Navarro. This chapter presents three main results, one is the definition of several operations over binary relations and reductions among them. The second result is showing how to obtain structures that support these operations efficiently, and in little space. Finally, we show how to build an adaptive representation that supports a restricted set of operations while achieving very little space in some cases.
- Chapters 4 presents grammar compressed text indexes. This chapter is joint work with Gonzalo Navarro. We first present an index for a restricted class of grammar compressed text, called SLPs. Then we extend this result to a more general class, and also improve the query times for searching.

- Chapter 5 presents a solution for the document listing problem on highly repetitive collections, joint work with J. Ian Munro. This uses the results of Chapter 4 to tackle this particular problem. This chapter has more of practical twist compared to the others, we show that our index performs well on Wikipedia historic data, and also on synthetic DNA databases.
- Finally, in Chapter 6 we present some general conclusions to the results presented in this thesis.

All these results are connected to each other. However, Chapter 2 can be studied independently, or skipped, except for Section 2.1 which explains in more detail how wavelet trees are defined and used; this structure is heavily used in this thesis.

Chapter 3 presents results that are useful for representing databases and inverted lists. The ideas discussed are important for understanding all the following chapters.

Chapter 4 presents indexes for several classes of grammar compressors. These indexes take as input a grammar-compressed representation of a sequence and produce a structure that supports the three main operations described for text indexes.

Finally, Chapter 5 can be read independently, even though it combines the results from previous chapters, in particular, it makes heavy use of Theorem 4.10.

2 Wavelet Trees

One of the most elegant generalizations of structures for binary rank, select and access is the *wavelet tree* [59]. Wavelet trees have not only proven to be a crisp theoretical solution, but also perform well in practice [44, 32]. As such, they are used by many practical compressed text indexes, such as the SSA [92], LZ77-Index [77], and SLP-Index (see Chapter 4). Another fact that makes wavelet trees interesting as a structure is that they support richer queries than initially expected. For example, they can be used for representing binary relations (see Chapter 3), discrete set of points [18], and permutations [12] among others. In all of these domains, wavelet trees support a rich set of operations efficiently.

A recent proposal, the *wavelet matrix* [37] (see Section 2.2), maintains all the nice properties of wavelet trees, while offering interesting practical simplifications.

There has been a great deal of study on the performance of different variants of wavelet trees [32], considering the shape and internal representation of the bitmaps [67, 25, 101]. However, not much effort has been devoted into space efficient construction algorithms for wavelet trees.

This chapter presents several new algorithms for constructing wavelet trees, using very little space. This significantly improves upon the naïve construction algorithm, as well as a previously known technique by $O(n \lg \sigma)$ bits.

2.1 The Wavelet Tree

The wavelet tree extends the results for binary rank, select and access to arbitrary alphabets the following way. Every internal node in a wavelet tree has two children (we identify them as left and right). Each node represents a range $R \subseteq [1, \sigma]$ of the alphabet Σ, its left child represents a subset $R_\ell \subset R$ and the right child subset $R_r = R \setminus R_\ell$. Every node represents a subsequence A' of the input sequence A composed of elements whose value are in R. This subsequence is represented by a bitmap of length $|A'|$, and, for each position i in the bitmap, a 0 bit means that position i belongs to R_ℓ and a 1 bit means that it belongs to R_r. The bitmaps inside each node need to support rank and select operations in order for the wavelet tree to operate efficiently.

One can access a position in A by following the path from the root to a leaf guided by the bit representing the position at each level. Let B be the bitmap at the node we are at. When moving to the left child the position i is mapped to $B.\mathtt{rank}(0, i)$, and when moving to the right child the position is mapped to $B.\mathtt{rank}(1, i)$. By similar observations, it is an easy exercise to show that a wavelet tree can support rank, select, and access operations in $O(\lg \sigma)$ time, which is proportional to the height of the tree. Furthermore, the wavelet tree uses $n \lg \sigma + o(n \lg \sigma)$ bits of space, because the bitmaps representing each node use $n \lg \sigma$ space in total, and the little-oh term covers the cost of the auxiliary rank and select structures.

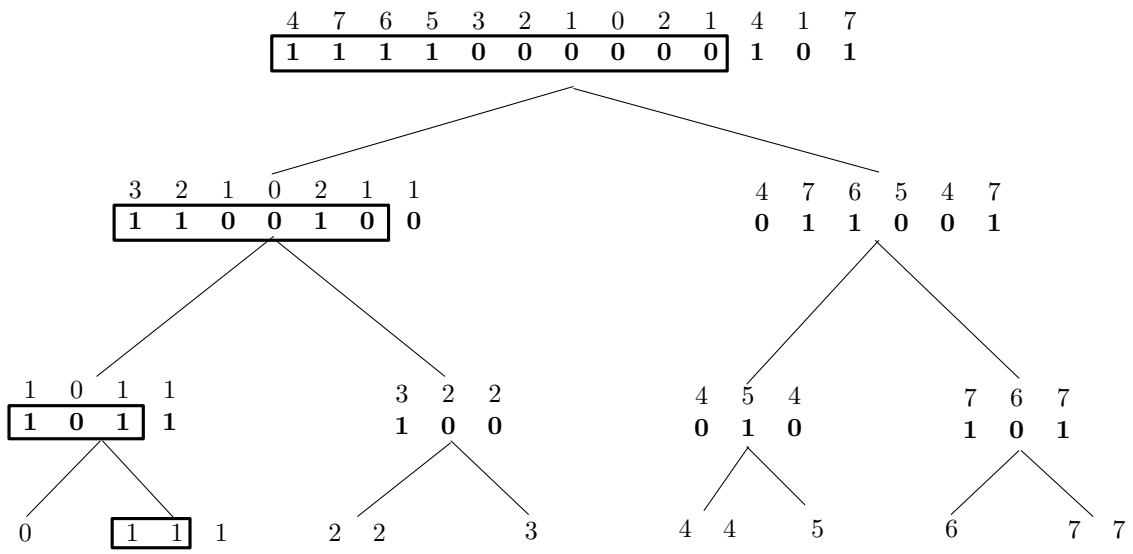

Figure 2.1: Example of a wavelet tree for sequence 4765321021417.

Figure 2.1 shows a wavelet tree for the sequence 4765321021417. We highlight the access operation for position 10, marking with boxes the nodes up to the position we look at in each level. Remember we only have the bitmaps in the internal nodes, the subsequences shown are there just for guidance. We call B_i to the highlighted node at level i. We start at position 10 in B_0 and see a 0. By performing a $B_0.\mathsf{rank}(0,10)$, we get that this element maps to position 6, where we see a 0 again in B_1 and repeat the process. We obtain that the next position, $B_1.\mathsf{rank}(0,6)$, is position 3 on the left child. Finally, here, we see a 1 in B_2, and that determines the symbol *1*. Also, by mapping to the right child, we can determine the rank for that symbol up to position 10.

Wavelet trees were later extended to generalized wavelet trees [47], where the fan out of each node is increased from 2 to $O(\lg^{\epsilon} n)$, increasing the speed of the operations to $O(\lg\sigma/\lg\lg n)$ time [47]. In favour of clarity, we focus mainly on binary wavelet trees, although our results also extend to the generalized version.

A simple construction algorithm works in the following way. First, we build the root of the wavelet tree. To do this, we partition the alphabet Σ into Σ_ℓ and Σ_r according to some balancing rule, and then create a bitmap of length n storing a 0 in position i if $A[i] \in \Sigma_\ell$, and a 1 in position i otherwise. Then we generate a copy of the subsequence of elements whose bit at position i is a 0, and recurse on this subsequence to build the left subtree. Finally, we recurse on the 1 bits to construct the right subtree. It is easy to see that this method is inefficient, since we create a copy of the array at every level, and thus, the space consumed is $O(n\lg^2\sigma)$ bits. This can be improved by realizing that at each level we can re-use the space required by the sequence. The only exception to this is the root, since our application might require us to leave A unmodified after the construction process. This requires $O(n\lg\sigma)$ extra bits, since we must copy A once.

An example of such an application is a library for succinct data structures, such as Libcds [38], where the user might want to further process the sequence used to build the wavelet tree.

2.2 The Wavelet Matrix

The wavelet matrix [37] is a structure inspired by the wavelet tree that allows faster queries in practice by freeing the structure of the restriction of maintaining the nodes of the wavelet tree. Hence, it is called wavelet matrix, we have $\lg \sigma$ sequences of n bits, each one representing a row in the matrix. We refer to the bitmap in row i as M_i, and all these bitmaps are augmented with structures to support rank and select queries.

To simplify the explanation of this structure, we present it in terms of the binary representation of its symbols. The first row is built by keeping the most significant bit of the symbol. We then stably sort the values according to this bit in order to recurse at the next row. At row ℓ, we will split the alphabet in half. The first part consists of the symbols whose ℓ-th bit is a 0 and the second half contains those with a 1 at that position. Then we stably sort the symbols according to the ℓ-th bit before recursing to the next row. An interesting way of looking at this structure is as performing a radix sort on the reversed binary representations of the symbols. In fact, if we reverse all symbols before building the structure, the leaves end up in order. In general, we do not want that, since the construction explained maintains the original nodes of the wavelet tree, supporting geometrical-like queries [37]. It is easy to see that this structure requires $n \lg \sigma + o(n \lg \sigma)$ bits of space.

Data: Text T
Result: Wavelet Matrix
for $i \in [\lceil \log \sigma \rceil, \ldots, 1]$ **do**
  for $j \in [1, n]$ **do**
    $M_{\lceil \log \sigma \rceil - i}[j] \leftarrow i$-th bit of $T[j]$
  Stably sort elements in T according to the i-th bit

Algorithm 1: High-level construction algorithm for the wavelet matrix.

This structure supports access, rank and select following basically the same ideas as for the wavelet tree. For accessing position i, we start by looking at the i-th bit at row 0, M_0, say its value is b. We write this bit down, since it corresponds to the most significant bit of the result, and continue at row 1 at position $M_0.\mathsf{rank}(0, i)$ if $b = 0$ and $M_0.\mathsf{rank}(1, i) + M_0.\mathsf{rank}(0, n)$ otherwise. We keep repeating this same procedure until reaching row $\lg \sigma$, where we finish rebuilding the binary representation of the symbol, retrieving one bit at each row.

Rank is similar, we move as for access, but considering the representation of the symbol we are computing rank for, and ignoring the bit we are looking at each time. In addition to that, we have to keep track of the left boundary of the candidates so far, so as to compute rank as an offset from there. This is keeping the starting position of the original wavelet tree node inside the matrix.

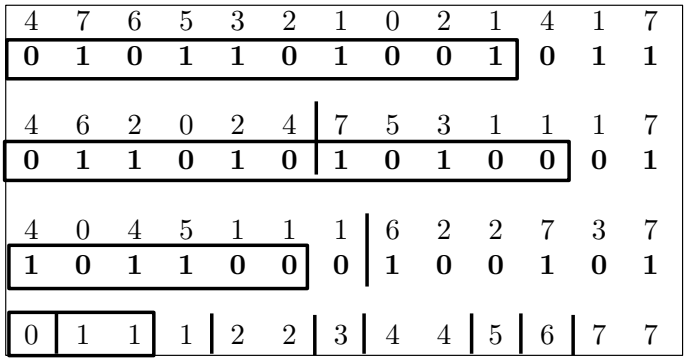

Figure 2.2: Example of a wavelet matrix for sequence 4765321021417.

Select can be easily implemented by performing rank of the symbol at the end of the sequence, identifying the range of values for the symbol at the last row. Then going to the desired occurrence, and tracking its way up the matrix using binary select queries. Reaching row 0 at the desired position.

If we store the number of 0s at each row, we only need one rank operation per row in order to answer access and rank. In the case of wavelet trees, if we represent each level as a single bitmap, we need 2 rank operations [37]. Therefore, the wavelet matrix allows for faster running time in practice with only $O(\lg\sigma \lg n)$ bits of extra space.

Figure 2.2 shows a wavelet matrix for the sequence 4765321021417, reading the bits from least significant to most significant, as this is more intuitive in terms on how the elements appear at the bottom. We use vertical bars to mark the number of 0s in the previous row. We highlight the access operation for position 10, marking with boxes the rows up to the position we look inside each one of them. Remember we only have the bitmaps in each row, the subsequences shown above each row are there just for guidance. We start at position 10 and see a 1. By performing a $M_0.\mathsf{rank}(1,10)$, we get that this element maps to position 5 inside the ones. This is 5 positions after the marker. By repeating the process, we obtain that the next position, $M_1.\mathsf{rank}(0,11)$, is position 6 inside the region of the 0s. Finally, here, we see a 0 at position 6 in M_2, and that determines the symbol *1*.

2.3 Summary of results

In this chapter we present several new algorithms for constructing wavelet trees, and matrices, space efficiently. Throughout this section, we denote A as the input array and T/M as the wavelet tree/matrix. All of our results are for uncompressed structures, that is, the case where the bitmaps of the wavelet tree/matrix occupy $n\lg\sigma$ bits. Thus T occupies $n\lg\sigma + S(n\lg\sigma)$ space, where $S(m)$ denotes the extra space required by auxiliary rank and select structures on a bitmap of length m. Before moving on to discuss the results, we need the following definitions.

We refer to a construction algorithm as *non-destructive* if A is unmodified after T/M has been constructed. If A is modified, we say the construction algorithm is *destructive*.

There are several choices of rank and select structures to use for the bitmaps. Let $C(m)$ denote the time required to construct auxiliary rank and select structures on a bitmap of length m, and $E(m)$ denote the extra space required to construct these structures. We note that in many existing algorithms, $E(n) = O(\lg n)$ extra bits. Given a bitmap $B[0..n-1]$, suppose we construct auxiliary rank and select structures in time $C(p)$ on a prefix of B, $B[0..p]$ where $0 \leq p < n$, such that we can answer rank and select queries in constant time on $B[0..p]$. If we can extend our rank and select structures to support queries on $B[0..q]$ for $p < q < n$ in $C(q-p)$ time, then these rank and select structures can be constructed *incrementally*.

Based on these definitions and assumptions, we have the following results:

1. In Section 2.4 we present a non-destructive algorithm for constructing the wavelet tree T in $O(n \lg \sigma + C(n \lg \sigma))$ time, using $O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits in addition to the space occupied by A and T. This section serves as a warm-up, introducing many of the concepts used later in the chapter.

2. In Section 2.5 we present a destructive algorithm for constructing the wavelet tree T in $O(n \lg n \lg \sigma + C(n \lg \sigma))$ time using $O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits in addition to the space required for T. In other words, this algorithm replaces A with the bitmaps for each node in T. We also present a more practical algorithm that runs in $O(n \lg \sigma + C(n \lg \sigma))$ time and uses $n + O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits in addition to the space required for T. In all of the previous results, we show how to replace the $O(\lg n \lg \sigma)$ bit term in the space bound with $O(\lg n)$ bits, if the rank and select structures for T can be constructed incrementally.

3. In Section 2.6 we show how our results extend to compute the wavelet matrix M in $O(n \lg \sigma \lg n + C(n \lg \sigma))$ time, using $O(\lg n \lg \sigma) + E(n \lg \sigma)$ extra bits in addition to the space occupied by M, and how to reduce the time to $O(n \lg \sigma + C(n \lg \sigma))$ by using n extra bits.

2.4 Encoding Scheme

In this section, we show how to reorder the elements of an array $A[0..n-1]$ according to the bitmap $B[0..n-1]$ representing the root of the wavelet tree. This allows us to construct T without copying the subsequences of A into separate arrays at each level. We refer to this process as *partitioning*. After describing partitioning, we show how to reverse a partitioning step. That is, given the subsequence of elements from the left subtree, A_ℓ, the elements from the right subtree, A_r, and the bitmap representing the root of the wavelet tree, B, we show how to rebuild A. We refer to this process as *merging*. In other words, denote string concatenation by $\cdot$, and let $A' = A_\ell \cdot A_r$. Partitioning is the process of constructing A' from A and B, and merging is the process of reconstructing A from A' and B.

2.4.1 Partitioning

We describe the method implemented in LIBCDS that is a simple way of partitioning the array A given that we can support constant time rank and select queries on the bitmap B [81]. Let n_ℓ denote $B.\mathtt{rank}(0, n-1)$ (i.e., the number of 0s in B). It is easy to see that the bitmap B defines a permutation π on the elements of array A as follows:

$$\pi(i) = \begin{cases} B.\mathtt{rank}(0,i) - 1 & \text{if } B[i] = 0, \\ B.\mathtt{rank}(1,i) - 1 + n_\ell & \text{otherwise.} \end{cases} \tag{2.1}$$

One way of partitioning A is to create an *auxiliary bitmap* Aux of length n, where initially all of the bits are set to 0. We then do a scan of the array A and, for each position $p = 0..n-1$, if $Aux[p] = 0$, we move the element at position p to its corresponding place, position $q = \pi(p)$, and set $Aux[p] = 1$. We repeat this process with the element that was at position q in A until returning to a position q' where $Aux[q'] = 1$ [81]. Following standard terminology [48], we call the positions where $Aux[p] = 0$ the *cycle leaders* of π, and we say that the elements in the cycle starting at position p are *rotated* according to π. Thus, the auxiliary array is used to identify the cycle leaders of π[1].

The procedure just described requires $n + O(\lg n)$ extra bits to identify cycle leaders and performs the partitioning in $O(n)$ time, assuming we can support rank operations on B in constant time. The $O(\lg n)$ term comes from the constant number of pointers need to scan the array and rotate the elements.

Denote $\pi^k(i)$ as π iterated k times starting at position i, that is $\pi^k(i) = \pi^{k-1}(\pi(i))$, for $k > 1$ and $\pi^1(i) = \pi(i)$. If $\pi(i) = i$ then we say position i is a *fixed point*. Let A' denote the array A after it has been partitioned. In order to improve the space requirements, we examine some of the properties of π:

Lemma 2.1. *If $\pi(i) \neq i$ (i.e., position i is not a fixed point), then $\pi^k(i) > n_\ell$ for some $k \geq 1$. Similarly, if $j > n_\ell$, and $\pi(j) \neq j$, then $\pi^k(j) \leq n_\ell$ for some $k \geq 1$.*

Proof. The relative ordering of the elements that are symbols in Σ_ℓ in $A'[0..n_\ell - 1]$ is the same as the relative ordering of these elements in $A[0..n-1]$. Thus, if a rotation begins at an element in Σ_ℓ in position i, it will be moved to a position $0 \leq j < i$. Since the rotation will end at position i since it is a cycle, at some point we must encounter an element in Σ_r. By the definition of π, this element will be moved to a position $j' \geq n_\ell$. The second part of the lemma follows by symmetry. □

Based on the previous lemma, we make the following observation:

Observation 2.1. *Rotating only the cycle leaders in positions where $B[i] = 0$ is sufficient to complete the partitioning of A into A', since every cycle that is not a self-cycle involves elements from both Σ_ℓ and Σ_r.*

[1] Note that the cycle leaders are the lowest element position of each cycle

We now show how to perform the partitioning *without* access to Aux. We continue assuming that $|\Sigma_\ell| < |\Sigma_r|$. If this condition does not hold, we can apply Observation 2.1 symmetrically, considering only positions where $B[i] = 1$.

The idea that allows us to discard Aux is to encode it inside A as we perform the partitioning. We do this by defining an invertible function $f : \Sigma_\ell \to \Sigma_r$. This function exists since $|\Sigma_\ell| < |\Sigma_r|$. We run exactly the same partitioning algorithm described in the beginning of this section, except that, during a rotation, every time we move a symbol $s \in \Sigma_\ell$ at position i to position $\pi(i)$, we write $f(s)$ in position $\pi(i)$ instead. Since we do not have to rotate cycle leaders from Σ_r by Observation 2.1, this encoding step is functionally equivalent to having access to Aux. Every time we encounter an element in Σ_ℓ, that element would have had a 0 in its corresponding position in Aux, and we can ignore elements that either would have a 1 in Aux, or were originally in Σ_r.

```
Data: B, A
Result: A becomes A'
for i ∈ [1,...,n] do
    if A[i] ∈ Σ_ℓ then
        pos ← π(i)
        fin ← i
        aux ← A[i]
        while pos ≠ fin do
            A[pos], aux ← aux, A[pos]
            if A[pos] ∈ Σ_ℓ then
                A[pos] ← f(A[pos])
            pos ← π(pos)
        A[pos] ← aux
        if A[pos] ∈ Σ_ℓ then
            A[pos] ← f(A[pos])
for i ∈ [1,...,n] do
    if B[i] = 0 then
        A[i] = f^{-1}(A[i])
```

Algorithm 2: Permuting algorithm to transform A into A'.

After finishing this process, we need one extra pass to decode the values that are supposed to be in Σ_ℓ. We do this by traversing A' and replacing position i by $f^{-1}(A[i])$ if $i < n_\ell$.

Lemma 2.2. *Given an array A over an alphabet Σ and support for constant time rank and select operations on the bitmap B, we can partition A in-place to generate A' in $O(n)$ time with $O(\lg n)$ extra bits of space.*

2.4.2 Merging

The merging process is just the partitioning process in reverse. We describe this problem in a similar way, using the inverse permutation π^{-1}:

$$\pi^{-1}(i) = \begin{cases} B.\texttt{select}(0, i+1) & \text{if } i < n_l, \\ B.\texttt{select}(1, i+1-n_l) & \text{otherwise.} \end{cases} \tag{2.2}$$

It is easy to see that Lemma 2.1 and Observation 2.1 also hold in the merging case. The only difference is that now elements in Σ_ℓ are rotated to the right, and elements from Σ_r are rotated to the left. Thus, there is at least one element in Σ_ℓ and one in Σ_r for each cycle of length greater than 1. These observations allow us to apply the same method as in Lemma 2.2, obtaining the following lemma.

Lemma 2.3. *Given the array A' and support for constant time rank and select operations on the bitmap B, we can reconstruct A in-place in $O(n)$ time with $O(\lg n)$ extra bits of space.*

Using a stack of size $O(\lg \sigma)$ pointers to keep track of the node in T that we are currently processing, we can recursively apply *partitioning* to A, to construct T. After T is constructed, we can reverse the process by *merging* to recover A. We need to construct the rank and select data structures on the node bitmaps we generate, therefore we pay a time penalty of $C(n \lg \sigma)$ and a space penalty of $E(n \lg \sigma)$ bits.

Theorem 2.1. *There exists a non-destructive algorithm for constructing a wavelet tree T that uses $O(n \lg \sigma + C(n \lg \sigma))$ time, and $O(\lg n \lg \sigma) + E(n \lg \sigma)$ bits beyond the space required for A and T.*

2.4.3 Extension to Generalized Wavelet Trees

The encoding scheme for generalized wavelet trees works in a similar way to that of the binary case. Let us first state the partitioning problem, and then show how the generalization works.

Given an array $A[0, n-1]$ with values in $\Sigma = [0, \sigma - 1]$, and a sequence $S[0, n-1]$ with values in $\Gamma = [0, k-1]$ that partition Σ into k disjoint sets $\Sigma_0, \ldots, \Sigma_{k-1}$, we want to generate an array A' where elements of A are stably sorted by their corresponding value in Γ.

We can describe the re-ordering in A as a permutation:

$$\pi_k(i) = S.\texttt{rank}(j, i) + \left(\sum_{v<j} S.\texttt{rank}(v, n-1) \right) - 1, \text{ where } j = S[i].$$

Lemma 2.4. *π_k is strictly increasing for positions containing the same value in Γ.*

Proof. We can write $\pi_k(i)$ as $S.\texttt{rank}(j, i) + g(j), j = S[i]$, and $S.\texttt{rank}(j, i)$ is strictly increasing in i. □

Lemma 2.5. *Any cycle $\mathcal{C}$ in π_k, such that $|\mathcal{C}| > 1$, contains at least two positions i, j such that $S[i] \neq S[j]$.*

Proof. By contradiction, assume $|\mathcal{C}| > 1$ and all positions $p_i \in \mathcal{C}$ satisfy $S[p_i] = s$ for a fixed s. Let $p = \min \mathcal{C}$. Since all positions in $\mathcal{C}$ point to elements in S whose value is the same, then π_k is strictly increasing in $\mathcal{C}$, thus, there is no p_j such that $\pi_k(p_j) = p$, therefore, $\mathcal{C}$ is not a cycle. □

Now we can present the encoding method. Let $m \in [0, k-1]$ be the index such that $|\Sigma_m| \geq |\Sigma_j|$ for all $j \neq m$. We then generate $f_j : \Sigma_j \to \Sigma_m$ such that ${f_j}^{-1}$ exists. Then, the algorithm for partitioning works in the following way. For each cycle leader, we start rotating the elements iff the position is not in Σ_m. Every time we rotate an element corresponding to partition j, we write down f_j applied to that element.

Once we finish the process, we go through A' fixing the values at position p using ${f_j}^{-1}$, where j is determined by the range we are at, that is, $j = \min\{r | \sum_{v<r} S.\mathsf{rank}(v, n-1) > p\}$.

There is one detail remaining, and this is how to compute $g(j) = \sum_{v<j} S.\mathsf{rank}(v, n-1)$. We can do this by pre-computing all possible answers in linear time. This option requires $O(\sigma \lg n)$ bits of extra space. Another option is to use compressed bitmaps to represent this in $\min(\sigma \lg(n/\sigma), n) + o(n)$ bits, while supporting queries in constant time [101, 85].

Now we can state the partitioning theorem for the generalized wavelet tree.

Theorem 2.2. *We can solve the partitioning problem for the generalized wavelet tree in $O(n\tau)$ time using $\min(\sigma \lg(n/\sigma) + o(n), n + o(n), O(\sigma \lg n)) + O(\lg n)$ bits of extra space. In here τ represents the maximum between the time to answer rank and access in a sequence over an alphabet of size k.*

The reverse process is similar, the permutation π_k^{-1} is defined as follows:

$$\pi_k^{-1}(i) = S.\mathtt{select}(j, i + 1 - g(j)), \text{ where } j = S[i], g(j) = \sum_{v<j} S.\mathsf{rank}(v, n-1)$$

Lemmata 2.4 and 2.5 also apply to π_k^{-1}, since select is strictly increasing for positions that contain the same element. This allows to apply the same encoding technique using the set of functions, the only difference is how the elements are transformed at the end. Instead of counting in which range we are, we just look at S at the position we are at.

Theorem 2.3. *We can solve the merging problem for the generalized wavelet tree in $O(n\tau)$ time using $\min(n \lg(n/\sigma) + o(n), n + o(n), O(\sigma \lg n)) + O(\lg n)$ bits of extra space. In here τ represents the maximum between the time to answer access and select in a sequence over an alphabet of size k.*

Regarding the rank, select, and access times in Theorems 2.2 and 2.3, there are many alternatives [59, 47, 54], in particular, it is possible to adapt the solution by [47] to compute $g()$ in constant time, achieving the following Corollary.

Corollary 2.1. *If $k = O(\text{polylog}\, n)$, we can solve the partitioning and merging problems for the generalized wavelet tree in $O(n)$ time using $O(\lg n)$ bits of extra space.*

2.5 Construction by Permuting Bits

In this section we show how to destructively permute the bits of an input array A, converting them into the bit vectors of a wavelet tree T.

Let B_v represent the bitmap stored at the root v of T, and v_l and v_r represent the left and right children of v. Define $\mathbb{B}(v) = B_v \cdot \mathbb{B}(v_l) \cdot \mathbb{B}(v_r)$. Thus, $\mathbb{B}(v)$ is the concatenation of the bitmap stored at the nodes of T, in depth first order from the root, v. We describe an algorithm for computing $\mathbb{B}(v)$ that works by permuting the bits of A in place. For our purposes in this section, we consider A to be a bitmap of length $n \lg \sigma$. Let ϕ be the permutation that maps A to $\mathbb{B}(v)$; we abuse notation and denote this as $\phi(A) = \mathbb{B}(v)$. We show that ϕ is the composition of 2σ permutations: two permutations, χ and ψ, applied to each node in T.

2.5.1 Overview of Permutations

In the next section we describe the two permutations χ and ψ that correspond to the root v of T. Before specifying the technical details of these permutations, we first briefly outline *what* they do to the array A.

The first permutation χ shifts the most significant bit in A to a prefix of the bitmap, preserving relative order. More precisely, $\chi(A)$ consists of an n bit prefix $B_v[0..n-1]$, representing the most significant bits of each element in A (which *is* the bitmap stored at the root of T), followed by n *truncated characters* of length $\lg \sigma - 1$; the i-th truncated character is $A[i]$ without its most significant bit, for $0 \leq i < n$.

Let $A_t[0..n-1]$ represent the n truncated characters. The second permutation, ψ, partitions $A_t[0..n-1]$ according to the bits $B_v[0..n-1]$. Thus, applying ψ is equivalent to the partitioning step in a standard wavelet tree construction algorithm: if $B_v[i]$ is a 0, then $A_t[i]$ is partitioned to the left, and if $B_v[i]$ is a 1 then $A_t[i]$ is partitioned to the right.

After applying χ and ψ to A, $\psi(\chi(A))$ consists of $B_v[0..n-1]$, which are the first n bits of $\mathbb{B}(v)$, followed by the partitioned truncated characters, which can then be recursively converted into the remaining bits of $\mathbb{B}(v)$ in a depth first manner. In the sequel, we rely heavily on the following result of Fich, Munro and Poblete:

Theorem 2.4 (Theorem 2.2 [48])**.** *Permuting an array of length n, given the permutation and its inverse, can be done in $O(n \lg n)$ time and $O(\lg n)$ additional bits of storage, in the worst case.*

The algorithm that achieves the bound in the theorem is about 20 lines of C++ code. However, in order to apply this theorem we must define the permutations χ and ψ as well as their respective inverses. The next two subsections are devoted to this task.

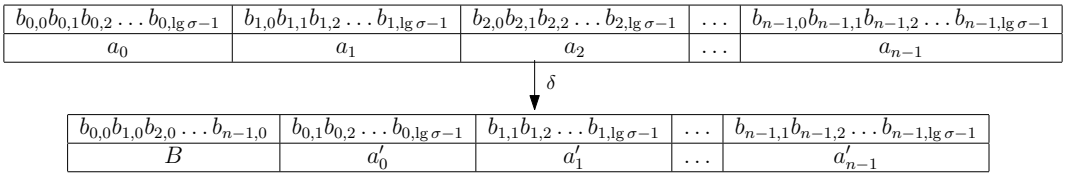

Figure 2.3: Effect of χ on a sequence of symbols $a_0a_1\dots a_{n-1}$.

2.5.2 Chopping the Most Significant Bits

Since A is a bitmap of length $n\lg\sigma$, we refer to individual bits in A. Let $A = b_0,\dots,b_{n\lg\sigma-1}$, where $b_{j\lg\sigma},\dots,b_{(j+1)\lg\sigma-1}$ are the bits in $A[j]$ for $0 \le j < n$, in *decreasing order of significance*[2]. Using this notation, we can now describe $\chi(A,i)$, the i-th bit of $\chi(A)$ in terms of the bits in A, for $0 \le i < n\lg\sigma$. If $\chi(i) = j$, then the i-th bit of $\chi(A)$ is the j-th bit of A.

$$\chi(i) = \begin{cases} \frac{i}{\lg\sigma} & \text{if } \lg\sigma \text{ divides } i, \\ i + n - \left\lfloor \frac{i}{\lg\sigma} \right\rfloor - 1 & \text{otherwise.} \end{cases}$$

Figure 2.3 shows the effect of χ in an array of symbols. Similarly, we can describe the permutation $\chi^{-1}(i)$ as follows:

$$\chi^{-1}(i) = \begin{cases} i\lg\sigma & \text{if } i < n, \\ i - n + \left\lfloor \frac{i-n}{\lg\sigma-1} \right\rfloor + 1 & \text{otherwise.} \end{cases}$$

Running Time: Using χ and χ^{-1} we can apply Theorem 2.4 to A. This allows us to compute $\pi(A)$ in place using $O(n\lg\sigma\lg(n\lg\sigma)) = O(n\lg\sigma(\lg n + \lg\lg\sigma)) = O(n\lg\sigma\lg n)$ time.

We can speedup this process by using word parallelism. The idea is to cut the sequence of bits into blocks of size $s\lg\sigma$ bits, where s fits in a word ($s = \Theta(\lg n)$). For each block we apply χ. This takes

$$O\left(\frac{n\lg\sigma}{s\lg\sigma} s\lg\sigma\lg(s\lg\sigma)\right) = O(n\lg\sigma\lg\lg n)$$

After that, we proceed to move the first cell of size s of each block to the beginning. This is virtually the same permutations as χ, but applied over elements of size s. There are no special boundary cases, since the block size is divisible by s, and adjacent cells will be adjacent in the resulting sequence, so we do not need to respect the symbol boundaries at this stage.

[2]If the characters are stored in increasing order of significance, then we can easily reverse their bits in $O(n\lg\sigma)$ time and $O(\lg n)$ extra bits. If the characters are byte sized, then we can do slightly better with bit tricks [13].

The number of elements we have to permute is $O\left(\frac{n \lg \sigma}{s \lg \sigma}\right) = O(n/s)$, therefore, this second step requires $O(\frac{n}{s} \lg n) = O(n)$ time. The dominating term is the one of applying χ to the blocks. If we spend s extra bits, we can perform the chopping inside each block in $O(s)$ time, by writing the first bit of each symbol in the s extra bits, moving all element to the end of the block (ignoring their first bit), and copying the s bits at the beginning. This allows to cut the time down to $O(n)$ time. By setting $s = \lg n$ we get the following lemma.

Lemma 2.6. *We can apply χ to a sequence of $n \lg \sigma$ bits in $O(n)$ time in the word-RAM model, with word size $\Theta(\lg n)$, using only $O(\lg n \lg \sigma)$ extra bits.*

2.5.3 Partitioning the Truncated Letters

We now describe how to compute $\psi(\chi(A))$ from $\chi(A)$. Note that $\chi(A) = B_v[0..n-1] \cdot A_t[0..n-1]$, and suppose we build a rank and select data structure over B_v, denoted RS; we discuss the space issues associated with this in the next section. The permutation ψ and ψ^{-1} make use of queries to RS in order to determine how to partition $A_t[0..n-1]$.

Not surprisingly, ψ and ψ^{-1} are *almost* identical to the permutations described in Equations 2.1 and 2.2, respectively. The only difference is that we need to account for the fact that the truncated characters A_t begin at an offset of n from the beginning of A, and consist of $\lg \sigma - 1$ bits.

Running Time: As before, using ψ and ψ^{-1} we can apply Theorem 2.4 to $\chi(A)$. Observe that we can easily swap $\lg \sigma - 1$ bit elements in constant time, assuming the word size is $\Omega(\lg \sigma)$. This allows us to compute $\psi(\chi(A))$ in place using $O(n \lg n)$ time.

2.5.4 Overall Requirements

Running Time We must apply χ and ψ to each node in T in order to construct $\phi(A)$. This means that our overall running time is $T(n, \lg \sigma) = T(n_l, \lg \sigma - 1) + T(n - n_l, \lg \sigma - 1) + O(n \lg \sigma + n \lg n)$, where $T(n, 1) = O(1)$. Since the height of the tree is bounded in terms of $\lg \sigma$ rather than n, $T(n, \lg \sigma) = O(n \lg n \lg \sigma)$.

Extra Space As discussed in Section 2.5.3, at each node v in T we construct auxiliary rank and select structures for the bitmap of length $m \leq n$, associated with v. Let $S(m)$ denote the number of bits required for the auxiliary structures, and $E(m)$ denote the number of bits required for their construction. The auxiliary structures for T require $S(n \lg \sigma) + E(n \lg \sigma)$ extra bits, since $\mathbb{B}(v)$ is a bitmap of length $n \lg \sigma$. Furthermore, we can release the memory used by the auxiliary structures for each $v \in T$ after we have applied the permutations to v. Thus, we can avoid using extra space for the auxiliary structures associated with v, with careful memory management.

In addition to the $O(1)$ extra pointers required by Theorem 2.4, we need a stack of size $O(\lg \sigma)$ pointers in order to remember our current location within T. However, we can get rid

of the stack at the cost of some complexity. Suppose $w_1, ..., w_\sigma$ are the nodes of T in depth-first order. Then by storing only n and the bits representing the path to w_i, we can compute the offset and length of the bitmap representing w_{i+1} in $O(\lg \sigma)$ time using the rank and select structures constructed for $w_1, ..., w_i$. Note that $w_1, ..., w_i$ represent a contiguous prefix of the bitmap $\mathbb{B}(v)$. Thus, if we can construct rank and select structures for $\mathbb{B}(v)$ incrementally, we can discard the stack.

Trade off If we have an extra n bits available to us, we can apply χ and ψ to each node in T in $O(n)$ time per level of T, using an auxiliary bitmap of length n. To apply χ, we use the result from Lemma 2.6. To apply ψ we use the auxiliary bitmap to store the cycle leaders, as described in Section 2.4. With the n extra bits, we can compute $\phi(A)$ in $O(n \lg \sigma)$ time overall.

Theorem 2.5. *Suppose we are given an array A of n symbols drawn from an alphabet of size σ. Let $C(m)$ denote the time required to construct auxiliary structures on a bitmap of length m, $E(m)$ denote the extra bits required to construct these structures, and $S(m)$ denote the total number of bits occupied by these structures. We can permute the $n \lg \sigma$ bits of A, replacing A with the bitmaps of a wavelet tree T occupying $n \lg \sigma + S(n \lg \sigma)$ bits in:*

1. *$O(n \lg n \lg \sigma + C(n \lg \sigma))$ time using $O(\lg \sigma \lg n) + E(n \lg \sigma)$ extra bits beyond the space occupied by T.*
2. *$O(n \lg \sigma + C(n \lg \sigma))$ time, and using $n + O(\lg n \lg \sigma) + E(n \lg \sigma)$ extra bits beyond the space occupied by T.*

In both of the previous results, we can replace the $O(\lg n \lg \sigma)$ bit term in the space bound with $O(\lg n)$ bits, if we can construct the rank and select structures for T incrementally.

2.5.5 Building Generalized Wavelet Trees

The previous result can also be extended to generalized wavelet trees. There is one necessary assumption, which is that the groups are determined by a fixed size prefix of the binary representation of each symbol, and that this prefix size divides $\lg \sigma$. This makes it easy to define the permutations described before. We call them χ_k and ψ_k. Assuming that we use the r most significant bits of each symbol as their group, their definition is as follows:

$$\chi_k(i) = \begin{cases} r\lfloor i / \lg \sigma \rfloor + j & \text{if } \lg \sigma \text{ divides } w \text{ and } i = w + j, \text{ where } j < r \\ i + r\left(n - \left\lfloor \frac{i}{\lg \sigma} \right\rfloor - 1\right) & \text{otherwise.} \end{cases}$$

Similarly, we can describe the permutation $\chi_k^{-1}(i)$ as follows:

$$\chi_k^{-1}(i) = \begin{cases} \lg \sigma \left\lfloor \frac{i}{r} \right\rfloor + (i \mod r) & \text{if } i < rn, \\ i - rn + \left\lfloor \frac{i - rn}{\lg \sigma - r} \right\rfloor + r & \text{otherwise.} \end{cases}$$

Using χ_k and χ_k^{-1}, we can apply the same technique as for the binary wavelet tree. But we cannot speed up the truncating procedure by taking $s = \lg n$ elements at the time, since sr may be $\omega(\lg n)$. If we take $s = (\lg n)/r$ the overall process takes $O(n + \frac{rn}{\lg n} \lg(rn)) = O(rn)$ time.

From this, we can state the construction corollary, based on Theorem 2.5, for the generalized case:

Corollary 2.2. *Suppose we are given an array A of n symbols drawn from an alphabet of size $\lg \sigma$. Let $C(m)$ denote the time required to construct auxiliary structures on a bitmap of length m, $E(m)$ denote the extra bits required to construct these structures, and $S(m)$ denote the total number of bits occupied by these structures. We can permute the $n \lg \sigma$ bits of A, replacing A with the bitmaps of the generalized wavelet tree T, when the branching factor is of the form $r = 2^x - 1$, occupying $n \lg \sigma + S(n \lg \sigma)$ bits in:*

1. *$O(n \lg n \lg_r \sigma + nr \lg_r \sigma + C(n \lg \sigma))$ time using $O(\lg \sigma \lg n) + E(n \lg \sigma)$ extra bits beyond the space occupied by T.*
2. *$O(nr \lg_r \sigma + C(n \lg \sigma))$ time, and using $n + O(\lg n \lg \sigma) + E(n \lg \sigma)$ extra bits beyond the space occupied by T.*

In both of the previous results, we can replace the $O(\lg n \lg \sigma)$ bit term in the space bound with $O(\lg n)$ bits, if we can construct the rank and select structures for T incrementally.

2.6 Constructing Wavelet Matrices

Wavelet matrices are easier to build, since we only need to build the first level of the wavelet tree, and then recurse on the whole remaining chunk.

We will call $B(A)$ the bitmap the most significant bit of each element in A, and $\bar{C}(A)$ result bitsequence resulting from removing the most significant bit for each element in A. The encoding for the wavelet matrix is defined as $M(A) = B(A) \cdot M(\pi(\bar{C}(A))$.

We know how to compute B in linear time using $O(\lg n \lg \sigma)$ bits. And the partitioning can be performed in $O(n \lg n)$ time within that same space. This allows to build the whole structure in $O(n \lg n \lg \sigma)$ time.

We can also make use of n extra bits to achieve $O(n \lg \sigma)$ time, therefore, we can state our final corollary, obtained directly from Theorem 2.5.

Corollary 2.3. *Suppose we are given an array A of n symbols drawn from an alphabet of size $\lg \sigma$. Let $C(m)$ denote the time required to construct auxiliary structures on a bitmap of length m, $E(m)$ denote the extra bits required to construct these structures, and $S(m)$ denote the total number of bits occupied by these structures. We can permute the $n \lg \sigma$ bits of A, replacing A with the bitmaps of a wavelet matrix M occupying $n \lg \sigma + S(n \lg \sigma)$ bits in:*

1. *$O(n \lg n \lg \sigma + C(n \lg \sigma))$ time using $O(\lg \sigma \lg n) + E(n \lg \sigma)$ extra bits beyond the space occupied by T.*

2. *$O(n \lg \sigma + C(n \lg \sigma))$ time, and using $n + O(\lg n \lg \sigma) + E(n \lg \sigma)$ extra bits beyond the space occupied by T.*

In both of the previous results, we can replace the $O(\lg n \lg \sigma)$ bit term in the space bound with $O(\lg n)$ bits, if we can construct the rank and select structures for T incrementally.

2.7 Concluding Remarks

In this chapter, we have introduced two novel algorithms for constructing wavelet trees that use little space on top of the input sequence. These algorithms are motivated by libraries such as Libcds that cannot destroy the input sequence. The results have direct applications to the construction of the structures presented in Chapters 3, 4, and 5.

It would be interesting to determine if our permutation based construction algorithm can be adapted to create Huffman shaped wavelet trees, where the resulting wavelet tree is not a permutation of the input bits anymore.

We can also achieve compression by building the bitmaps using Raman, Raman and Rao's proposal [101]. This structure for representing bitmaps can be built incrementally. We would have to keep track of the bits that are not used to later move them back to the end and free that space. This is easy to do in our first variant that used n extra bits, but more challenging in the permuting case.

It would also be interesting to find other succinct data structures that can be constructed space efficiently by permuting individual bits, since those structures would be good candidates for applying the techniques presented in this chapter.

3 Binary Relations

Binary relations appear everywhere in Computer Science. Graphs, trees, inverted indexes, strings and permutations are just a few of the examples. They also arise as a tool to complement existing data structures (such as trees [9] or graphs [6]) with additional information, such as weights or labels on the nodes or edges, that can be indexed and searched. Interestingly, the data structure support for binary relations has not undergone a systematic study, but rather one triggered by particular applications. This chapter presents a systematic study of such structures, exploring five different proposals from our framework's point of view.

A binary relation $\mathcal{R}$ relates *objects* in $[1,n]$ with *labels* in $[1,\sigma]$, containing t pairs of the $n\sigma$ possible ones. We focus on space-efficient representations considering a simple zeroth order entropy measure,

$$H(\mathcal{R}) \;=\; \lg\binom{n\sigma}{t} \;=\; t\lg\frac{n\sigma}{t} + O(t)$$

bits, which ignores any other possible regularity, such as, similarity among the sets of labels related to different objects [26]. Figure 3.1 illustrates a binary relation (we identify labels with rows and objects with columns henceforth).

Previous work focused on relatively basic primitives for binary relations: extract the list of all the labels associated with an object or of all the objects associated with a label (an operation called `access`), or extracting the j-th such element (an operation called `select`), or counting how many of these are there up to some object/label value (called operation `rank`).

The first representation specifically designed for binary relations [9] supports `rank`, `select` and `access` on the rows (labels) of the relation, for the purpose of supporting faster joins on labels. The idea is to write the labels of the pairs in object-major order and to operate on the resulting string plus some auxiliary data. This approach was extended to support more general operations needed for text indexing [36]. The first technique [9] was later refined [10] into a scheme that allows one to compress the string while still supporting the basic operations on both labels and objects. The idea is to store auxiliary data on top of an arbitrary representation of the binary relation, which can thus be compressed. This was used to support labeled operations on planar and quasi-planar labeled graphs [6].

Ad hoc compressed representations for inverted lists [116] and Web graphs [35, 29, 19] can also be considered as supporting binary relations. The idea here is to write the objects of the pairs, in label-major order, and to support extracting substrings of the resulting string, that is, little more than `access` on labels. One can add support for `access` on objects by means of string `select` operations [34]. The string can be compressed by various means depending on the application.

In this chapter we aim at settling the foundations of efficient compact data structures for binary relations. In particular, we address the following points:

	1	2	3	4	5	6	7	8	9
A	.	.	1	.	.	.	.	.	.
B	.	.	.	.	.	1	1	.	.
C	.	.	.	1	.	1	.	1	.
D	.	1	.	.	.	.	.	.	.
E	1	.	.	1	1	.	.	.	.
F	.	.	.	.	.	.	.	.	1
G	.	.	.	.	1	.	1	.	.
H	1	1	.	.	.	.	.	.	.

Figure 3.1: An example of binary relation.

- We define a large set of operations of relevance to binary relations, widely extending the classic set of `rank`, `select` and `access`. We give a number of reductions among operations in order to define a core set that allows one to efficiently support the extended set of operations.
- We explore the power of the reduction to string operators [9] when the operations supported on the string are limited to `rank`, `select` and `access`. This structure is called BinRel-Str, and we show it achieves interesting bounds only for a reduced set of operations.
- We show that a particular string representation, the wavelet tree (see Chapter 2), although not being the fastest, provides native support for a much wider set of operations in logarithmic time. We call BinRel-WT this binary relation representation.
- We extend wavelet trees to generalized wavelet trees [47], and design new algorithms for various operations that take advantage of their larger fan-out. As a result we speed up most of the operations within the same space. This structure is called BinRel-GWT.
- We further analyze a structure called *binary relation wavelet tree* (BRWT), that is tailored to represent binary relations [26]. Although the BRWT gives weaker support to the operations, it is the only one that approaches the entropy space $H(\mathcal{R})$ within a multiplicative factor (of 1.272).

For the sake of brevity, we aim at the simplest description of the operations, ignoring any practical improvement that does not make a difference in terms of asymptotic time complexity, or trivial extensions such as interchanging labels and objects to obtain other space/time tradeoffs.

3.1 Operations on Binary Relations

3.1.1 Formal Definition of Operations

We formally define our set of operations. Figure 3.2 graphically illustrates some of them.

- $\texttt{rel\_access}(\alpha,\beta,x,y) = \{(\gamma,z) \in \mathcal{R},\ \gamma \in [\alpha,\beta] \ \wedge\ z \in [x,y]\}$
- $\texttt{rel\_select\_label\_major}(\alpha,j,x,y) =$ j-th smallest pair of $\texttt{rel\_access}(\alpha,\sigma,x,y)$ in order $(\alpha,x) \le (\beta,y) \Leftrightarrow \alpha < \beta \vee (\alpha = \beta \wedge x \le y)$
- $\texttt{rel\_min\_label\_major}(\alpha,x,y,z) =$ under the same order, smallest pair of $\texttt{rel\_access}(\alpha,\alpha,z,y) \cup \texttt{rel\_access}(\alpha{+}1,\sigma,x,y)$
- $\texttt{rel\_select\_object\_major}(\alpha,\beta,x,j) =$ j-th smallest pair of $\texttt{rel\_access}(\alpha,\beta,x,n)$ in order $(\alpha,x) \le (\beta,y) \Leftrightarrow x < y \vee (x = y \wedge \alpha \le \beta)$
- $\texttt{rel\_min\_object\_major}(\alpha,\beta,\gamma,x) =$ under the same order, smallest pair of $\texttt{rel\_access}(\gamma,\beta,x,x) \cup \texttt{rel\_access}(\alpha,\beta,x{+}1,n)$
- $\texttt{label\_access}(\alpha,\beta,x,y) = \{\gamma,\ \exists z,\ (\gamma,z) \in \texttt{rel\_access}(\alpha,\beta,x,y)\}$
- $\texttt{label\_access1}(\alpha,\beta,x) = \texttt{label\_access}(\alpha,\beta,x,x)$
- $\texttt{label\_select}(\alpha,j,x,y) =$ j-th smallest label of $\texttt{label\_access}(\alpha,\sigma,x,y)$
- $\texttt{label\_select1}(\alpha,j,x) = \texttt{label\_select}(\alpha,j,x,x)$, or $\texttt{rel\_select\_label\_major}(\alpha,j,x,x)$
- $\texttt{label\_min}(\alpha,x,y) = \texttt{label\_select}(\alpha,1,x,y)$
- $\texttt{label\_min1}(\alpha,x) = \texttt{label\_min}(\alpha,x,x)$, or $\texttt{label\_select1}(\alpha,1,x)$
- $\texttt{object\_access}(\alpha,\beta,x,y) = \{z,\ \exists\gamma,\ (\gamma,z) \in \texttt{rel\_access}(\alpha,\beta,x,y)\}$
- $\texttt{object\_access1}(\alpha,x,y) = \texttt{object\_access}(\alpha,\alpha,x,y)$
- $\texttt{object\_select}(\alpha,\beta,x,j) =$ j-th smallest object of $\texttt{object\_access}(\alpha,\beta,x,n)$
- $\texttt{object\_select1}(\alpha,x,j) = \texttt{object\_select}(\alpha,\alpha,x,j)$, or $\texttt{rel\_select\_object\_major}(\alpha,\alpha,x,j)$
- $\texttt{object\_min}(\alpha,\beta,x) = \texttt{object\_select}(\alpha,\beta,x,1)$
- $\texttt{object\_min1}(\alpha,x) = \texttt{object\_min}(\alpha,\alpha,x)$, or $\texttt{object\_select1}(\alpha,x,1)$
- $\texttt{rel\_count}(\alpha,\beta,x,y) = |\texttt{rel\_access}(\alpha,\beta,x,y)|$
- $\texttt{rel\_rank}(\alpha,x) = \texttt{rel\_count}(1,\alpha,1,x)$
- $\texttt{rel\_rank\_label\_major}(\alpha,x,y,z) = \texttt{rel\_count}(1,\alpha{-}1,x,y){+}\texttt{rel\_count}(\alpha,\alpha,x,z)$

- rel_rank_object_major(α,β,γ,x) = rel_count$(\alpha,\beta,1,x-1)$+rel_count(α,γ,x,x)
- label_count(α,β,x,y) = |label_access(α,β,x,y)|
- label_rank(α,x,y) = label_count$(1,\alpha,x,y)$
- label_rank1(α,x) = label_rank(α,x,x), or rel_count$(1,\alpha,x,x)$
- object_count(α,β,x,y) = |object_access(α,β,x,y)|
- object_rank(α,β,x) = object_count$(\alpha,\beta,1,x)$
- object_rank1(α,x) = object_rank(α,α,x), or rel_count$(\alpha,\alpha,1,x)$

3.1.2 Motivation for operations

We now motivate the set of operations we defined for binary relations. The full list, formal definition, and illustration are given in Section 3.1.1.

One of the most pervasive examples of binary relations are directed graphs, which are precisely binary relations between a vertex set V and itself. Extracting rows or columns in this binary relation supports direct and reverse navigation from a node. To support powerful direct access to rows and columns we define operations object_access1(α,x,y), which retrieves the objects in $[x,y]$ related to label α, in arbitrary order, and the symmetric one, label_access1(α,β,x). If we want to retrieve the pairs in order, object_min1(α,x), which gives the first object $\geq x$ related to label α, can be used as an iterator, and similarly label_min1(α,x). These operations are also useful to find out whether the link (α,x) exists. Note that adjacency list representations only support efficiently the retrieval of direct neighbors, and adjacency matrix only support efficiently the test for the existence of a link.

Web graphs, and their compact representation supporting navigation, have been a subject of intense research in recent years [17, 19, 35] (see many more references therein). In a Web graph, the nodes are Web pages and the edges are hyperlinks. Nodes are usually sorted by URL, which not only gives good compression but also makes ranges of nodes correspond to domains and subdirectories[1]. For example, counting the number of connections between two ranges of nodes allows estimating the connectivity between two domains. This count of points in a range is supported by our operation rel_count(α,β,x,y), which counts the number of related pairs in $[\alpha,\beta]\times[x,y]$. The individual links between the two domains can be retrieved, in arbitrary order, with operation rel_access(α,β,x,y). In general, considering domain ranges enables the analysis and navigation of the Web graph at a coarser granularity (e.g., as a graph of hosts, or institutions). Our operations object_access(α,β,x,y), which gives the objects in $[x,y]$ related to a label in $[\alpha,\beta]$, extends object_access1 to ranges of labels, and similarly label_access(α,β,x,y)

[1] More precisely, we have to sort by the reversed site string concatenated with the path.

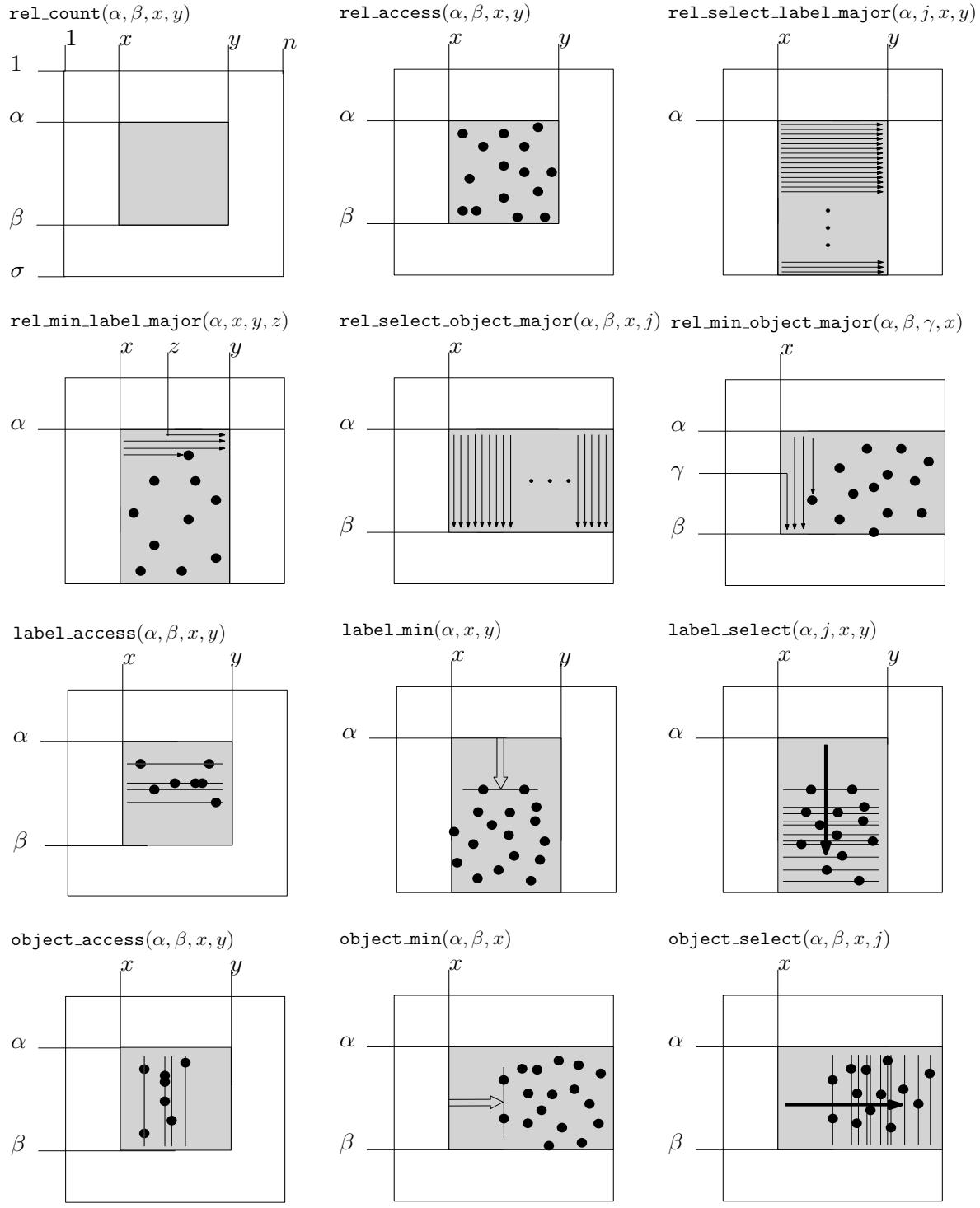

Figure 3.2: Some operations illustrated.

extends label_access1. Ordered enumeration and (coarse) link testing are supported by operations object_min(α,β,x), which gives the first object $\geq x$ related to a label in $[\alpha,\beta]$, and similarly label_min(α,x,y) for labels.

A second pervasive example of a binary relation is formed by the two-dimensional grids, where objects and labels are simply coordinates, and where pairs of the relation are points at those coordinates. Grids arise in Geographic Information Systems and many other geometric applications. Operation rel_count allows us to count the number of points in a rectangular area. A second essential operation in these applications is to retrieve the points from such an area. If the retrieval order is not important, rel_access is sufficient. Otherwise, operation rel_access_label_major(x,y,α,z) serves as an iterator to retrieve the points in label-major order. It retrieves the first point in $[\alpha,\alpha]\times[z,y]$, if any, and otherwise the first point in $[\alpha{+}1,\sigma]\times[x,y]$. Operation rel_access_object_major(α,β,γ,x) is similar, for object- major order. For an even more sophisticated processing of the points, rel_select_label_major(α,j,x,y) and rel_select_object_major(α,β,x,j) give access to the j-th element in such lists.

Grids also arise in more abstract scenarios. For example, several text indexing data structures [23, 36, 68, 80] resort to a grid, which relates for example text suffixes (in lexicographic order) with their text positions, or phrase prefixes and suffixes in Lempel-Ziv compression, or two labels that form a rule in grammar-based compression, etc. The operations most commonly needed are, again, counting and retrieving (in arbitrary order) the points in a rectangle. One example is presented in Chapter 4, and later extended in chapter 5.

Another important example of binary relations is inverted indexes [116], which support word-based searches on natural language text collections. Inverted indexes can be seen as a relation between vocabulary words (the labels) and the documents where they appear (the objects). Another popular operation is the conjunctive query (e.g., in Google-like search engines), which retrieves the documents where k given words appear. These are solved using a combination of the complementary queries object_rank1(α,x) and object_select1(α,x,j) [9, 11]. The first operation counts the number of points in $[\alpha,\alpha]\times[1,x]$, and the second gives the j-th point in $[\alpha,\alpha]\times[x,n]$.

Extending these operations to a range of words allows for stemmed and/or prefix searches (by properly ordering the words), which are implemented using object_rank(α,β,x) and object_select(α,β,x,j), extending object_rank1 and object_select1 to ranges of labels. Extracting a column, on the other hand (label_access1), gives important *summary* information on a document: the list of its distinct words. Intersecting columns (using the symmetric operations label_rank1(α,x) and label_select1(α,x,j)) allows for analysis of content between documents (e.g., plagiarism or common authorship detection). Handling ranges of documents (supported with the symmetric operations label_access, label_rank(α,x,y), and label_select(α,j,x,y)) allows for considering hierarchical document structures such as XML or file systems (where one operates over a whole subtree or subdirectory).

Similar representations are useful to support join operations on relational databases and, in combination with data structures for ordinal trees, to support multi-labeled trees, such as those featured by semi-structured documents (e.g., XML) [9]. A similar technique [6] combining

various data structures for graphs with binary relations yields a family of data structures for edge-labeled and vertex-labeled graphs that support labeled operations on the neighborhood of each vertex. For example, operations `rel_min_label_major` and `rel_min_object_major` support the search for the highest neighbor of a point, when the binary relation encodes the levels of points in a planar graph representing a topography map [6].

The extension of those operations to the union of labels in a given range allows them to handle more complex queries, such as conjunctions of disjunctions. For example, in a relational database, consecutive labels may represent a range of parameter values (e.g., people of age between 20 and 40).

We define other operations for completeness: `rel_rank_label_major` acts like the inverse of `rel_select_label_major`, and similarly `rel_rank_object_major`; `label_count` and `object_count` are more complete versions of `label_rank` and `object_rank`; and `rel_rank` is a more basic version of `rel_count`.

3.1.3 Reductions among operations

We give a set of reductions among the operations introduced. The results are sumarized in the following theorem.

Theorem 3.1. *For any solid arrow op → op' in Figure 3.3, it holds that if op is solved in time t, then op' can be solved in time $O(t)$. For the dotted arrows with associated penalty factors $O(t')$, it holds that if op is solved in time t, then op' can be solved in time $O(tt')$.*

Proof. Several reductions are immediate from the definition of the operations in 3.1.1 (those arrows are in bold in Figure 3.3). We prove now the other ones.

- `rel_rank` → `rel_count`

$$\begin{aligned}\texttt{rel\_count}(\alpha,\beta,x,y) &= \texttt{rel\_rank}(\beta,y)-\texttt{rel\_rank}(\alpha-1,y)\\ &\quad -\texttt{rel\_rank}(\beta,x-1)+\texttt{rel\_rank}(\alpha-1,x-1).\end{aligned}$$

- `rel_rank_label_major` → `rel_count`

$$\begin{aligned}\texttt{rel\_count}(\alpha,\beta,x,y) &= \texttt{rel\_rank\_label\_major}(\beta,x,y,y)\\ &\quad -\texttt{rel\_rank\_label\_major}(\alpha-1,x,y,y).\end{aligned}$$

- `rel_rank_object_major` → `rel_count`

$$\begin{aligned}\texttt{rel\_count}(\alpha,\beta,x,y) &= \texttt{rel\_rank\_object\_major}(\alpha,\beta,\beta,y)\\ &\quad -\texttt{rel\_rank\_object\_major}(\alpha,\beta,\beta,x-1).\end{aligned}$$

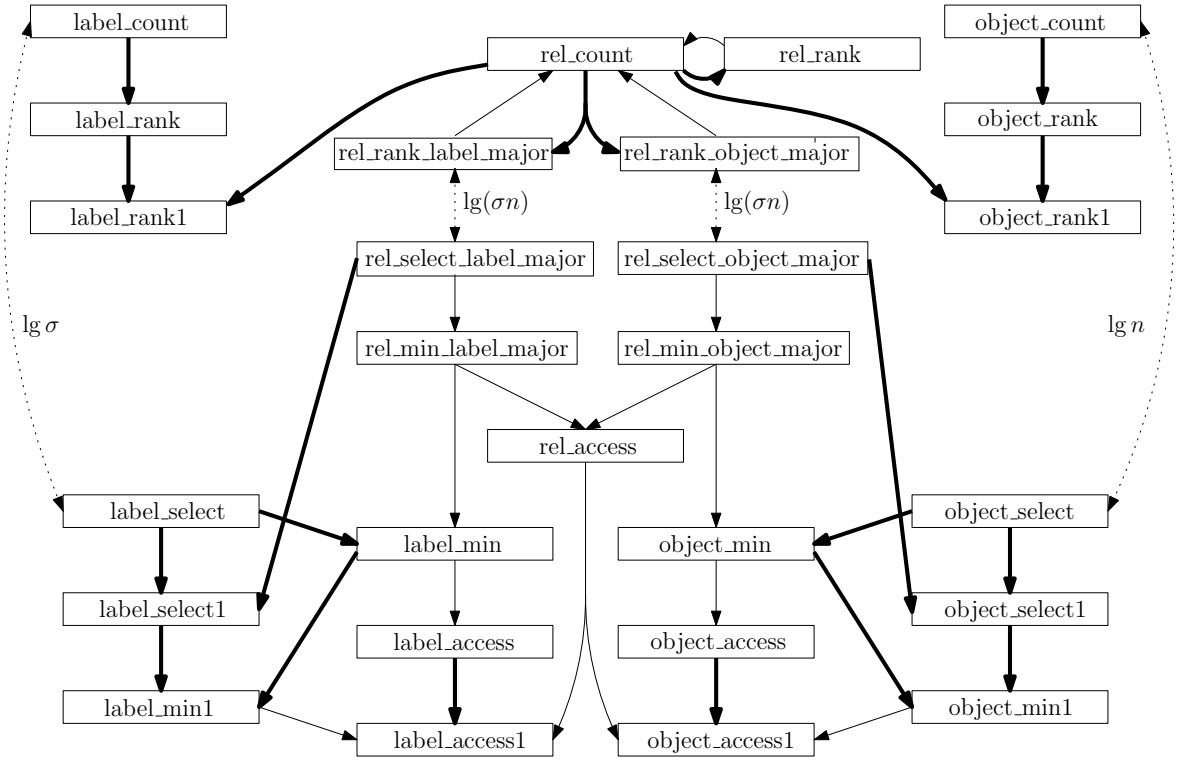

Figure 3.3: Reductions among operations.

- `rel_access` $\rightarrow$ (`label_access1`, `object_access1`)

$$\begin{aligned} \texttt{label\_access1}(\alpha,\beta,x) &= \{\gamma,\ (\gamma,x) \in \texttt{rel\_access}(\alpha,\beta,x,x)\}, \\ \texttt{object\_access1}(\alpha,x,y) &= \{z,\ (\alpha,z) \in \texttt{rel\_access}(\alpha,\alpha,x,y)\}. \end{aligned}$$

- `rel_select_label_major` $\rightarrow$ `rel_min_label_major`: in order to solve query $\texttt{rel\_min\_label\_major}(\alpha,x,y,z)$ we first test if $\texttt{rel\_select\_label\_major}(\alpha,1,z,y)$ gives a pair of the form (α,w), in which case we return it. Otherwise, we return $\texttt{rel\_select\_label\_major}(\alpha+1,1,x,y)$.

- `rel_select_object_major` $\rightarrow$ `rel_min_object_major`: in order to solve query $\texttt{rel\_min\_object\_major}(\alpha,\beta,\gamma,x)$ we first test if $\texttt{rel\_select\_object\_major}(\gamma,\beta,x,1)$ gives a pair of the form (δ,x), in which case we return it. Otherwise, we return $\texttt{rel\_select\_object\_major}(\alpha,\beta,x+1,1)$.

- `rel_min_label_major` $\rightarrow$ `rel_access`: to solve $\texttt{rel\_access}(\alpha,\beta,x,y)$, we find a first point $(\gamma,z) = \texttt{rel\_min\_label\_major}(\alpha,x,y,x)$. The next element is obtained as $(\gamma',z') = \texttt{rel\_min\_label\_major}(\gamma,x,y,z+1)$ and so on, until we reach the first answer with label greater than β.

- rel_min_object_major → rel_access: to solve rel_access(α,β,x,y), we find a first point (γ,z) = rel_min_object_major(α,β,α,x). The next element is obtained as (γ',z') = rel_min_object_major$(\alpha,\beta,\gamma+1,z)$ and so on, until we reach the first answer with label greater than β.

- rel_min_label_major → label_min: let (γ,z) = rel_min_label_major(α,x,y,x), then label_min$(\alpha,x,y)=\gamma$.

- rel_min_object_major → object_min: let (γ,z) = rel_min_object_major(α,β,α,x), then object_min$(\alpha,\beta,x)=z$.

- label_min → label_access: we report γ = label_min(α,x,y), γ' = label_min$(\gamma+1,x,y)$, and so on until reaching a result larger than β. The points reported form label_access(α,β,x,y).

- label_min1 → label_access1: similar to the previous reduction.

- object_min → object_access: we report z = object_min(α,β,x), z' = object_min$(\alpha,\beta,z+1)$, and so on until reaching a result larger than y. The points reported form object_access(α,β,x,y).

- object_min1 → object_access1: similar to the previous reduction.

Finally, the non-constant time reductions are explained the following way:

- (rel_rank_label_major, rel_select_label_major) works in both ways by doing a binary search over the results of the other operation, in the worst case considering $n\sigma$ elements.

- (label_count, label_select) operates the same way as the previous one, but searching among σ elements in the worst case, thus the $O(\lg\sigma)$ penalty.

- (rel_rank_object_major, rel_select_object_major) works in both ways by doing a binary search over the results of the other operation, in the worst case considering $n\sigma$ elements.

- (object_count, object_select) operates the same way as the previous one, but searching among n elements in the worst case, thus the $O(\lg n)$ penalty.

□

The reductions presented here allow us to focus on a small subset of the most difficult operations. In some cases, however, we will present more efficient solutions for the simpler operations and will not use the reduction.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	E	H	D	H	A	C	E	E	G	B	C	B	G	C	F
B	1	10	1	10	10	1	10	1	10	1	10	1	10	10	10

Figure 3.4: Sequence S and bitmap B for representing the binary relation shown in Figure 3.1.

3.2 Reduction to Strings: BinRel-Str

A simple representation [9, 36] of a binary relation $\mathcal{R}$ formed by t pairs in $[1,n]\times[1,\sigma]$ uses a bitmap $B[1,n+t]$ and a string $S[1,t]$ over the alphabet $[1,\sigma]$. The bitmap B concatenates the cardinalities of each column in unary. The string S contains the rows of the pairs in the relation in column-major order. Figure 3.4 shows the representation for the binary relation shown in Figure 3.1. Barbay et al. [9] showed that an easy way to support operations `object_rank1` and `object_select1` on the binary relation is to support the operations `rank` and `select` on B and S, using any data structure known for bitmaps and strings (see Chapter 1). Note also that the operation $\texttt{rel\_count}(1,\sigma,x,y)$ can be answered in $O(1)$ time using B. In the sequel we extend Barbay et al.'s work as much as possible to our considerably larger set of operations. This approach, building only on `rank`, `select` and `access` on B and S, will be called BinRel- Str.

We define some notation to simplify the methods described. First, we define $\texttt{map}(x)$ to be the mapping from a column number x to its last element in S: $\texttt{map}(x) = \texttt{rank}_1(B, \texttt{select}_0(B,x))$. The inverse, from a position in S to its column number, is called $\texttt{unmap}(m) = \texttt{rank}_0(B, \texttt{select}_1(B,m))+1$. Both mappings take constant time. Finally, let us also define for shortness $\texttt{rank}_c(B,x,y) = \texttt{rank}_c(B,y) - \texttt{rank}_c(B,x-1)$.

Assume our representation of S supports `access` in time a, `rank` in time r and `select` in time s. Table 3.1 shows the complexity achieved for each binary relation operation using this approach. As it can be seen, the scheme extends nicely only to operations involving one row or one column. In all the other cases, the complexities are at least linear in the lengths of the ranges to consider[2].

In the next few lemmas, we prove some of the complexities in Table 3.1. The others can be derived using Theorem 3.1.

Lemma 3.1. BinRel-Str *supports* $\texttt{rel\_count}(\alpha,\beta,x,y)$ *in* $O(\min((\beta-\alpha+1)r, (y-x+1)a\lg\beta))$ *time.*

Proof. We can compute $\texttt{rel\_count}(\alpha,\beta,x,y)$ in two ways:

- Using that $\texttt{rel\_count}(\alpha,\beta,x,y) = \sum_{\alpha\le\gamma\le\beta} \texttt{rel\_count}(\gamma,\gamma,x,y)$ and that $\texttt{rel\_count}(\gamma,\gamma,x,y) = \texttt{rank}_\gamma(S, \texttt{map}(x-1)+1, \texttt{map}(y))$, we achieve time $O((\beta-\alpha+1)r)$.
- Using that $\texttt{rel\_count}(\alpha,\beta,x,y) = \sum_{x\le z\le y} \texttt{rel\_count}(\alpha,\beta,z,z)$, we can compute the value for each z by searching for the successor of α and the predecessor of β in $S[\texttt{map}(z-1)+1, \texttt{map}(z)]$.

[2]To simplify, in Table 3.1 we omit some complexities that are most likely to be inferior to their alternatives.

As this range of S is sorted we can find the predecessors and successors using exponential search, which requires in $O(\lg \beta)$ access operations. Thus the overall process takes time $O((y-x+1)a \lg \beta)$.

□

Lemma 3.2. BinRel-Str *supports* $\texttt{label\_count}(\alpha, \beta, x, y)$ *in* $O(\min((\beta-\alpha+1)r, (y-x+1)(\lg \alpha + \beta - \alpha)a))$ *time.*

Proof. We can compute $\texttt{label\_count}(\alpha, \beta, x, y)$ in two ways:

- Using $\texttt{label\_count}(\alpha, \beta, x, y) = \sum_{\alpha \le \gamma \le \beta} \texttt{label\_count}(\gamma, \gamma, x, y)$, and $\texttt{label\_count}(\gamma, \gamma, x, y) = 1$ iff $\texttt{rel\_count}(\gamma, \gamma, x, y) > 0$ and zero otherwise, we can achieve the same time as in the first alternative of Lemma 3.1.
- Using $\texttt{label\_count}(\alpha, \beta, x, y) = |\cup_{x \le z \le y} \texttt{label\_access}(\alpha, \beta, z, z)|$, we can collect the labels in $[\alpha, \beta] \times [z, z]$ for each z and insert them into a dictionary. The labels related to a single z can be found by using method similar to that of Lemma 3.1: use exponential search to find the first element $\ge \alpha$ in z's area of S, and then scan the next symbols until surpassing β. We mark each label found in a bitmap of length $\beta - \alpha + 1$, and then we report the number of ones in it.

□

Lemma 3.3. BinRel-Str *supports* $\texttt{object\_count}(\alpha, \beta, x, y)$ *in* $O(\min((y-x+1)a \lg \alpha, (\beta-\alpha+1)(r + (y-x+1)s)))$ *time.*

Proof. We can compute $\texttt{object\_count}(\alpha, \beta, x, y)$ in two ways:

- Using $\texttt{object\_count}(\alpha, \beta, x, y) = \sum_{x \le z \le y} \texttt{object\_count}(\alpha, \beta, z, z)$, and $\texttt{object\_count}(\alpha, \beta, z, z) = 1$ iff $\texttt{rel\_count}(\alpha, \beta, z, z) > 0$ and zero otherwise, we can proceed in a manner similar to that in the second alternative of Lemma 3.1, by exponentially searching for the first value $\ge \alpha$ in z's area of S and checking whether it is $\le \beta$.
- Using $\texttt{object\_count}(\alpha, \beta, x, y) = |\cup_{\alpha \le \gamma \le \beta} \texttt{object\_access}(\gamma, \gamma, x, y)|$, we can collect the objects in $[\gamma, \gamma] \times [x, y]$ for each γ and insert them into a dictionary, as in Lemma 3.2. The objects related to a single γ can be found by using successive $\texttt{select}_\gamma(S, j)$ operations on $S[\texttt{map}(x-1)+1, \texttt{map}(y)]$, starting with $j = \texttt{rank}_\gamma(S, \texttt{map}(x-1))+1$. The complexity considers the worst case where each such γ appears $y-x+1$ times.

□

Note that in the reduction to implement $\texttt{object\_rank}$, the $\texttt{rank}_\gamma(S, \cdot)$ operation is not necessary. For $\texttt{object\_rank1}$ it is better to reduce from $\texttt{rel\_count}$.

Lemma 3.4. BinRel-Str *supports* `rel_select_label_major`(α, j, x, y) *in* $O(\min((\sigma-\alpha)r+s, (y-x+1)(\sigma-\alpha+1)a))$ *time.*

Proof. Again, we have two possible solutions:

- Set $c \leftarrow 0$. For each label γ in $[\alpha, \sigma]$, compute $c' \leftarrow c + \mathsf{rank}_\gamma(S, \mathsf{map}(x-1)+1, \mathsf{map}(y))$. If at some step it holds $c' \geq j$, the answer is $(\gamma, \mathsf{unmap}(\mathtt{select}_\gamma(S, j-c+\mathsf{rank}_\gamma(S, \mathsf{map}(x-1)))))$. Otherwise, update $c \leftarrow c'$. The overall process takes $O((\sigma-\alpha+1)r+s)$ time.
- This is done in a similar way, but first accumulating all the occurrences of all the labels γ and then finding j using the accumulators. We simply traverse the area $S[\mathsf{map}(z-1)+1, \mathsf{map}(z)]$ backwards, for each $z \in [x, y]$, accessing each label $S[k]$ and incrementing the corresponding counter. The process takes $O((y-x+1)(\sigma-\alpha+1)a)$ time.

□

From this operation we can obtain upper bounds for `rel_min_label_major`, `label_min`, `label_min1`, `label_access`, and `label_access1`. Some of the results we give are better than those obtained by a blind reduction. We also note that an obvious variant of this algorithm is our best solution to compute `label_select`, within the same time.

Lemma 3.5. BinRel-Str *supports* `rel_select_object_major`(α, β, x, j) *in* $O(\min((n-x+1)a\lg\beta, (\beta-\alpha+1)((n-x+1)s+r)))$ *time.*

Proof. Once again, we have two possible solutions:

- Set $c \leftarrow 0$. For each object z in $[x, n]$, use exponential search on z's area of S to find the range $S[a, b]$ corresponding to $[\alpha, \beta]$, and set $c' \leftarrow c + b - a + 1$. If at some step it holds $c' \geq j$, the answer is $(S[j-c+a-1], z)$. Otherwise, update $c \leftarrow c'$. The overall process takes $O((n-x+1)a\lg\beta)$ time.
- This is similar to the first solution, but first accumulating all the occurrences of all the objects z and then finding j using the accumulators. We traverse the area $S[\mathsf{map}(x-1)+1, \mathsf{map}(y)]$ for each label γ in $[\alpha, \beta]$, using successive $\mathtt{select}_\gamma(S, j')$ queries, starting at $j' = \mathsf{rank}_\gamma(S, \mathsf{map}(x-1))+1$. The process takes $O((\beta-\alpha+1)((n-x+1)s+r))$ time.

□

From this operation we can obtain bounds for `rel_min_object_major`, `rel_access`, `object_min`, `object_min1`, `object_access`, and `object_access1`. Once again, some of the results we give are better than a blind reduction Finally, an obvious variant of this algorithm is our best solution to compute `object_select`.

Similarly, `label_select1`(α, j, x) can be solved in time $O(a\lg\alpha)$, and that `object_select1`(α, x, j) is solved in time $O(r+s)$.

Operation	BinRel-Str	BinRel-WT
rel_count(α,β,x,y)	$O((\beta-\alpha+1)r)$ $O((y-x+1)a\lg\beta)$	$O(\lg\sigma)$
rel_rank(α,x)	$O(\alpha r)$ $O(xa\lg\alpha)$	$O(\lg\sigma)$
rel_rank_label_major(α,x,y,z)	$O(\alpha r)$ $O((y-x+1)a\lg\alpha)$	$O(\lg\sigma)$
rel_select_label_major(α,j,x,y)	$O((\sigma-\alpha+1)r+s)$	$O(\lg\sigma)$
rel_min_label_major(α,x,y,z)	$O((\sigma-\alpha+1)r+s)$ $O((y-x+1)a\lg\alpha)$	$O(\lg\sigma)$
rel_rank_object_major(α,β,γ,x)	$O((\beta-\alpha+1)r)$ $O(xa\lg\beta)$	$O(\lg\sigma)$
rel_select_object_major(α,β,x,j)	$O((n-x+1)a\lg\beta)$	$O(\lg j\lg(\beta-\alpha+1)\lg\sigma)$ $O(\lg n\lg\sigma)$
rel_min_object_major(α,β,γ,x)	$O((\beta-\alpha+1)(s+r))$ $O((n-x+1)a\lg\alpha)$	$O(\lg\sigma)$
rel_access(α,β,x,y)	$O((\beta-\alpha+1)r+sk)$ $O((y-x+1)a\lg\alpha+ak)$	$O((k+1)\lg\sigma)$
label_count(α,β,x,y)	$O((\beta-\alpha+1)r)$	$O(\beta-\alpha+\lg\sigma)$
label_rank(α,x,y)	$O(\alpha r)$	$O(\alpha+\lg\sigma)$
label_select(α,j,x,y)	$O((\sigma-\alpha+1)r)$	$O(j\lg\sigma)$
label_access(α,β,x,y)	$O((\beta-\alpha+1)r)$ $O((y-x+1)a\lg\alpha)$	$O((k+1)\lg\sigma)$
label_min(α,x,y)	$O((\sigma-\alpha+1)r)$ $O((y-x+1)a\lg\alpha)$	$O(\lg\sigma)$
object_count(α,β,x,y)	$O((y-x+1)a\lg\alpha)$	$O((y-x+1)\lg\sigma)$
object_rank(α,β,x)	$O(xa\lg\alpha)$	$O(x\lg\sigma)$
object_select(α,β,x,j)	$O((n-x+1)a\lg\alpha)$	$O(j\lg\sigma)$
object_access(α,β,x,y)	$O((\beta-\alpha+1)(r+sk))$ $O((y-x+1)a\lg\alpha)$	$O((k+1)\lg\sigma)$
object_min(α,β,x)	$O((\beta-\alpha+1)(r+s))$ $O((n-x+1)a\lg\alpha)$	$O(\lg\sigma)$
label_rank1(α,x)	$O(a\lg\alpha)$	$O(\lg\sigma)$
label_select1(α,j,x)	$O(a\lg\alpha)$	$O(\lg\sigma)$
label_min1(α,x)	$O(a\lg\alpha)$	$O(\lg\sigma)$
label_access1(α,β,x)	$O(a(k+\lg\alpha))$	$O((k+1)\lg\sigma)$
object_rank1(α,x)	$O(r)$	$O(\lg\sigma)$
object_select1(α,x,j)	$O(r+s)$	$O(\lg\sigma)$
object_min1(α,x)	$O(r+s)$	$O(\lg\sigma)$
object_access1(α,x,y)	$O(r+sk)$	$O((k+1)\lg\sigma)$

Table 3.1: Time complexity for the operations using BinRel-Str and BinRel-WT. The parameter k represents the size of the output for the access operators; one can consider $k = 1$ for the reductions given in Theorem 3.1.

Various string representations [54, 8] offer times a, r, and s that are constant or log-logarithmic in σ. These are the best time complexities we know of for the row-wise and column-wise operations, although this forms a rather limited subset of the operations we have defined.

The space used by most techniques based on representing B and S (including BinRel-WT and BinRel-GWT) is unrelated to $H(\mathcal{R})$, the entropy of the binary relation. Various representations for $S[1,t]$ achieve space $tH_0(S)$ plus some redundancy [59, 54, 8]. This is $tH_0(S) = \sum_{\alpha\in[1,\sigma]} n_\alpha \lg \frac{t}{n_\alpha}$, where n_α is the number of pairs of the form $(\alpha,\cdot)$ in $\mathcal{R}$. While this can be lower than $H(\mathcal{R})$ (which shows that our measure $H(\mathcal{R})$, which is a 0-th order entropy measure, is rather crude), it can also be arbitrarily higher. For example an almost full binary relation has an entropy $H(\mathcal{R})$ close to zero, but its $tH_0(S)$ is close to $n\sigma\lg\sigma$. A clearer picture is obtained if we assume that S is represented in plain form using $t\lg\sigma$ bits. This is to be compared to $H(\mathcal{R}) = t\lg\frac{n\sigma}{t} + O(t)$, which shows that the string representation is competitive for sparse relations, $t = O(n)$.

3.3 Using Wavelet Trees: BinRel-WT

Among the many representations of string S we can choose for the BinRel-Str approach, wavelet trees [59] are particularly interesting. Although the time wavelet trees offer for a, r and s is $O(\lg\sigma)$, not the best ones for large σ, wavelet trees allow us to support many more operations efficiently, via other algorithms than those used by the three basic operations. We call this representation BinRel-WT. Table 3.1 summarizes the time complexity for each operation using BinRel-WT, in comparison to a general BinRel-Str. Next, we show how to support some key operations efficiently; the other complexities are inferred from Theorem 3.1.

The first lemma states a well-known algorithm on wavelet trees [80].

Lemma 3.6. BinRel-WT *supports* `rel_rank`(α,x) *in* $O(\lg\sigma)$ *time.*

Proof. This operation is $\texttt{rank}_{\leq\alpha}(S,\texttt{map}(x))$, where operation $\texttt{rank}_{\leq\alpha}(S,p)$ counts the number of symbols $\leq\alpha$ in $S[1,p]$. It can be supported in time $O(\lg\sigma)$ on the wavelet tree of S by following the root-to-leaf branch corresponding to α, while counting at each node the number of objects preceding position p that are related with a label preceding α, as follows. Start at the root v with counter $c \leftarrow 0$. If α corresponds to the left subtree, then enter the left subtree with $p \leftarrow rank_0(B_v,p)$. Otherwise enter the right subtree with $c \leftarrow c + rank_0(B_v,p)$ and $p \leftarrow \texttt{rank}_1(B_v,p)$. The process continues recursively until a leaf is reached (indeed, that of α), so the answer is $c+p$. □

The next lemma solves an extended variant of a query that has been called *range_quantile*. It was also solved with wavelet trees within the same time bound [50]. Note that the lemma gives also a solution within the same time complexity for `label_min`, which in the literature [50] was called *range_next_value* and solved with an ad-hoc algorithm, in the same time bound.

Lemma 3.7. BinRel-WT *supports* $\texttt{rel\_select\_label\_major}(\alpha, j, x, y)$ *in* $O(\lg\sigma)$ *time.*

Proof. We first get rid of α by setting $j \leftarrow j + \texttt{rel\_count}(1, \alpha-1, x, y)$ and thus reduce to the case $\alpha = 1$. Furthermore we map x and y to the domain of S by $p \leftarrow \texttt{map}(x-1)+1$ and $q \leftarrow \texttt{map}(y)$. We first find the symbol β whose row contains the j-th element. For this we first find β such that $\texttt{rank}_{\leq\beta-1}(S,p,q) < j \leq \texttt{rank}_{\leq\beta}(S,p,q)$. This is achieved in time $O(\lg\sigma)$ as follows. Start at the root v and set $j' \leftarrow j$. If $\texttt{rank}_0(B_v,p,q) \geq j$, then continue to the left subtree with $p \leftarrow \texttt{rank}_0(B_v,p-1)+1$ and $q \leftarrow \texttt{rank}_0(B_v,q)$. Otherwise continue to the right subtree with $j' \leftarrow j' - \texttt{rank}_0(B_v,p,q)$, $p \leftarrow \texttt{rank}_1(B_v,p-1)+1$, and $y \leftarrow \texttt{rank}_1(B_v,q)$. The leaf arrived at is β. Finally, we answer $(\beta, \texttt{unmap}(\texttt{select}_\beta(S, j' + \texttt{rank}_\beta(S,p-1))))$. □

The wavelet tree is asymmetric with respect to objects and labels. The transposed problem, $\texttt{rel\_select\_object\_major}$, turns out to be harder. We present, however, a polylogarithmic-time solution.

Lemma 3.8. BinRel-WT *supports* $\texttt{rel\_select\_object\_major}(\alpha, \beta, x, j)$ *in* $O(\min(\lg n, \lg j \lg(\beta-\alpha+1))\lg\sigma)$ *time.*

Proof. Recall that the elements are written in S in object major order. First, we note that the particular case where $[\alpha,\beta] = [1,\sigma]$ is easily solved in $O(\lg\sigma)$ time, by doing $j' \leftarrow j + \texttt{rel\_count}(1,\sigma,1,x-1)$ and returning $(S[j'], \texttt{unmap}(j'))$. In the general case, one can obtain time $O(\lg n \lg\sigma)$ by binary searching the column y such that $\texttt{rel\_count}(\alpha,\beta,x,y-1) < j \leq \texttt{rel\_count}(\alpha,\beta,x,y)$. Then the answer is $(\texttt{label\_select1}(\alpha, j - \texttt{rel\_count}(\alpha,\beta,x,y-1), y), y)$ (note that Lemma 3.7 already gives us $\texttt{label\_select1}$ in time $O(\lg\sigma)$).

To obtain the other bound in the min function, we find the $O(\lg(\beta-\alpha+1))$ wavelet tree nodes that cover the interval $[\alpha,\beta]$; let these be $v_1, v_2, \ldots, v_k$. We map position $p = \texttt{map}(x-1)+1$ from the root towards those v_is, obtaining all the mapped positions p_i in $O(k + \lg\sigma)$ time [50]. Now the answer is within the positions $[p_i, p_i + j - 1]$ of some i. We cyclically take each v_i, choose the middle element of its interval, and map it towards the root, obtaining position q, corresponding to pair $(S[q], \texttt{unmap}(q))$. If $\texttt{rel\_rank\_object\_major}(\alpha,\beta,S[q],\texttt{unmap}(q)) - \texttt{rel\_count}(\alpha,\beta,1,x-1) = j$, the answer is $(S[q], \texttt{unmap}(q))$. Otherwise we know whether q is before or after the answer. So we discard the left or right interval in v_i. After $O(k \lg j)$ such iterations we have reduced all the intervals of length j of all the nodes v_i, finding the answer. Each iteration costs $O(\lg\sigma)$ time. □

The next lemma solves a more general variant of a problem that has been called *prevLess*, and also solved using wavelet trees [77]. We achieve the same complexity for this more general variant. Note theirs is a simplification of $\texttt{rel\_select\_object\_major}$ that we can solve within time $O(\lg\sigma)$, whereas for general j we cannot.

Lemma 3.9. BinRel-WT *supports* $\texttt{rel\_min\_object\_major}(\alpha, \beta, \gamma, x)$ *in* $O(\lg\sigma)$ *time per pair output.*

Proof. We first use $\texttt{label\_min1}(\gamma, x)$ (which we already can solve in time $O(\lg\sigma)$ as a consequence of Lemma 3.7) to search for a point in the band $[\gamma,\beta]\times[x,x]$. If we find one, then this is the answer, otherwise we continue with the area $[\alpha,\beta]\times[x+1,n]$.

Just as for the second solution of Lemma 3.8, we obtain the positions p_i at the nodes v_i that cover $[\alpha,\beta]$. The first element to deliver is precisely one of those p_i. We have to merge the results, always choosing the smaller, as we return from the recursion that identifies the v_i nodes. If we are in v_i, we return $q = p_i$. Otherwise, if the left child of v returned q, we map it to $q' \leftarrow \texttt{select}_0(B_v, q)$. Similarly, if the right child of v returned q, we map it to $q'' \leftarrow \texttt{select}_1(B_v, q)$. If we have only q' (q''), we return $q = q'$ ($q = q''$); if we have both we return $q = \min(q', q'')$. The process takes $O(\lg\sigma)$ time. When we arrive at the root we have the next position q where a label in $[\alpha,\beta]$ occurs in S, and thus return $\texttt{unmap}(q)$. □

The next lemma can also be obtained by considering the complexity of the BinRel-Str scheme implemented over a wavelet tree.

Lemma 3.10. BinRel-WT *supports* $\texttt{object\_select1}(\alpha, x, j)$ *in* $O(\lg\sigma)$ *time.*

Proof. This is a matter of selecting the j-th occurence of the label α in S, after the position of the pair (α, x). The formula is $\texttt{unmap}(\texttt{select}_\alpha(S, j + \texttt{object\_rank1}(\alpha, x-1)))$. □

The next operation is the first of the set we cannot solve within polylogarithmic time.

Lemma 3.11. BinRel-WT *supports* $\texttt{label\_count}(\alpha,\beta,x,y)$ *in* $O(\beta - \alpha + \lg\sigma)$ *time.*

Proof. After mapping $[x,y]$ to positions $S[p,q]$, we descend in the wavelet tree to find all the leaves in $[\alpha,\beta]$ while remapping $[p,q]$ appropriately. We count one more label each time we arrive at a leaf, and we stop descending from an internal node if its range $[p,q]$ is empty. The complexity comes from the number of wavelet tree nodes accessed to reach such leaves [50]. □

The remaining operations are solved naïvely: `label_select` and `object_select` use, respectively, `label_min` and `object_min` successively, and `object_count` and `object_rank` use `object_rank1` successively,

The overall result is stated in the next theorem and illustrated in Figure 3.5.

Theorem 3.2. *The structure* BinRel-WT, *for a binary relation $\mathcal{R}$ of t pairs over $[1,\sigma]\times[1,n]$, uses $t\lg\sigma + O(n+t)$ bits of space and supports the operations within the time complexities given in Table 3.1.*

Proof. The space assumes a plain uncompressed wavelet tree and bitmap representations, and the time complexities have been discussed throughout the section. □

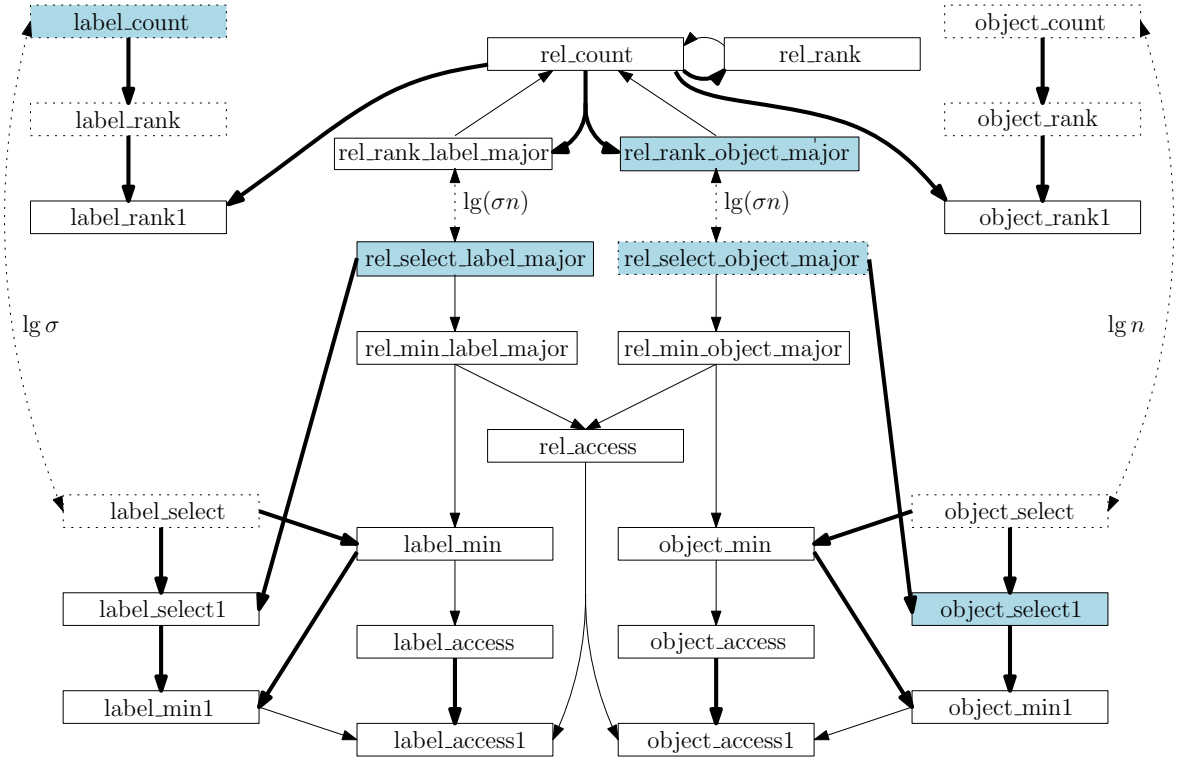

Figure 3.5: Reductions among operations supported by BinRel-WT. The dotted boxes are operations supported in $\omega(\lg \sigma)$ time, the filled boxes represent operations we address directly, and the blank boxes are supported via reductions.

3.4 Using a Generalized Wavelet Tree: BinRel-GWT

The results we obtained for the wavelet tree can be extended to the generalized wavelet tree, improving complexities in many cases (recall Chapter 2). We refer to the structure that represents S using the generalized wavelet tree as BinRel-GWT, and we use μ to represent the fan-out of the tree. We require $\mu \in O(\lg^{\epsilon}(n\sigma))$, assuming that the RAM machine can address up to $n \times \sigma$ cells. To simplify we will assume $\sigma \leq n$ and then simply use $\mu \in O(\lg^{\epsilon} n)$.

A first simple result stems from the fact that the string operations are sped up on generalized wavelet trees.

Lemma 3.12. BinRel-GWT *supports* `object_rank1` *and* `object_select1` *in time* $O(\lg_{\mu} \sigma)$ *for any* $\mu \in \Theta(\lg^{\epsilon} n)$ *and any constant* $0 < \epsilon < 1$.

Proof. This follows directly from the results on general BinRel-Str structures. □

The following notation will be useful to describe our algorithms within wavelet tree nodes. Note that all are easily computed in constant time.

- $\texttt{child}(k)$: Given a symbol $k \in [1,\mu]$, this is the subtree labeled k of the current node.
- $\mathrm{g}(\alpha)$: Given a symbol $\alpha \in [1,\sigma]$, $\mathrm{g}(\alpha)$ is the symbol k such that $\texttt{child}(k)$ contains the leaf corresponding to α.
- $\mathrm{g}^{-1}(k)$: Given a symbol $k \in [1,\mu]$, $\mathrm{g}^{-1}(k) = \min\{\alpha \mid k = \mathrm{g}(\alpha)\}$.

The next lemma shows how to speed up all the range counting operations. The result is known in the literature for $n \times n$ grids, even within $n \lg n(1+o(1))$ bits [18].

Lemma 3.13. BinRel-GWT *supports* $\texttt{rel\_rank}(\alpha,x)$ *in* $O(\lg_\mu \sigma)$ *time, for any* $\mu \in \Theta(\lg^\epsilon n)$ *and any constant* $0 < \epsilon < 1$.

Proof. As in the case of BinRel-WT, we reduce this problem to the one of computing $\texttt{rank}_{\le\alpha}(S, \texttt{map}(x))$, which can be done by following a similar procedure: We follow the path for α starting at the root in position $p = \texttt{map}(x)$ and with a counter $c \leftarrow 0$. Every time we move to a subtree, we increase $c \leftarrow c + \texttt{rank}_{\le g(\alpha)-1}(S_v, p)$. When we arrive at the leaf, the answer is $c + p$.

Operation $\texttt{rank}_{\le k}(S_v, p)$ can be solved in constant time for $\mu \in \Theta(\lg^\epsilon n)$ analogously as done for $\texttt{rank}_k$ on small alphabets (recall Chapter 1). We store for each $k \in [1,\mu]$ a bitmap $B_{\le k}$ such that $B_{\le k}[i] = 1$ iff $S_v[i] \le k$. Thus $\texttt{rank}_{\le k}(S_v, p) = \texttt{rank}_1(B_{\le k}, p)$ is computed in constant time and the whole process takes $O(\lg_\mu \sigma)$ time. □

The next lemma covers operation $\texttt{rel\_select\_label\_major}$, on which we cannot improve the complexity given by BinRel-WT. Note this means that $O(\lg \sigma)$ is still the best time complexity for supporting *range_quantile* queries within linear space [50].

Lemma 3.14. BinRel-GWT *supports* $\texttt{rel\_select\_label\_major}(\alpha, j, x, y)$ *in* $O(\lg \sigma)$ *time.*

Proof. This is solved in a similar way to the one presented for BinRel-WT. We find v such that $\texttt{rank}_{\le\beta-1}(S,p,q) < v \le \texttt{rank}_{\le\beta}(S,p,q)$. The only difference is that in this case we have to do, at each node, a binary search for the right child $k \in [1,\mu]$ to descend, and thus the time is $O(\lg \mu \lg_\mu \sigma) = O(\lg \sigma)$. □

For the next operations we augment the generalized wavelet tree with a set of bitmaps inside each node v. More specifically, we add $\mu(\mu+1)/2$ bitmaps $B_{k,l}$, where $B_{k,l}[i] = 1$ iff $S_v[i] \in [k,l]$. Just as with bitmaps $B_{\le k}$, bitmaps $B_{k,l}$ are not represented explicitly, but only their index is stored, and their content is simulated in constant time using S_v. Their total space for a sequence $S_v[1,n]$ is $O(n\mu^2 \lg\lg n / \lg_\mu n)$. To make this space negligible, that is, $o(n \lg \mu)$, it is sufficient that $\mu = O(\lg^\epsilon n)$ for any constant $0 < \epsilon < 1/2$. (A related idea has been used by Farzan et al. [43].)

The next lemma shows that the current solution for operation *prevLess* [77] can be sped up by an $O(\lg\lg n)$ factor.

Lemma 3.15. BinRel-GWT *supports* `rel_min_object_major`(α,β,γ,x) *in* $O(\lg_\mu \sigma)$ *time, for any* $\mu \in \Theta(\lg^\epsilon n)$ *and any constant* $0 < \epsilon < 1/2$.

Proof. We first run query `rel_min_object_major`(γ,β,γ,x), and if the result is on column x, we report it. Otherwise we run query `rel_min_object_major`$(\alpha,\beta,\alpha,x+1)$. This means that we can focus on a simpler query of the form `rel_min_object_major`(α,β,x), which finds the first pair in $[\alpha,\beta]\times[x,n]$, in object-major order. We map $[x,n]$ to $S[p,t]$ as usual and then proceed recursively on the wavelet tree, remapping p. At each node v, we decompose the query into three subqueries, and then take the minimum result of the three:

1. `rel_min_object_major`$(\alpha, \mathrm{g}^{-1}(\mathrm{g}(\alpha)+1)-1, x)$ on node `child`(α);
2. `rel_min_object_major`$(\mathrm{g}^{-1}(\mathrm{g}(\alpha)+1), \mathrm{g}^{-1}(\mathrm{g}(\beta))-1, x)$ on the same node v;
3. `rel_min_object_major`$(\mathrm{g}^{-1}(\mathrm{g}(\beta)), \beta, x)$ on node `child`(β).

Note that queries of type 1 will generate, recursively, only $O(\lg_\mu \sigma)$ further queries of type 1 and 2, and similarly queries of type 3 will generate $O(\lg_\mu \sigma)$ further queries of type 3 and 2. The only queries that actually deliver values are those of type 2, and we will have to take the minimum over $O(\lg_\mu \sigma)$ such results.

A query of type 2 is solved in constant time using bitmap $B_{\mathrm{g}(\alpha)+1,\mathrm{g}(\beta)-1}$, by computing $q = \mathtt{select}_1(B_{\mathrm{g}(\alpha)+1,\mathrm{g}(\beta)-1}, \mathtt{rank}_1(B_{\mathrm{g}(\alpha)+1,\mathrm{g}(\beta)-1}, p-1)+1)$. This returns a position $S_v[q]$. As we return from the recursion, we remap q in its parent in the usual way, and then (possibly) compare q with the result of a query of type 1 or 3 carried out on the parent. We keep the minimum q value along the way, and when we arrive at the root we return $(S[q], \mathtt{unmap}(q))$. □

For the next lemma we need an additional data structure. For each sequence $S_v[1,n]$, we store an RMQ structure, using $O(n) = o(n \lg \mu)$ bits and finding in constant time position of a minimum symbol in any range $S_v[i,j]$ [49]. This result improves upon the result for query *range_next_value* [50].

Lemma 3.16. BinRel-GWT *supports* `rel_min_label_major`(α,x,y,z) *in* $O(\lg_\mu \sigma + \lg \mu)$ *time, for any* $\mu \in \Theta(\lg^\epsilon n)$ *and any constant* $0 < \epsilon < 1$.

Proof. Again, we can focus on a simpler query `rel_min_label_major`(α,x,y). We map $[x,y]$ to $S[p,q]$ as usual, and the goal is to find the leftmost minimum symbol in $S[p,q]$ that is larger than α.

Assume we are in a wavelet tree node v and the current interval of interest is $S_v[p,q]$. Then, if $S_v[p,q]$ contains symbol $\mathrm{g}(\alpha)$ (which is known in constant time with $\mathtt{rank}_{\mathrm{g}(\alpha)}(S_v,p,q) > 0$), we have to consider it first, by querying recursively the child labeled $\mathrm{g}(\alpha)$. If this recursive call returns an answer p', we return it in turn, remapping it to the parent node. If it does not, then any symbol larger than α in the range must correspond to a symbol strictly larger than $\mathrm{g}(\alpha)$ in

$S_v[p,q]$. We check in constant time whether there is any value larger than $\mathrm{g}(\alpha)$ in $S_v[p,q]$, using $\mathtt{rank}_{\leq \mathrm{g}(\alpha)}(S_v,p,q) < q-p+1$. If there is none, we return with no answer.

If there is an answer, we binary search for the smallest $k \in [\mathrm{g}(\alpha)+1, \mu]$ such that $\mathtt{rank}_{\leq k}(S_v,p,q) > \mathtt{rank}_{\leq \mathrm{g}(\alpha)}(S_v,p,q)$. This binary search takes $O(\lg \mu)$ time and is done only once along the whole process. Once we identify the right k, we descend to the appropriate child and start the final stage of the process.

The final stage starts at a node where all the local symbols represent original symbols that are larger than α, and therefore we simply look for the position $m = \textsc{rmq}(S_v,p,q)$, which gives us, in constant time, the first occurrence of the minimum symbol in S_v, and descend to child $S[m]$. This is done until reaching a leaf, from where we return to the root, at position p', and return $(S[p'], \mathtt{unmap}(p'))$. It is easy to see that we work $O(1)$ time on $O(\lg_\mu \sigma)$ nodes and $O(\lg \mu)$ once. □

Lemma 3.17. BinRel-GWT *supports* `rel_select_object_major`(α,β,x,j) *in* $O(\min(\lg n, \lg j \lg(\beta-\alpha+1))\lg_\mu \sigma)$ *time, for any* $\mu \in \Theta(\lg^\epsilon n)$ *and any constant* $0 < \epsilon < 1/2$.

Proof. The complexities are obtained the same way as for BinRel-WT. The binary search over `rel_count` is sped up because BinRel-GWT supports this operation faster. The other complexity is in principle higher, because the interval $[\alpha,\beta]$ is split into as many as $O(\mu \lg(\beta-\alpha+1))$ nodes. However, this can be brought down again to $O(\lg(\beta-\alpha+1))$ by using the parent node v of each group of (up to μ) contiguous leaves $[k,l]$, and using $\mathtt{select}_1$ on the bitmaps $B_{k,l}$ of those parent nodes in order to simulate a contiguous range with all the values in $[k,l]$. So we still have $O(\lg(\beta-\alpha+1))$ binary searches of $O(\lg j)$ steps, and now each step costs $O(\lg_\mu \sigma)$. □

Lemma 3.18. BinRel-GWT *supports* `label_count`(α,β,x,y) *in* $O(\beta-\alpha+\lg_\mu \sigma)$ *time, for any* $\mu \in \Theta(\lg^\epsilon n)$ *and any constant* $0 < \epsilon < 1/2$.

Proof. We follow the same procedure as for BinRel-WT. The main difference is how to compute the nodes covering the range $[\alpha,\beta]$. This can be done in a naïve way by just verifying whether each symbol appears in the range of S_v, but this raises the complexity by a factor of μ. Thus we need a method to list the symbols appearing in a range of S_v without probing non-existent ones. We resort to a technique loosely inspired by Muthukrishnan [88]. To list the symbol from a range $[k,l]$ that exist in $S_v[p,q]$, we start with the first symbol of the range that appears in $S_v[p,q]$. This is obtained with $p' = \mathtt{select}_1(B_{k,l}, \mathtt{rank}_1(B_{k,l}, p-1)+1)$. If $p' > q$ then there are no such symbols. Otherwise, let $k' = S_v[p']$. Then we know that k' appears in $S_v[p,q]$. Now we continue recursively with subranges $[k,k'-1]$ and $[k'+1,l]$. The recursion stops when no p' is found, and it yields all the symbols appearing in $S_v[p,q]$ in $O(1)$ time per symbol. □

The remaining operations are obtained by brute force, just as with BinRel-WT. Figure 3.6 illustrates the reductions used.

Theorem 3.3. *The structure* BinRel-GWT, *for a binary relation* $\mathcal{R}$ *of* t *pairs over* $[1,\sigma]\times[1,n]$, *requires* $t\lg\sigma(1+o(1))+O(n+t)$ *bits of space and supports the operations within the time complexities given in Table 3.2.*

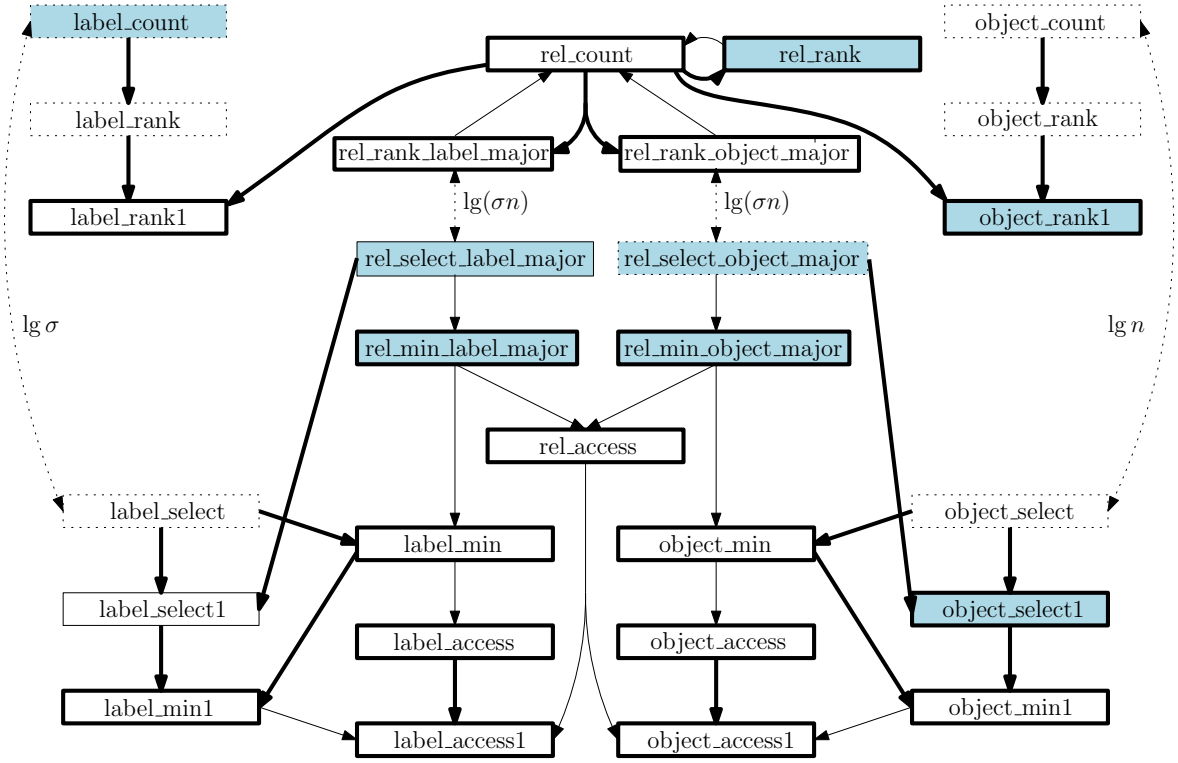

Figure 3.6: Reductions among operations supported by BinRel-GWT. The dotted boxes are operations supported in $\omega(\lg\sigma)$ time, the solid ones in $O(\lg\sigma)$ time, and the thick ones in time $O(\lg_\mu\sigma)$ or $O(\lg_\mu\sigma+\lg\mu)$. The filled boxes represent operations addressed directly, and the blank boxes are supported via reductions.

3.5 Binary Relation Wavelet Trees (BRWT)

We now propose a special wavelet tree structure tailored to the representation of binary relations. This wavelet tree contains two bitmaps, B^l and B^r, at each node, so for each node v we have B^l_v and B^r_v. At the root, $B^l_v[1,n]$ has the bit x set to 1 iff there exists a pair (α,x) with $\alpha\in[1,\lfloor\sigma/2\rfloor]$, and B^r_v has the bit x set to 1 iff there exists a pair (α,x) with $\alpha\in[\lfloor\sigma/2\rfloor+1,\sigma]$. Left and right subtrees are recursively built on the positions set to 1 in B^l_v and B^r_v, respectively. The leaves (where no bitmap is stored) correspond to individual rows of the relation. We store a bitmap $B[1,\sigma+t]$ recording in unary the number of elements in each row. See Figure 3.7 for an example. For ease of notation, we define the following functions on B, easily supported in constant-time: $\mathtt{lab}(r)=1+\mathtt{rank}_0(B,\mathtt{select}_1(B,r))$ gives the label of the r-th pair in a label-major traversal of R; while its inverse $\mathtt{poslab}(\alpha)=\mathtt{rank}_1(B,\mathtt{select}_0(B,\alpha))$ gives the position in the traversal where the pairs for label α start.

Note that, because an object x may propagate both left and right, the sizes of the second-level bitmaps may total more than n bits. Indeed, the last level contains t bits and represents all the

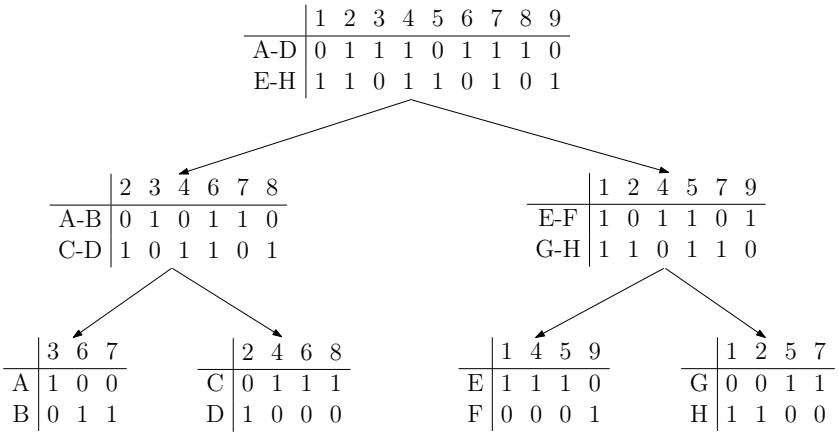

Figure 3.7: Example of the BRWT for the binary relation in Figure 3.1.

pairs sorted in row-major order. As we will see, the BRWT has weaker functionality than our former structures based on wavelet trees, but it reaches space proportional to $H(\mathcal{R})$.

Lemmata 3.19 to 3.23 give a set of operations that can be supported with the BRWT structure.

Lemma 3.19. BRWT *supports* `rel_count`(α, β, x, y) *in* $O(\beta - \alpha + \lg \sigma)$ *time.*

Proof. We project the interval $[x, y]$ from the root to each leaf in $[\alpha, \beta]$, computing the sum of the resulting interval sizes at the leaves. Of course we can stop earlier if the interval becomes empty. Note that we can only count pairs at the leaves, not at internal nodes. □

Lemma 3.20. BRWT *supports* `rel_min_label_major`(α, x, y, z) *in* $O(\lg \sigma)$ *time.*

Proof. As before (Lemma 3.16), we need only consider the simpler query `rel_min_label_major`(α, x, y). We reach the $O(\lg \sigma)$ wavelet tree nodes $v_1, v_2, \ldots$ that cover the interval $[\alpha, \sigma]$, and map $[x, y]$ to all those nodes, in $O(\lg \sigma)$ time [50]. We choose the first such node, v_k, left to right, with a nonempty interval $[x, y]$. Now we find the leftmost leaf of v_k that has a nonempty interval $[x, y]$, which is easily done in $O(\lg \sigma)$ time. Once we arrive at such a leaf γ with interval $[x, y]$, we map x back to the root, obtaining x', and the answer is (γ, x'). □

Lemma 3.21. BRWT *supports* `rel_min_object_major`$(\alpha, \beta, \gamma, x)$ *in* $O(\lg \sigma)$ *time.*

Proof. As before, we need only consider the simpler query `rel_min_object_major`(α, β, x). Analogously to the proof of Lemma 3.9, we cover $[\alpha, \beta]$ with $O(\lg \sigma)$ wavelet tree nodes $v_1, v_2, \ldots$, and map x to x_i at each such v_i, all in $O(\lg \sigma)$ time. Now, on the way back of this recursion, we obtain the smallest $y \geq x$ in the root associated to some label in $[\alpha, \beta]$. In this process we keep track of the node v_i that is the source of y, preferring the left child in case of ties. Finally, if we arrive at the root with a value y that came from node v_i, we start from position $x' = x_i$ at node v_i and find

the leftmost leaf of v_i related to y. This is done by going left whenever possible (i.e., if $B_v^l[x'] = 1$) and right otherwise, and remapping x' appropriately at each step. Upon reaching a leaf γ, we report (γ, y). □

Lemma 3.22. BRWT *supports* `object_select1`(α, x, j) *in* $O(\lg \sigma)$ *time.*

Proof. We map $x-1$ from the root to x' in leaf α, then walk upwards the path from $x'+j$ to the root and report the position obtained. □

Lemma 3.23. BRWT *supports* `label_count`(α, β, x, y) *in* $O(\beta - \alpha + \lg \sigma)$ *time.*

Proof. We map $[x, y]$ from the root to each leaf in $[\alpha, \beta]$, adding one per leaf where the interval is non-empty. Recursion can also stop when $[x, y]$ becomes empty. □

The remaining constructions are obtained by brute force: `rel_select_label_major` and `rel_select_object_major` are obtained by iterating with `rel_min_label_major` and `rel_min_object_major`, respectively; and similarly `label_select`, `object_select` and `label_select1` using `label_min`, `object_min`, and `label_min1`. Finally, as before `object_count` and `object_rank` are obtained by iterating over `object_rank1`. We have obtained the following theorem, illustrated in Figure 3.8.

Theorem 3.4. *The* BRWT *structure, for a binary relation $\mathcal{R}$ of t pairs over $[1, \sigma] \times [1, n]$, uses* $\lg(1+\sqrt{2})H(\mathcal{R}) + O(t+n+\sigma)$ *bits of space and supports the operations within the time bounds given in Table 3.3.*

Proof. The operations have been discussed throughout the section. For the space, B is of length $\sigma + t$. Thus $O(t+n+\sigma)$ bits account for B and for the $2n$ bits at the root of the wavelet tree. The rest of the bits in the wavelet tree can be counted by considering that each bit not in the root is induced by the presence of a pair.

Each pair has a unique representative bit in a leaf, and also induces the presence of bits up to the root. Yet those leaf-to-root paths get merged, so that not all those bits are different. Consider an element x related to t_x labels. It induces t_x bits at t_x leaves, and each such bit at a leaf induces a bit per level on a path from the leaf towards the single x at the root.[3] At worst, all the $O(t_x)$ bits up to level $\lg t_x$ are created for these elements, and from there on all the t_x paths are different, adding up a total of $O(t_x) + t_x \lg \frac{\sigma}{t_x}$ bits. Adding over all x we get $O(t) + \sum_x t_x \lg \frac{\sigma}{t_x}$. This is maximized when $t_x = t/n$ for all x, yielding $O(t) + t \lg \frac{\sigma n}{t} = H(\mathcal{R}) + O(t)$ bits.

Instead of representing two bitmaps (which would multiply the above value by 2), we can represent a single sequence B_v with the possible values of the two bits at each position, 00,

[3] For example, in Figure 3.7, object $x = 4$ is related to labels C and D (see also Figure 3.1). Its 1 at the second leaf, for C, induces a 1 at its parent, for C-D, and a 1 at the root, for A-D. Its 1 at the third leaf, for E, induces a 1 at its parent for E-F and the 1 at the root for E-H. The fact that $(4, \mathrm{C}) \in \mathcal{R}$ induces the creation of one column at the leaf for C and one at its parent. On the other hand, there are two pairs related to object 1, but they are merged at the second level and thus there is only one path arriving at the root.

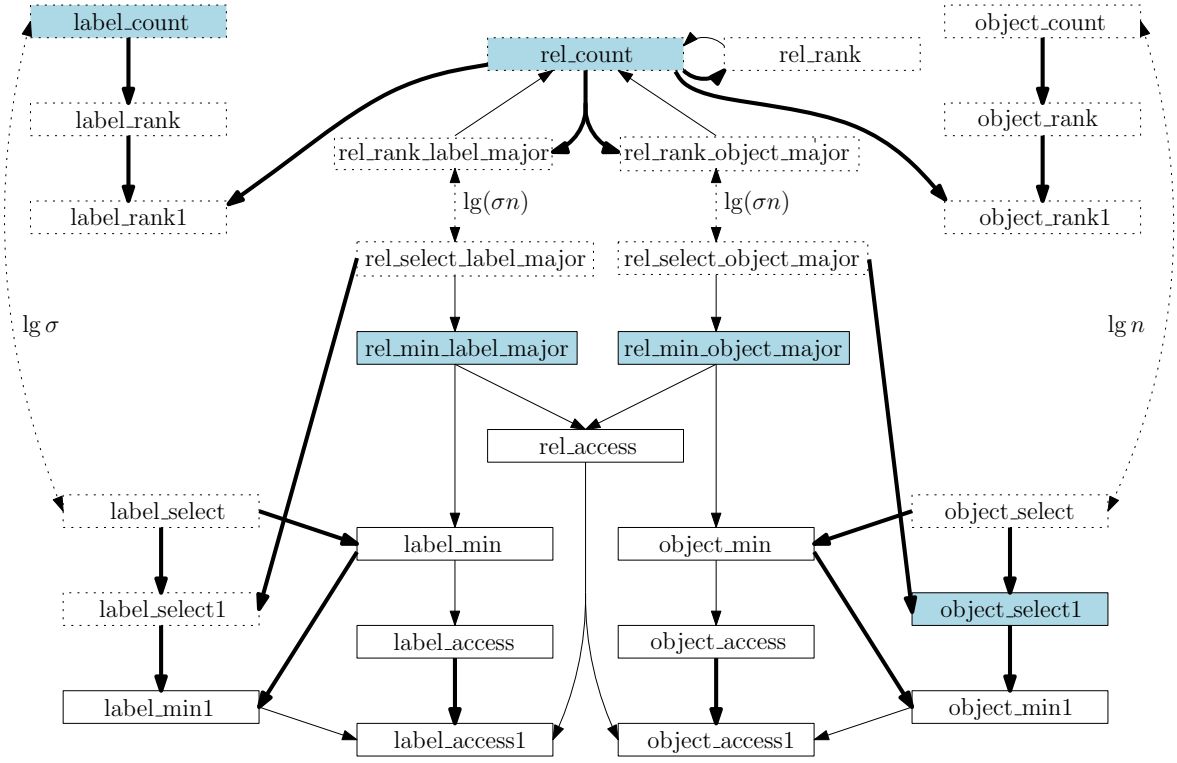

Figure 3.8: Reductions among operations supported by BRWT. The dotted boxes are operations supported in $\omega(\lg\sigma)$ time, the filled boxes represent operations directly addressed in this chapter, and the blank boxes are supported via reductions.

01, 10, 11. Only at the root is 00 possible. Except for those $2n$ bits, we can represent the sequence over an alphabet of size 3 with a zero-order representation [55], to achieve at worst $(\lg 3)H(\mathcal{R}) + o(t)$ bits for this part while retaining constant-time `rank` and `select` over each B_v^l and B_v^r. (To achieve this, we maintain the directories for the original bitmaps, of sublinear size.)

To improve the constant $\lg 3$ to $\lg(1+\sqrt{2})$, we consider that the zero-order representation actually achieves $|B_v|H_0(B_v)$ bits. We call B_x the concatenation of all the symbols induced by x, $\ell_x = |B_x| \le t_x$, and $H_x = |B_x|H_0(B_x)$. Assume the t_x bits are partitioned into t_{01} 01's, t_{10} 10's, and t_{11} 11's, so that $t_x = t_{01} + t_{10} + 2t_{11}$, $\ell_x = t_{01} + t_{10} + t_{11}$, and $H_x = t_{01}\lg\frac{\ell_x}{t_{01}} + t_{10}\lg\frac{\ell_x}{t_{10}} + t_{11}\lg\frac{\ell_x}{t_{11}}$. As $t_{11} = (t_x - t_{01} - t_{10})/2$, the maximum of H_x as a function of t_{01} and t_{10} yields the worst case at $t_{01} = t_{10} = \frac{\sqrt{2}}{4}t_x$, so $t_{11} = (\frac{1}{2} - \frac{\sqrt{2}}{4})t_x$ and $\ell_x = (\frac{1}{2} + \frac{\sqrt{2}}{4})t_x$, where $H_x = \lg(1+\sqrt{2})t_x$ bits. This can be achieved separately for each symbol. Using the same distribution of 01's, 10's, and 11's for all x we add up to $\lg(1+\sqrt{2})t\lg\frac{\sigma n}{t} + O(t) = \lg(1+\sqrt{2})H(\mathcal{R}) + O(t)$ bits. (Note that, if we concatenate all the wavelet tree levels, the H_x strings are interleaved in this concatenation.) □

Note that when $H(\mathcal{R}) = \Omega(t)$, the higher order term of the space used by the structure is $\lg(1+\sqrt{2}) \approx 1.272$ times the entropy of R.

3.6 Adaptive Representations

In this section, we present space-efficient data structures that have adaptive space and time complexities. Our approach comes from a geometrical perspective, and for permutations, converges to the representation by Barbay and Navarro [12]. However, our approach brings a new perspective, showing how to support range searching operations, extending the results of [12] to binary relations with Theorem 3.5, and show an alternative tradeoff deriving directly from our formulation.

3.6.1 Monotonic Decomposition of Sequences.

Arroyuelo et al. [3] presented an adaptive data structure for range searching that decomposes the set of points into non-crossing ascending and descending chains. Let χ be the number of chains generated by such a decomposition, the search time for a range query is $O(\chi + \lg n + k)$, where k is the number of points in the answer. As an alternative, one can support the same query in $O(\lg \chi \lg n + \chi' + m)$ time, where χ' corresponds to the number of chains intersecting the query rectangle and to the size of the output, m. This is in principle the same decomposition we will use in this work. However, since we do not make use of fractional cascading, our complexities are a bit worse.

We only require the chains[4] to be untangled if we want to support range queries, otherwise any monotonic decomposition would do. The decomposition into non-crossing chains can be computed in polynomial time if we are given an optimal decomposition into monotonic subsequences [3]. The optimal decomposition into monotonic subsequences is NP-Hard [113], yet it is interesting that the optimal decomposition for a permutation of length n is bounded by $c\sqrt{n}$, where $c \leq 2$, and that we can get a constant factor approximation in polynomial time [117]. In this work we consider the optimal decomposition and show how this allows for a representation that is adaptive in the number of monotonic subsequences a permutation or binary relation can be decomposed. The results as stated apply also for the case when we compute a constant factor approximation, thus making the data structure feasible in practice.

3.6.2 An Adaptive Representation for Permutations

Our representation works by decomposing the permutation into ascending and descending subsequences. A simple way to visualize this is to consider the representation of the permutation in a grid, as shown in Figure 3.9. Every row represents the index i, the columns represent the value of $\Pi[i]$. The permutation shown in the example is $[3,6,5,1,4,7,2]$, and it is easy to see

[4] We also use the term subsequence to refer a chain.

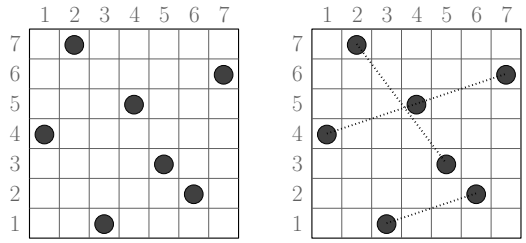

Figure 3.9: Example of permutation seen as a grid (left) and a possible decomposition into ascending/descending chains (right).

that the inverse permutation corresponds just to the transposed matrix. In order to simplify the presentation of this work, we will only consider ascending subsequences, the results extend easily to the general case.

In Figure 3.9 (right) we can see a possible decomposition of a permutation into subsequences or chains. We show a decomposition into 3 chains, two ascending ($[(4,1),(5,4),(6,7)]$ and $[(1,3),(2,6)]$ and one descending ($[(7,2),(3,5)]$).

First, we show how to represent a chain using bitmaps that support rank, select and access operations.

Definition 3.1. *A chain* $[(x_1,y_1),(x_2,y_2)\ldots,(x_n,y_n)]$ *is ascending iff* $x_i \leq x_{i+1}$, $1 \leq i < n$ *and* $y_i \leq y_{i+1}$, $1 \leq i < n$.

From this definition it is easy to prove our first result, stated in the following lemma. In order to present this result in a general way we use $S(n,m)$ as the space (in bits) required for representing a bitmap of length n with m ones that supports rank in t_r, select in t_s, and access in t_a time. We use t_b as $max(t_r,t_s,t_a)$.

Lemma 3.24. *Given an ascending chain* $\mathcal{C} = [(i_1,j_1),(i_2,j_2),\ldots,(i_m,j_m)]$, *of length* m, *where the values do not exceed* n, *we can represent the chain in* $2S(n,m)$ *bits and support the following queries:*

- $\text{getj}_{\mathcal{C}}(i)$*: gets* j *such that* $(i,j) \in \mathcal{C}$ *or* $\perp$ *if such pair does not exist. We also define* $\text{geti}_{\mathcal{C}}(j)$ *in an analogous way. Both queries are supported in time* $O(t_b)$.
- $\text{range}_{\mathcal{C}}(i_1,i_2,j_1,j_2)$*: find the* $(i,j) \in \mathcal{C}$ *such that* $(i,j) \in [i_1,i_2]\times[j_1,j_2]$ *or* $\perp$ *if such point does not exist. This runs in time* $O(t_b)$.

Proof. The chain $\mathcal{C}$ corresponds to two arrays: $I = [i_1,i_2,\ldots,i_m] \subseteq [n]$ and $J = [j_1,j_2,\ldots,j_m] \subseteq [n]$. We represent each array using an indexed bitmap supporting rank, select, and access. We call B_I the bitmap representing I and B_J the one representing J. The space is $2S(n,m)$, we just need to show how to support the required operations.

Get operations. We show how to support $\text{getj}_{\mathcal{C}}(i)$, the other *get* operation is analogous.

$$\text{getj}_{\mathcal{C}}(i) = \begin{cases} \perp & \text{if } I[i] = 0 \\ \texttt{select}_{B_J}(\texttt{rank}_{B_I}(i)+1) & \text{otherwise} \end{cases}$$

Range operation. Given a rectangle, we want to know whether the chain intersects the rectangle. In case the intersection exists, we would like to know whether there are any points inside the query rectangle.

The first simple fact we take into consideration is that an ascending chain $\mathcal{C}$ is going to intersect a query rectangle $[i_1, i_2] \times [j_1, j_2]$ if and only if $\mathcal{C}$ intersects the line segment defined by (i_2, j_1) and (i_1, j_2)[5]. We only need to answer if the chain intersects a segment given by two points inside the grid. To determine whether a chain $\mathcal{C}$ intersects a line segment defined by (i_2, j_1) and (i_1, j_2) we look for the predecessor in $\mathcal{C}$ whose i coordinate is less than i_1 and for the successor of i_2 in $\mathcal{C}$. This is easy to answer with a combination of `rank` and `select` queries. If the two points obtained do not intersect the segment, then the chain does not. Otherwise, the chain intersects the rectangle. □

We can represent each chain using Lemma 3.24, this leads to the following theorem:

Lemma 3.25. *Let m_i be the number of elements in chain i. The total space of the structure for a permutation that can be decomposed into χ chains is $2\sum_{i=1}^{\chi} S(n, m_i)$, and supports range queries in $O(t_b \lg \chi + t_b \chi' + k)$, where χ' is the number of chains that intersect the range. The next table summarizes some of the tradeoffs we can achieve.*

Bitmap Representation	*Total Space*	t_b
Raman et al. [101]	$2n \lg \chi + O\left(\chi \lg n + \frac{\chi \lg \lg n}{\lg n}\right)$	$O(1)$
Pătraşcu [99]	$2n \lg \chi + O\left(\chi \lg n + \frac{\chi n}{\lg^c n}\right)$	$O(c)$
Okanohara and Sadakane [98]	$2n \lg \chi + O(\chi \lg n + n)$	$O\left(\lg \frac{n}{m_i} + \frac{\lg^4 m_i}{\lg n}\right)$

The complexity for range queries derives from the work by Arroyuelo et al. [3]. The computation of the final space is similar to the one used in proof of Theorem 3.6.

One problem with this representation is the cost of the operations π and π^{-1}. If we answer these queries using the machinery for range queries we end up with a tradeoff that is not attractive at all, since $\chi' = \chi$. A simple work-around is to pay some extra space. As we will see, solving this problem and reducing the lower order term of our structure converges to a previously known structure [12]. This new way of arriving at the representation shows something not explored in the original proposal: how to solve orthogonal range queries.

The simplest way to support π and π^{-1} is to keep two arrays S_π and $S_{\pi^{-1}}$. They store the chain in which each position is contained in the structure. For example, $S_\pi[i] = j$ iff the coordinate i is

[5] We make all chains enter and exit the grid with dummy points to make sure a chain does not end inside a query.

contained in chain j. S_π and $S_{\pi^{-1}}$ require $n \lg \chi$ bits each, thus the space is almost doubled. Yet they allow us to support π and π^{-1} in $O(t_b)$ time.

If we aim at reducing the lower order term, which makes sense when we do not expect χ to be too small [6], we need the following simple observation.

Observation 3.1. *Given a set of χ bitmaps of length n, where the total number of ones in the set is n and no two bitmaps contain a 1 in the same position, we can represent them as a sequence of length n over an alphabet of size χ. Furthermore, any sequence representation supporting rank, select, and access in times t_r, t_s, and t_a, allows us to support the same operations in each individual bitmap within the same time.*

It is interesting that we can represent our structure using two sequences (x and y coordinates), and they correspond exactly to S_π and $S_{\pi^{-1}}$. Furthermore, they also correspond to the representation proposed by Barbay and Navarro [12], which was originally proposed using wavelet trees, but can be modified to work with any representation, offering a wider set of tradeoffs. The most interesting tradeoff at the time of this writing if offered by the representation of Barbay at el. [8, 7].

Another point to highlight, is that this shows that the original structure of Barbay and Navarro also supports adaptive range searching. This particular searching algorithm has proven to be efficient in practice [30]. This allows to state the following corollary.

Corollary 3.1. *Given a permutation Π, that can be decomposed into χ monotonically ascending and descending chains. And also given a sequence representation that requires $S(n, \sigma)$ for representing a sequence of length n over an alphabet of size σ, supporting rank, select and access queries in $O(t_b)$ time. There exists a structure requiring $2S(n, \chi)$ bits that supports computing π and π^{-1} in $O(t_b)$ and range search queries in $O(t_B \lg \chi + t_b \chi' + k)$, where χ' is the number of chains that touch the query rectangle, and k the size of the output.*

3.6.3 Extending to Binary Relations

We use the same approach on the grid representing the binary relation. Given a binary relation $\mathcal{R}$, the pair (i, j) is marked iff i relates to j in $\mathcal{R}$. We follow the notation of the previous sections. Recall that σ is the number of rows, n the number of columns, and t the number of pairs in $\mathcal{R}$.

We assume all columns and rows have at least one element, we can trivially map the problem when we accept empty row/columns adding a bitmap of length $n + \sigma$ supporting rank, select, and access.

We focus mostly on three operations: `rel_access` and iterating over consecutive elements in `rel_access_label_major`/`rel_access_object_major`.

The technique presented in Section 3.6.2 does not apply directly to this case. The main problem is that a chain could contain many elements that are in the same row or column, and would

[6] It is also an important issue with the structure from Lemma 3.25, as the worst case can be far above the information theoretical lower bound.

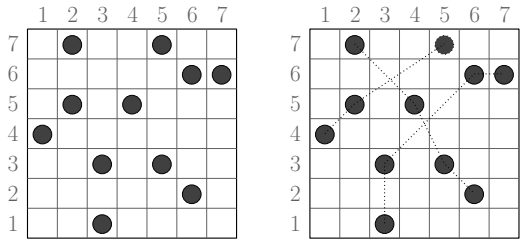

Figure 3.10: Example of binary relation represented in a grid (left) and a possible decomposition into ascending/descending chains (right).

result in multiple chains in the same position in a bitmap. We first give a representation that matches the result for the permutations in time and space and then we show how to potentially improve the space by using a more elegant technique. This new approach has a worse query time.

Using Permutations

We show how to transform a binary relation into a permutation by just considering a simple row/column addition algorithm that moves points around and allows one to answer the queries of interest.

The main idea is to create multiples copies of rows and columns having more than one point and then distribute the points across them so that each of them has only one point, leading to a permutation on t elements. This is inefficient in terms of space, but it allows us to match the performance of our structure for permutations. Algorithm 3 shows the procedure for converting a binary relation $\mathcal{R}$ into a permutation Π of length t. In order to be able to extract the original information we need to add $2t + o(t)$ bits, stored in B_1 and B_2. These bitmaps tell the length, in unary, of each expanded row/column.

For example, consider the binary relation in Figure 3.10, the bitmap B_2 is 1**0**1011010101010**1**1, the highlighted section corresponds to the number of labels related to objects 1 and 8. We do the same for the labels and store it in X_b.

Using these two bitmaps, we can answer $\texttt{label\_count}(1, \sigma, x, x)$ and $\texttt{object\_count}(\alpha, \alpha, 1, n)$ in constant time.

- $\texttt{label\_count}(1, \sigma, x, x) = X_a.\texttt{select}(1, x+1) - X_a.\texttt{select}(1, x) - 1$
- $\texttt{object\_count}(\alpha, \alpha, 1, n) = X_b.\texttt{select}(1, \alpha+1) - X_b.\texttt{select}(1, \alpha) - 1$

Lemma 3.26. *When we apply Algorithm 3 on a binary relation $\mathcal{R} \subseteq \{(i, j) | i \in [\sigma], j \in [n]\}$ where $t = |\mathcal{R}|$, we obtain a permutation Π over $[t]$.*

Data: $\mathcal{R}$
Result: Π, B_1, B_2
$\bar{\mathcal{R}} \leftarrow$ empty binary relation of size $t \times n_2$
$row \leftarrow 1$
$B_1 \leftarrow 0^t$
for $i \leftarrow 1$ *to* σ **do**
  $B_1[row] \leftarrow 1$
  for j *such that* $(i, j) \in \mathcal{R}$ **do**
    add (row, j) to $\bar{\mathcal{R}}$
    $row \leftarrow row + 1$
$col \leftarrow 1$
$B_2 \leftarrow 0^t$
for $j \leftarrow 1$ *to* n **do**
  $B_2[col] \leftarrow 1$
  for i *such that* $(i, j) \in \bar{\mathcal{R}}$ **do**
    $\Pi[i] = col$
    $col \leftarrow col + 1$
return (Π, B_1, B_2)

Algorithm 3: Transform($\mathcal{R}$)

Proof. It is clear that variables row, col, i and j never exceed t in the execution and that we generate t elements in Π. We only need to prove that no two indexes i_1, i_2, where $i_1 \neq i_2$, satisfy $\Pi[i_1] = \Pi[i_2]$. By contradiction, assume it happens, we have two indexes i_1, i_2 where $i_1 < i_2$ and $\Pi[i_1] = \Pi[i_2]$. But every time we assign cols value to Π, col gets incremented, so that is not possible, reaching a contradiction. □

This allows to state a theorem similar to the one presented by Barbay et al. [9], but supporting a different subset of operations.

Theorem 3.5. *A binary relation $\mathcal{R} \subseteq \{(i, j) | i \in [\sigma], j \in [n]\}$, where $t = |\mathcal{R}|$, that can be decompossed into k monotonic chains, can be represented as a permutation of length t with $(n + \sigma)(1 + o(1))$ extra bits. Furthermore, the resulting permutation can be decompossed into χ monotonic chains, and iterating over the results of* `rel_access_label_major` *and* `rel_access_object_major` *can be mapped to π and π^{-1}. The* `rel_access` *operation can be solved using $\mathcal{R}$. The counting operations can be solved using bitmaps B_1 and B_2 obtained from Algorithm 3.*

Proof. The theorem is a direct consequence of our permutation representation, plus the fact that we can map $\mathcal{R}$ to a permutation.

The main idea for solving the queries is similar to that of Barbay et al.'s [9]. We have to map the queries to a set of elements, and the ranges, using B_1 and B_2, which reduces to a simple combination of binary `rank` and `select` queries.

In order to prove that both have the same value for χ, consider only descending chains. Algorithm 3 would make two points in the same column compatible, since it displaces the lower point. We can do the same for ascending chains (it corresponds to expending in the other order), thus we only need to modify the algorithm to take both cases into account. This is not hard, provided the fact that we know the direction of the chains to which each point belongs in the original grid. □

Using Chains Directly

An alternative method can be obtained by decomposing the binary relation directly. A chain could now contain more than one occurrence of a given row or column, and because of that, the transformation that converges to the structure by Barbay and Navarro does not work. At this point, the departure from the original proposal by Barbay and Navarro pays off, allowing the representation of a class wider than that of permutations.

We first take a look at the space consumption of our structure when decomposing $\mathcal{R}$ into chains. For that, we present an alternative representation for the chains. For simplicity, we will use the bitmaps representation by Pătraşcu [99], yet the results translate in a similar way as for Lemma 3.25, and thus, we can offer a wide set of bounds.

Lemma 3.27. *An ascending chain of length m with at most $\bar{n} = n + \sigma$ points in $[n] \times [\sigma]$ can be represented in $2m \lg \frac{\bar{n}}{m} + O\left(m + \frac{\bar{n}}{\lg^c n}\right)$ bits of space, but now* geti, getj *and* range *take $O(c \lg m)$ time.*

Proof. We represent the chains using two bitmaps, as in Lemma 3.24, but representing each element c_i as $c_i + i$, this allows for duplicates in the bitmap and select remains $O(1)$. Using the representation of Pătraşcu [99] we obtain $m \lg \bar{n}/m + O(\bar{n}/\lg^c n)$ bits. The main problem is rank, that requires $O(c \lg m)$ time now, obtained by doing a binary search over selects queries. □

If we represent the structure using these chains, we obtain the following theorem:

Theorem 3.6. *A binary relation $\mathcal{R}$ over $[n] \times [\sigma]$, where $t = |\mathcal{R}|$, can be represented in $2t \lg \frac{n\chi}{t} + 2t \lg \chi + O(\frac{\chi n}{\lg^c n} + \chi \lg t)$ bits. The time for listing elements for* `rel_access_object_major` *and* `rel_access_label_major` *is $O(r)$ per datum retrieved, and the time for answering range queries is $O(r(\lg \chi + \chi') + k)$, where χ is the number of chains, χ' the number of chains hitting the query rectangle, k the size of the output, and $r = \max(c \lg \chi, c \lg \lg n)$.*

Proof. Let m_i the number of elements in chain $i \in [\chi]$. The total space of the representation, in bits, is bounded by:

$$\begin{aligned}
\sum_{1 \le i \le \chi} 2m_i \lg \frac{4n}{m_i} + \sum_{1 \le i \le \chi} O\left(\lg t + m_i + \frac{n}{\lg^c n}\right) &= \sum_{1 \le i \le \chi} 2m_i \lg \frac{n}{m_i} + O\left(t + \chi \lg t + \frac{\chi n}{\lg^c n}\right) \\
\le \sum_{1 \le i \le \chi} 2t/\chi \lg \frac{n\chi}{t} + O\left(t + \chi \lg t + \frac{\chi n}{\lg^c n}\right) &= 2t \lg \frac{n\chi}{t} + O\left(t + \chi \lg t + \frac{\chi n}{\lg^c n}\right)
\end{aligned}$$

The queries are answered in the same way as for the permutations, but with a logarithmic penalty factor in the worst case. However, we can improve the time for rank, and thus the queries, by sampling the rank positions evenly every $\frac{\chi n \lg n}{t} \leq \chi \lg n$ positions. This adds an extra space of $O(t)$ and allows to support rank in $O(c \lg \chi + c \lg \lg n)$ time.

The extra $2t \lg \chi$ bits are for S_π and $S_{\pi^{-1}}$, now we need this two sequences in order to support rel_access_label_major and rel_access_object_major efficiently. We can trade that space for time, then iterating through a row or column becomes more expensive. In case we opt for this smaller version, the dominant term becomes $2t \lg \frac{n\chi}{t}$, which is adaptive and close to the entropy. □

Note that we could try merging the sequences marking with a bitmap where each position starts, and this would lead to the result obtained in Theorem 3.5.

An interesting observation of the representation presented in Theorem 3.6 is that we can answer range minimum queries (RMQs) over the binary relation in the same time as for relrange. This comes from the fact that each chain can be seen as a one dimensional set of elements, and thus we can solve RMQs in constant time inside each chain by only adding $O(m_i)$ bits (obtain the position of the smallest element, not its value) [49]. This does not add any extra space (asymptotically), but we have to include the weight of the elements in order to retrieve their value. We cannot avoid this, since we need to compare χ' of them.

Lemma 3.28. *By adding $O(t)$ extra bits to the representation from Lemma 3.25, or Theorems 3.5 and 3.6, and adding weights to each pair in the permutation/relation, we can support range minimum queries in the same complexity as the one required for answering* rel_access.

3.6.4 Applications

In the following subsections we motivate some applications of our results to well known problems.

A Simple Text Self-Index

Consider the suffix array A of a text T of length n, drawn from an alphabet $\Sigma = [\sigma]$. By representing A with the structure from Lemma 3.25 and adding $n + o(n)$ extra bits, we get a structure that replaces the text and supports pattern matching over it.

Lemma 3.29. *Given a text T of length n over an alphabet $\Sigma = [\sigma]$, and given its suffix array A represented with the structure from Lemma 3.25, by adding a bitmap Occ, counting in unary the number of occurrences of each symbol in T, we can retrieve any position of T by using A and Occ.*

Proof. The suffix array A is sorted alphabetically, so all suffixes starting with a symbol s appear in a contiguous range in A. Given a position p, by computing $p_A = \pi^{-1}(p)$, we obtain the position

where p appears in A. Then, by computing $s = \mathtt{rank}_{Occ}(p_A)$ we obtain the symbol at position p. □

Extracting any snippet of length ℓ is supported in $O(\ell)$ applications of π^{-1}. We also have access to the suffix array in the same time as π requires. We can count the occurrences of a pattern P of length m in $O(\lg n)$ calls to π plus $O(m)$ calls to π^{-1} [92]. We can retrieve each position where P appears by applying π (after counting). We can also support range restricted pattern matching using the same approach combined with range queries.

If we apply this same idea to Sadakane's Ψ permutation [92] we get an alternative tradeoff and guarantee that $k \leq \sigma$. This makes the index attractive, since its worst case space requirement is in terms of the plain representation of the text instead of the suffix array.

Inverted Lists

One example of applications of binary relations are inverted lists. We commented briefly on this at the beginning of this chapter. We now re-visit them using our adaptive representation. Given a collection of n documents D, written over a vocabulary of size $m < n$, we can represent the inverted lists with our structure from Theorem 3.6 (or Theorem 3.5 to obtain an alternative time and space bound). We denote as t the sum of total distinct words in each document, allowing repetitions among documents.

Consider the typical query where we want all the elements that contain a given word. This is simply the elements related to a given word identifier in the binary relation, which by Theorem 3.6 can be retrieved in $O(\max(\lg \chi, \lg \lg n))$ time per element. A more interesting case is when we want to retrieve all documents containing two words w_1 and w_2. We have two lists and we need to compute the intersection among them. This can be done in $O(\min(\mathtt{object\_count}(w_1, w_1, 1, n), \mathtt{object\_count}(w_2, w_2, 1, n)) \lg k \max(\lg k, \lg \lg n))$ time using the `rel_access` operation.

Lemma 3.30. *Given n documents over a vocabulary of size m, where there are t words used in the whole collection, we can obtain the same navigation operations and space requirements as the ones shown in Theorem 3.6. We can also support intersection of two rows i_1 and i_2 in time $O(m_1 r \lg \chi)$, where $m_1 = \min(\mathtt{object\_count}(i_1, i_1, 1, n), \mathtt{object\_count}(i_2, i_2, 1, n))$, and $r = \max(\lg \chi, \lg \lg n)$. The symmetric case works in a similar way*

Proof. We only need to prove the complexity of the intersection query. For that we just consider all the elements in the row with less elements. For each of those elements we search for that given element in the other row, and $\chi' \leq 1$, since only one chain goes through the point if it exists. □

Another interesting result is when we consider a simple stemming where all words with the same root are neighbours (contiguous rows). By supporting RMQs queries in time T, we can retrieve the k most relevant documents in an orthogonal range in $O(Tk \lg k)$ time. We can apply the same idea to retrieve the k (unique) words that are contained in the k most relevant documents in that range.

3.7 Concluding Remarks

We presented a thorough study of a wide set of operations over binary relations and how they relate to each other. Using this we proposed a framework that allows to extend the functionality of existing representations by means of simple reductions.

We analyzed five representations under our framework, providing the most complete set of structures for representing general-purpose binary relations efficiently. We also showed applications of such representations to well known problems, like text indexing and representation of inverted indexes.

Our representation BinRel-GWT presents an extension to the $n \times n$ grid representation by Bose et al [18]. This structure allows to speed up well known operations like *prevLess* and *range_next_value* [77, 50]. This has direct consequences for text indexing and structures used in information retrieval.

Operation	BinRel-GWT	BinRel-WT
rel_count(α,β,x,y)	$O(\lg\sigma/\lg\lg n)$	$O(\lg\sigma)$
rel_rank(α,x)	$O(\lg\sigma/\lg\lg n)$	$O(\lg\sigma)$
rel_rank_label_major(α,x,y,z)	$O(\lg\sigma/\lg\lg n)$	$O(\lg\sigma)$
rel_select_label_major(α,j,x,y)	$O(\lg\sigma)$	$O(\lg\sigma)$
rel_min_label_major(α,x,y,z)	$O(\lg\sigma/\lg\lg n+\lg\lg n)$	$O(\lg\sigma)$
rel_rank_object_major(α,β,γ,x)	$O(\lg\sigma/\lg\lg n)$	$O(\lg\sigma)$
rel_select_object_major(α,β,x,j)	$O(\lg j\lg(\beta-\alpha+1)\frac{\lg\sigma}{\lg\lg n})$ $O(\lg n\lg\sigma/\lg\lg n)$	$O(\lg j\lg(\beta-\alpha+1)\lg\sigma)$ $O(\lg n\lg\sigma)$
rel_min_object_major(α,β,γ,x)	$O(\lg\sigma/\lg\lg n)$	$O(\lg\sigma)$
rel_access(α,β,x,y)	$O((k+1)\lg\sigma/\lg\lg n)$	$O((k+1)\lg\sigma)$
label_count(α,β,x,y)	$O(\beta-\alpha+\lg\sigma/\lg\lg n)$	$O(\beta-\alpha+\lg\sigma)$
label_rank(α,x,y)	$O(\alpha+\lg\sigma/\lg\lg n)$	$O(\alpha+\lg\sigma)$
label_select(α,j,x,y)	$O(j(\lg\sigma/\lg\lg n+\lg\lg n))$	$O(j\lg\sigma)$
label_access(α,x,y)	$O((k+1)(\frac{\lg\sigma}{\lg\lg n}+\lg\lg n))$	$O((k+1)\lg\sigma)$
label_min(α,x,y)	$O(\lg\sigma/\lg\lg n+\lg\lg n)$	$O(\lg\sigma)$
object_count(α,β,x,y)	$O((y-x+1)\lg\sigma/\lg\lg n)$	$O((y-x+1)\lg\sigma)$
object_rank(α,β,x)	$O((y-x+1)\lg\sigma/\lg\lg n)$	$O((y-x+1)\lg\sigma)$
object_select(α,β,x,j)	$O(j\lg\sigma/\lg\lg n)$	$O(j\lg\sigma)$
object_access(α,β,x)	$O((k+1)\lg\sigma/\lg\lg n)$	$O((k+1)\lg\sigma)$
object_min(α,β,x)	$O(\lg\sigma/\lg\lg n)$	$O(\lg\sigma)$
label_rank1(α,x)	$O(\lg\sigma/\lg\lg n)$	$O(\lg\sigma)$
label_select1(α,j,x)	$O(\lg\sigma)$	$O(\lg\sigma)$
label_min1(α,j,x)	$O(\lg\sigma/\lg\lg n+\lg\lg n)$	$O(\lg\sigma)$
label_access1(α,j,x)	$O((k+1)\lg\sigma/\lg\lg n)$	$O((k+1)\lg\sigma)$
object_rank1(α,x)	$O(\lg\sigma/\lg\lg n)$	$O(\lg\sigma)$
object_select1(α,x,j)	$O(\lg\sigma/\lg\lg n)$	$O(\lg\sigma)$
object_min1(α,x,j)	$O(\lg\sigma/\lg\lg n)$	$O(\lg\sigma)$
object_access1(α,x,j)	$O((k+1)\lg\sigma/\lg\lg n)$	$O((k+1)\lg\sigma)$

Table 3.2: Time complexity for the operations for BinRel-GWT and BinRel-WT. The parameter k represents the size of the output for the access operators; one can consider $k = 1$ for the reductions shown in Theorem 3.1.

Operation	BRWT	BinRel-WT
rel_count(α,β,x,y)	$O(\beta-\alpha+\lg\sigma)$	$O(\lg\sigma)$
rel_rank(α,x)	$O(\alpha+\lg\sigma)$	$O(\lg\sigma)$
rel_rank_label_major(α,x,y,z)	$O(\alpha+\lg\sigma)$	$O(\lg\sigma)$
rel_select_label_major(α,j,x,y)	$O(j\lg\sigma)$	$O(\lg\sigma)$
rel_min_label_major(α,x,y,z)	$O(\lg\sigma)$	$O(\lg\sigma)$
rel_rank_object_major(α,β,γ,x)	$O(\beta-\alpha+\lg\sigma)$	$O(\lg\sigma)$
rel_select_object_major(α,β,x,j)	$O(j\lg\sigma)$	$O(\lg j\lg(\beta-\alpha+1)\lg\sigma)$ $O(\lg n\lg\sigma)$
rel_access(α,β,x,y)	$O((k+1)\lg\sigma)$	$O((k+1)\lg\sigma)$
rel_min_object_major(α,β,γ,x)	$O(\lg\sigma)$	$O(\lg\sigma)$
label_count(α,β,x,y)	$O(\beta-\alpha+\lg\sigma)$	$O(\beta-\alpha+\lg\sigma)$
label_rank(α,x,y)	$O(\alpha+\lg\sigma)$	$O(\alpha+\lg\sigma)$
label_select(α,j,x,y)	$O(j\lg\sigma)$	$O(j\lg\sigma)$
label_access(α,x,y)	$O((k+1)\lg\sigma)$	$O((k+1)\lg\sigma)$
label_min(α,x,y)	$O(\lg\sigma)$	$O(\lg\sigma)$
object_count(α,β,x,y)	$O((y-x+1)\lg\sigma)$	$O((y-x+1)\lg\sigma)$
object_rank(α,β,x)	$O((y-x+1)\lg\sigma)$	$O((y-x+1)\lg\sigma)$
object_select(α,β,x,j)	$O(j\lg\sigma)$	$O(j\lg\sigma)$
object_access(α,β,x)	$O((k+1)\lg\sigma)$	$O((k+1)\lg\sigma)$
object_min(α,β,x)	$O(\lg\sigma)$	$O(\lg\sigma)$
label_rank1(α,x)	$O(\lg\sigma)$	$O(\lg\sigma)$
label_select1(α,j,x)	$O(j\lg\sigma)$	$O(\lg\sigma)$
label_min1(α,j,x)	$O(\lg\sigma)$	$O(\lg\sigma)$
label_access1(α,j,x)	$O((k+1)\lg\sigma)$	$O((k+1)\lg\sigma)$
object_rank1(α,x)	$O(\lg\sigma)$	$O(\lg\sigma)$
object_select1(α,x,j)	$O(\lg\sigma)$	$O(\lg\sigma)$
object_min1(α,x,j)	$O(\lg\sigma)$	$O(\lg\sigma)$
object_access1(α,x,j)	$O((k+1)\lg\sigma)$	$O((k+1)\lg\sigma)$

Table 3.3: Time complexity for the operations for BRWT and BinRel-WT. The parameter k represents the size of the output for the access operators; one can consider $k = 1$ for the reductions shown in Theorem 3.1.

4 Grammar-Based Indexes

Although the origins of grammar-based compression dates back to the seventies, it is still a very active area of research. From the different variants of the idea we focus on the case in which a given text $T[1,u]$ is replaced by a context- free grammar (CFG) $\mathcal{G}$, composed of n rules, that generates just the string T . One then can store $\mathcal{G}$ instead of T, thereby possibly achieving compression. The term grammar-based compression was introduced explicitly [79] by Kieffer and Yang [74] and Nevill- Manning [97]. Some grammar-based compressors are Sequitur [96] and the algorithm proposed by Apostolico and Lonardi [2]. Some well known compression algorithms have an easy mapping into a grammar, for example, LZ78 [119] and Re-Pair [78].

When a CFG deriving a single string is converted into Chomsky Normal Form, the result is called a *Straight-Line Program (SLP)*. This is a grammar where each nonterminal appears on the left-hand side of a unique rule, which defines it, and can be converted into either a terminal or the concatenation of two previously defined nonterminals. SLPs are as powerful as CFGs for compression purposes, up to a constant factor. In particular the three grammar-based compression methods listed above can be straightforwardly translated, with no significant penalty, into SLPs.

Grammar-based methods can achieve universal compression [74]. They belong to the wider class of textual substitution methods [111, 2, 73], which exploit repetitions in the text rather than frequencies. Textual substitution methods are particularly suitable for compressing *highly repetitive strings*, meaning strings containing a high degree of long identical substrings, not necessarily close to each other. Such texts arise in applications like computational biology, software repositories, transaction logs, versioned documents, temporal databases, etc.

A well-known textual substitution method that is more powerful than any grammar-based compressor is LZ77 [118]. Yet, SLPs are still able to capture most of the redundancy of highly repetitive strings, and are in practice competitive with the best compression methods [52]. In addition, they decompress in linear time and can decompress arbitrary substrings almost optimally. The latter property, not achieved on LZ77, is crucial for implementing compressed text databases, as we discuss next.

Finding the smallest SLP that represents a given text $T[1,u]$ is NP-hard [103, 22]. Moreover, some popular grammar-based compressors such as LZ78, Re-Pair and Sequitur, can generate a compressed file much larger than the smallest SLP [22]. Yet, a simple method to achieve an $O(\lg u)$-approximation is to parse T using LZ77 and then to convert it into an SLP [103], which in addition is *balanced*: the height of the derivation tree for T is $O(\lg u)$. (Also, any SLP can be balanced by paying an $O(\lg u)$ space penalty factor.)

In this chapter, we present an index for arbitrary SLPs. This index extends any SLP-compressed text into a text index that supports searching and extracting arbitrary substrings. Later, we extend this result to a more general class of grammars.

The main contributions of this chapter are:

- We present the *first* grammar-based representation of texts that can support operations `extract` and `find` (see Chapter 1) in $o(n)$ time. That is, a grammar-based self-index. Given an SLP with n rules that generates a text, a plain representation of the grammar takes $2n \lg n$ bits, as each new rule expands into two other rules. Our self-index takes $O(n \lg n) + n \lg u$ bits. It can carry out *extract* in time $O((m+h) \lg n / \lg \lg n)$, where h is the height of the derivation tree, and `find` in time $O((m(m+h) + h\,occ) \frac{\lg n}{\lg \lg n})$ (see the detailed results in Theorem 4.5 and Corollary 4.1). There are faster solutions for the extraction problem, that takes time $O(m + \lg u)$, yet that solutions use $O(n \lg u)$ bits of space [16]. On the other hand, no previous SLP representation has achieved $o(n)$ search time.
- A part of our index is a representation of SLPs which takes $2n \lg n(1 + o(1))$ bits and is able to retrieve any rule in time $O(\lg n / \lg \lg n)$. It is also capable of answering other queries on the grammar, such as finding the rules mentioning a given non-terminal within the same time. This extends the results of Chapter 3 to representing labeled binary relations.
- Our self-index can be particularly relevant on highly repetitive text collections, as already witnessed by some preliminary experiments [27]. Our method is independent of the way the SLP is generated, and thus it can be coupled with different SLP construction algorithms, which might fit different applications.
- We present the first index that supports searches in sublinear time and does not depend on the height of the grammar. Combined with the result of Bille et al. [16], this provides the first linear space representation of a grammar in which both the locate and extract operations do not depend on the height of the grammar.

4.1 Straight-Line Programs

We now define a Straight-Line Program (SLP) and highlight some properties.

Definition 4.1. *[71] A* Straight-Line Program (SLP) $\mathcal{G} = (X = \{X_1, \ldots, X_n\}, \Sigma)$ *is a grammar that defines a single finite sequence* $T[1,u]$*, drawn from an alphabet* $\Sigma = [1, \sigma]$ *of terminals. It has n rules, which must be of the following types:*

- $X_i \to \alpha$*, where* $\alpha \in \Sigma$*. It represents string* $\mathcal{F}(X_i) = \alpha$*.*
- $X_i \to X_l X_r$*, where* $l, r < i$*. It represents string* $\mathcal{F}(X_i) = \mathcal{F}(X_l)\mathcal{F}(X_r)$*.*

We call $\mathcal{F}(X_i)$ *the* phrase generated *by nonterminal* X_i*, and* $T = \mathcal{F}(X_n)$*.*

Definition 4.2. *[103] The* height *of a symbol* X_i *in the SLP* $\mathcal{G} = (X, \Sigma)$ *is defined as* $\texttt{height}(X_i) = 1$ *if* $X_i \to \alpha \in \Sigma$*, and* $\texttt{height}(X_i) = 1 + \max(\texttt{height}(X_l), \texttt{height}(X_r))$ *if* $X_i \to X_l X_r$*. The height of the SLP is* $\texttt{height}(\mathcal{G}) = \texttt{height}(X_n)$*. We will refer to* $\texttt{height}(\mathcal{G})$ *as h when* $\mathcal{G}$ *is clear from the context.*

As some of our results will depend on the height of the SLP, it is interesting to recall the following theorem, which establishes the cost of balancing an SLP.

Theorem 4.1. *[103] Let an SLP $\mathcal{G}$ generate text $T[1,u]$ with n rules. We can build an SLP $\mathcal{G}'$ generating T, with $n' = O(n \lg u)$ rules and* $\texttt{height}(\mathcal{G}') = O(\lg u)$ *in $O(n \lg u)$ time.*

Finally, as several grammar-compression methods are far from optimal with respect to the size of the smallest grammar [22], it is interesting that one can find in linear time a reasonable (and balanced) approximation.

Theorem 4.2. *[103] Let $\mathcal{G}$ be the minimal SLP generating text $T[1,u]$ over integer alphabet, with n rules. We can build in $O(u)$ time an SLP $\mathcal{G}'$ generating T, with $O(n \lg u)$ rules and* $\texttt{height}(\mathcal{G}') = O(\lg u)$.

4.2 Labeled Binary Relations with Range Queries

In this section we introduce a data structure for labeled binary relations with range query capabilities. This is based on the result of Chapter 3.

Consider a binary relation $\mathcal{R} \subseteq A \times B$, where $A = \{1, 2, \ldots, n_1\}$, $B = \{1, 2, \ldots, n_2 \leq n_1\}$, a function $\mathcal{L} : A \times B \rightarrow L \cup \{\perp\}$, which maps every pair in $\mathcal{R}$ to a label in $L = \{1, 2, \ldots, \ell\}, \ell \geq 1$, and pairs not in $\mathcal{R}$ to $\perp$. We support the following queries:

- $\mathcal{L}(a, b)$ for $(a, b) \in A \times B$.
- $A(b) = \{a,\ (a, b) \in \mathcal{R}\}$.
- $B(a) = \{b,\ (a, b) \in \mathcal{R}\}$.
- $R(a_1, a_2, b_1, b_2) = \{(a, b) \in \mathcal{R},\ a_1 \leq a \leq a_2,\ b_1 \leq b \leq b_2\}$.
- $\mathcal{L}(l) = \{(a, b) \in \mathcal{R},\ \mathcal{L}(a, b) = l\}$.
- The sizes of the sets: $|A(b)|, |B(a)|$, $|R(a_1, a_2, b_1, b_2)|$, and $|\mathcal{L}(l)|$.

We now show how to compose two binary relation representations (as the ones in Chapter 3) in order to answer each query. One binary relation corresponds to $\mathcal{R}$, for which we will use the BinRel-GWT. This gives an interesting property to exploit further to chain $\mathcal{R}$ with its labels: each point in the relation can be identified by its position in the string represented in the BinRel-GWT. We augment the relation with two bitmaps counting, in unary, how many elements relate to each element in A and B respectively. We call these bitmaps X_A and X_B and they are defined as follows: $X_B = 0^{|B(1)|}10^{|B(2)|}1 \ldots 0^{|B(n_1)|}1$ and $X_A = 0^{|A(1)|}10^{|A(2)|}1 \ldots 0^{|A(n_2)|}1$. From our previous results in Chapter 3, using the BinRel-Str and BinRel-GWT structures, we can support the following operations:

A \ B	1	2	3
1	**1**	2	
2		2	2
3		**1**	

S_B	1	2		2	3		2	
$S_{\mathcal{L}}$	**1**	2		2	2		**1**	
X_B	0	0	1	0	0	1	0	1
X_A	0	1	0	0	0	1	0	1

Figure 4.1: Example of a labeled relation (left) and our representation of it (right). Labels are in bold and the elements of B are in normal font.

- $A(b)$ and $B(a)$ in $O(\lg n_2/\lg\lg n_2)$ time per query, plus $(\lg n_2/\lg\lg n_2)$ time per element retrieved.
- $R(a_1, a_2, b_1, b_2)$ in $O(\lg n_2/\lg\lg n_2)$ time per query, plus $O(\lg n_2/\lg\lg n_2)$ time per element retrieved.
- $|A(b)|$ and $|B(a)|$ in constant time.
- $|R(a_1, a_2, b_1, b_2)|$ in $O(\lg n_2/\lg\lg n_2)$ time.

The labels are represented as another binary relation, this one maps positions in BinRel-GWT to labels in L. We also use the BinRel-STR representation, and call this relation $\mathcal{R}_{\mathcal{L}}$. Using this, we can easily answer the following queries:

- $\mathcal{L}(a, b)$ in $O(acc(\ell))$ time, where $acc(x)$ is the time to access an element in the string representation, for an alphabet size x.
- $\mathcal{L}(l)$ in $O((k+1)sel(\ell) + k\lg n_2/\lg\lg n_2)$ for retrieving k elements, where $sel(x)$ corresponds to the time to select a given symbol in the string representation, for an alphabet of size x. This is achieved by retrieving the occurrences of the label in $\mathcal{R}_{\mathcal{L}}$. For each one of them we use the representation of $\mathcal{R}$ to track the pair (a, b).
- $|\mathcal{L}(l)|$ can be answered in $O(rnk(x))$ time by performing a rank operation over the sequence representing the binary relation between positions in $\mathcal{R}$ and L. Again, $rnk(x)$ corresponds to the time to do rank in a sequence of alphabet size x.

Figure 4.1 shows an example of how the sequences look. S_B corresponds to the sequence representing $\mathcal{R}$ and $S_{\mathcal{L}}$ to the sequence representing the second binary relation (labels to positions).

We note that, if we do not support queries $\mathcal{R}(a_1, a_2, b_1, b_2)$, we can use also the BinRel-STR representation for $\mathcal{R}$. We have thus proved the next theorem.

Theorem 4.3. *Let $\mathcal{R} \subseteq A \times B$ be a binary relation, where $A = \{1, 2, \ldots, n_1\}$, $B = \{1, 2, \ldots, n_2\}$, and a function $\mathcal{L} : A \times B \to L \cup \{\perp\}$, which maps every pair in $\mathcal{R}$ to a label in $L = \{1, 2, \ldots, \ell\}, \ell \geq 1$, and pairs not in $\mathcal{R}$ to $\perp$. Then $\mathcal{R}$ can be indexed using $(r + o(r))(\lg n_2 + \lg \ell + o(\lg \ell) + O(1)) + o(n_1 + n_2)$ bits*

of space, where $r = |\mathcal{R}|$. Queries can be answered in the times shown below, where k is the size of the output. One can choose (i) $rnk(x) = acc(x) = \lg\lg x$ and $sel(x) = 1$, or (ii) $rnk(x) = \lg\lg x \lg\lg\lg x$, $acc(x) = 1$ and $sel(x) = \lg\lg x$, independently for $x = \ell$ and for $x = n_2$.

Operation	*Time (with range)*	*Time (without range)*
$\mathcal{L}(a,b)$	$O(\lg n_2 + acc(\ell))$	$O(rnk(n_2) + sel(n_2) + acc(\ell))$
$A(b)$	$O(1 + k \lg n_2)$	$O(1 + k\, sel(n_2))$
$B(a)$	$O(1 + k \lg n_2)$	$O(1 + k\, acc(n_2))$
$\|A(b)\|, \|B(a)\|$	$O(1)$	$O(1)$
$R(a_1, a_2, b_1, b_2)$	$O((k+1)\lg n_2)$	—
$\|R(a_1, a_2, b_1, b_2)\|$	$O(\lg n_2)$	—
$\mathcal{L}(l)$	$O((k+1)sel(\ell) + k \lg n_2)$	$O((k+1)sel(\ell) + k\, acc(n_2))$
$\|\mathcal{L}(l)\|$	$O(rnk(\ell))$	$O(rnk(\ell))$

We note the asymmetry of the space and time with respect to n_1 and n_2, whereas the functionality is symmetric. This makes it always convenient to arrange that $n_1 \geq n_2$.

4.3 A Powerful SLP Representation

We provide in this section an SLP representation that supports various queries on the SLP within asymptotically the same space as a plain representation.

Recalling that Σ is the alphabet of the SLP and σ its size, it will usually be the case that all the symbols of Σ are used in the SLP.

We will assume that the rules of the form $X_i \to \alpha$ are lexicographically sorted, that is, if there are rules $X_{i_1} \to \alpha_1$ and $X_{i_2} \to \alpha_2$, then $i_1 < i_2$ if and only if $\alpha_1 < \alpha_2$. The SLP can obviously be reordered so that this holds. If for some reason we need to retain the original order, then $\sigma \lg \sigma$ extra bits are needed to record the rule reordering.

A plain representation of an SLP with n rules over the alphabet $[1, \sigma]$ requires at least $2(n - \sigma)\lceil \lg n \rceil + \sigma \lceil \lg \sigma \rceil \leq 2n \lceil \lg n \rceil$ bits. Based on our labeled binary relation data structure of Theorem 4.3, we now give an alternative SLP representation which requires asymptotically the same space, $2n \lg n + o(n \lg n)$ bits, and is able to answer a number of interesting queries on the grammar in $O(\lg n / \lg\lg n)$ time. This will be a key part of our indexed SLP representation.

We again regard a binary relation as a table where the rows represent the elements of set A and the columns the elements of B. In our representation, rows, columns, and labels correspond to nonterminals. Every row corresponds to a symbol X_l (set A) and every column to a symbol X_r (set B). Pairs (l, r) are related, with label i, whenever there exists a rule $X_i \to X_l X_r$. Since $A = B = L = \{1, 2, \ldots n\}$ and $|\mathcal{R}| = n$, the structure uses $2n \lg n + o(n \lg n)$ bits. Note that the function $\mathcal{L}$ is invertible, $|\mathcal{L}(l)| = 1$.

To handle the rules of the form $X_i \to \alpha$, we set up a bitmap $Y[1, n]$ so that $Y[i] = 1$ if and only if $X_i \to \alpha$ for some $\alpha \in \Sigma$. Thus we know $X_i \to \alpha$ in constant time because $Y[i] = 1$ and $\alpha =$

$rank_Y(1,i)$. The space for Y is $n+o(n)$ bits. This works because these rules are lexicographically sorted and all the symbols in Σ are used. We have already explained how to proceed otherwise.

This representation supports the following queries.

- *Access to rules:* Given i, find l and r such that $X_i \to X_l X_r$, or α such that $X_i \to \alpha$. If $Y[i] = 1$ we obtain α in constant time as explained. Otherwise, we obtain $\mathcal{L}(i) = \{(l,r)\}$ from the labeled binary relation, in $O(\lg n/\lg\lg n)$ time.
- *Reverse access to rules:* Given l and r, find i such that $X_i \to X_l X_r$, if any. This is done in $O(\lg n/\lg\lg n)$ time via $\mathcal{L}(l,r)$ (it returns $\perp$, there is no such X_i). We can also find, given α, the $X_i \to \alpha$, if any, in $O(1)$ time via $i = select_Y(1,\alpha)$.
- *Rules using a left/right symbol:* Given i, find those j such that $X_j \to X_i X_r$ (top left) or $X_j \to X_l X_i$ (right) for some X_l, X_r. The first is answered using $\{\mathcal{L}(i,r), r \in B(i)\}$ and the second using $\{\mathcal{L}(l,i), l \in A(i)\}$, in $O(\lg n/\lg\lg n)$ time per each j found.
- *Rules using a range of symbols:* Given $l_1 \le l_2, r_1 \le r_2$, find those i such that $X_i \to X_l X_r$ for any $l_1 \le l \le l_2$ and $r_1 \le r \le r_2$. This is answered in $O(\lg n/\lg\lg n)$ time per symbol retrieved using $\{\mathcal{L}(a,b), (a,b) \in \mathcal{R}(l_1,l_2,r_1,r_2)\}$.

Again, if the last operation is not provided, we can choose alternative (i) in Theorem 4.3, to achieve $O(\lg\lg n)$ time for all the other queries. Or, if we want to provide "access to rules" in constant time as a plain SLP representation, we choose (i) for $S_{\mathcal{L}}$ and (ii) for S_B, obtaining $O(\lg\lg n \lg\lg\lg n)$ time for the other operations.

Theorem 4.4. *An SLP $\mathcal{G} = (X = \{X_1,\ldots,X_n\}, \Sigma)$, $\Sigma = [1,\sigma]$, can be represented using $2n\lg n + o(\sigma + n\lg n)$ bits, such that all the queries described above (access to rules, reverse access to rules, rules using a symbol, and rules using a range of symbols) can be answered in $O(\lg n)$ time per delivered datum. If we do not support the rules using a range of symbols, time drops to $O(\lg\lg n)$, or to $O(1)$ for access to rules and $O(\lg\lg n \lg\lg\lg n)$ for the others.*

4.4 Indexable Grammar Representations

We now provide an SLP-based text representation that permits indexed search and random access. We assume our text $T[1,u]$, over alphabet $\Sigma = [1,\sigma]$, is represented with an SLP $\mathcal{G}$ of n rules.

We will represent $\mathcal{G}$ using a variant of Theorem 4.4, where we carry out some reordering of the rules. First, we will reorder all the rules in lexicographic order of the strings represented, that is, $\mathcal{F}(X_i) \le \mathcal{F}(X_{i+1})$ for all $1 \le i < n$. Therefore the columns of the binary relation will still represent X_r, yet remain lexicographically sorted by $\mathcal{F}(X_r)$. Instead, the rows will represent X_l sorted by *reverse* lexicographic order, that is lexicographically sorted by $\mathcal{F}(X_l)^{rev}$, where S^{rev} is string S read backwards. We will also store a permutation π, which maps reverse to direct

lexicographic ordering. This must be used to translate row positions to nonterminal identifiers (as these are sorted in direct lexicographical order). We use Munro et al.'s representation for π [86](see also Chapter 1), with parameter $\epsilon = \frac{\lg\lg n}{\lg n}$, so that π can be computed in constant time and π^{-1} in $O(\lg n/\lg\lg n)$ time, and the structure needs $n\lg n + O\left(\frac{n\lg\lg n}{\lg n}\right)$ bits of space.

With the SLP representation and π, the space is $3n\lg n + o(\sigma + n\lg n)$ bits. We add another $n\lceil\lg u\rceil$ bits for storing the lengths $|\mathcal{F}(X_i)|$ of all the nonterminals X_i. Note that our reordering preserves the lexicographic ordering of the rules $X_i \to \alpha$ needed for our binary relation based representation.

Figure 4.2 (left) gives an example of a grammar representation. For now disregard the arrows and shadings, which illustrate the extraction and search process.

4.4.1 Extraction of Text from an SLP

To expand a substring $\mathcal{F}(X_i)[j, j+\ell]$, we first find position j by recursively descending in the parse tree rooted at X_i. Let $X_i \to X_l X_r$, then if $|\mathcal{F}(X_l)| \geq j$ we descend to X_l, otherwise to X_r, in this second case looking for position $j - |\mathcal{F}(X_l)|$. This takes $O(\mathtt{height}(X_i)\lg n/\lg\lg n)$ time (where the $\lg n/\lg\lg n$ factor is the time for "access to rules" operation). On our way back from the recursion, if we return from the left child, we fully traverse the right child left to right, until outputting $\ell + 1$ terminals.

This takes in total $O((\mathtt{height}(X_i) + \ell)\lg n/\lg\lg n)$ time, which is at most $O((h+\ell)\lg n/\lg\lg n)$. This is because, on one hand, we follow both children of a rule at most ℓ times, as each time we do this we increase the number of symbols to output. On the other, at most two times per level it happens that we follow only one child of a node, as otherwise two of them would share the same parent, since all the nodes traversed at a level are consecutive.

Figure 4.2 (top-right) illustrates the extraction of a substring. Note that there are at most 2 cases per level where we follow one child, and at most ℓ cases where we follow both.

4.4.2 Searching for a Pattern in an SLP

The SLP search problem is to find all the occurrences of a pattern $P = p_1 p_2 \ldots p_m$ in the text $T[1, u]$ defined by an SLP of n rules. As in previous work [70], except for the special case $m = 1$, occurrences can be divided into *primary* and *secondary*. A primary occurrence in $\mathcal{F}(X_i)$, $X_i \to X_l X_r$, is such that it spans a suffix of $\mathcal{F}(X_l)$ and a prefix of $\mathcal{F}(X_r)$, whereas each time X_i is used elsewhere (directly or transitively in other nonterminals that include it) it produces secondary occurrences. In the case $P = \alpha$, we say that the only primary occurrence is at $X_i \to \alpha$ and the other occurrences are secondary.

Our strategy is to first locate the primary occurrences and then track all their secondary occurrences in a recursive fashion. To find primary occurrences of P, we test each of the $m-1$ possible partitions $P = P_l P_r$, $P_l = p_1 p_2 \ldots p_k$ and $P_r = p_{k+1} \ldots p_m$, $1 \leq k < m$. For each partition $P_l P_r$, we first find all those X_ls such that P_l is a suffix of $\mathcal{F}(X_l)$, and all those X_rs such that

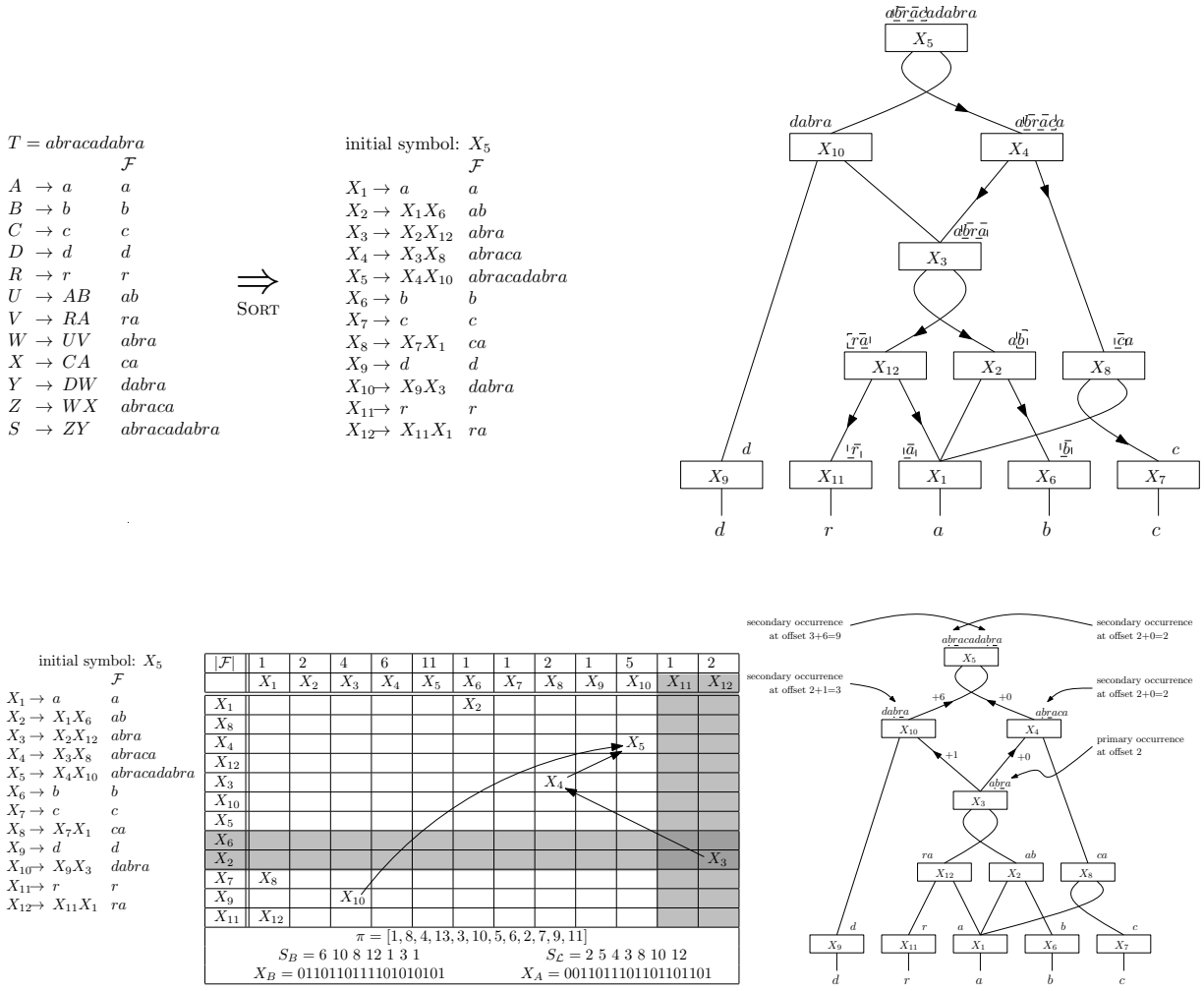

Figure 4.2: An example grammar for the text T = "abracadabra". On the top-left, the nonterminals are renamed according to their lexicographic order, so that A corresponds to X_1, U to X_2, and so on. On the top-right, the paths followed when extracting $T[2,5] = brac$. On the bottom-left, our data structure representing T. What the index stores is S_B, $S_{\mathcal{L}}$, X_B, X_A, π, and $|\mathcal{F}|$; all the rest is given for illustrative purposes (we omit bitmap $Y = 100001101010$). We also illustrate the search process for P = "br": We search the rows for the nonterminals finishing with "b" and the columns for the nonterminals starting with "r". The intersection contains X_3 (formerly W), where P has its only primary occurrence. The arrows show how we look for the rows and columns corresponding to X_3, to find out that it is used within X_4 (formerly Z) and X_{10} (formerly Y), and these in turn yield the two occurrences within X_5, the initial symbol. On the bottom-right we illustrate the process of extracting the secondary occurrences in the grammar. In the case of searching for a longer pattern, such as "bra", we would have to perform the same procedure for the 2 possible partitions: (b, ra) and (br, a).

P_r is a prefix of $\mathcal{F}(X_r)$. The latter form a lexicographic range $[r_1, r_2]$ in the $\mathcal{F}(X_r)$s, and the former a lexicographic range $[l_1, l_2]$ in the $\mathcal{F}(X_l)^{rev}$s. Thus, using our SLP representation, the X_is containing the primary occurrences correspond those labels i found between rows l_1 and l_2, and between columns r_1 and r_2, of the binary relation. Hence a query for *rules using a range of symbols* will retrieve each such X_i in $O(\lg n/\lg\lg n)$ time. If $P = \alpha$ our only primary occurrence is obtained in $O(1)$ time using *reverse access to rules*.

Now, for each primary occurrence at X_i, we track all the nonterminals that use X_i in their right hand sides. As we track the occurrences, we also maintain the *offset* of the occurrence within the nonterminal. This will give the position where the occurrence appears in the starting rule, and therefore, in the original text. The offset for the primary occurrence at $X_i \to X_l X_r$ is $|\mathcal{F}(X_l)| - k + 1$ (l is obtained with an *access to rule* query for i). As previously mentioned, each time we arrive at the initial symbol X_s, the offset gives the position of a new occurrence in the original text.

To track a given occurrence inside of X_i, we first find all those $X_j \to X_i X_r$ for some X_r, using query *rules using a left symbol* for $\pi^{-1}(i)$. The offset is unaltered within those new nonterminals. Second, we find all those $X_j \to X_l X_i$ for some X_l, using query *rules using a right symbol* for i. The offset in these new nonterminals is within X_i plus $|\mathcal{F}(X_l)|$, where again $\pi^{-1}(l)$ is obtained from the result using an *access to rule* query, and then we apply π to get l. We proceed recursively with all the nonterminals X_j found, reporting the offsets (and finishing) each time we arrive at X_s.

Note that we are tracking each occurrence individually. Therefore we may process the same nonterminal X_i several times, yet with different offsets. Each occurrence may require to traverse the syntax tree all the way up to the root. We spend $O(\lg n/\lg\lg n)$ time at each step. Moreover, we carry out $m-1$ range queries for the different pattern partitions. Thus the overall time to find the *occ* occurrences is $O((m + h\,occ)\lg n/\lg\lg n)$.

We remark that we do not need to output all the occurrences of P. If we just want *occ* occurrences, our cost is proportional to this *occ*. Moreover, the *existence problem*, that is, determining whether or not P occurs in T, can be answered just by considering whether or not there are any primary occurrences.

Figure 4.2 (bottom) illustrates the search process. We describe next how to solve the remaining problem of finding the range of phrases starting/ending with a suffix/prefix of P.

4.4.3 Prefix and Suffix Searching

We present different time/space tradeoffs to search for P_l and P_r in the respective sets.

Binary search based approach. We can perform a binary search over the $\mathcal{F}(X_i)$s and over the $\mathcal{F}(X_i)^{rev}$s to determine the ranges where P_r and P_l^{rev}, respectively, belong. In order to do the string comparisons in the first binary search, we extract the (at most) m first terminals of $\mathcal{F}(X_i)$, in time $O((m+h)\lg n/\lg\lg n)$ (Section 4.4.1). As the binary search requires $O(\lg n)$ string

comparisons, the total cost is $O((m+h)\lg^2 n/\lg\lg n)$ for each partition P_lP_r. The search within the reverse phrases is similar, except that we extract the (at most) m rightmost terminals and must use π to find the rule from the position in the reverse ordering. This variant needs no extra space.

Compact Patricia Trees. Another option is to build Patricia trees [84] for the $\mathcal{F}(X_i)$s and for the $\mathcal{F}(X_i)^{rev}$s (adding a terminator for them so that each phrase corresponds to a leaf). Consider a binary digital tree where each root-to-leaf path spells out one string (where the character values are converted to binary). Then the Patricia tree is formed by collapsing unary paths of that tree into edges, and storing at each node the number of bits skipped from its parent, which we call *skips* or *skip values*. By using the Y bitmap, our symbols can be thought of as drawn from an alphabet of size $\sigma' \leq \min(\sigma, n)$. A search proceeds normally on the explicit bits in $O(m\lg\sigma')$ steps, and a final check against any leaf of the subtree found is used to verify the matching of the skipped bits in the search path (this takes $O(m)$ symbol comparisons).

Our Patricia trees have $O(n)$ nodes. We use a succinct tree representation requiring $O(n)$ bits that supports navigation in constant time (see Chapter 1). The ith leaf of the tree for the $\mathcal{F}(X_i)$s corresponds to nonterminal X_i (and the ith leaf of the tree for the $\mathcal{F}(X_i)^{rev}$s, to $X_{\pi(i)}$). Hence, upon reaching the tree node corresponding to the search string, we obtain the lexicographic range by counting the number of leaves up to the first and last descendants of the node, which can also be done in constant time.

The skips can be stored in an array indexed by preorder number (excluding leaves, as the skips are unnecessary for these), which can also be computed in constant time from the tree nodes. The problem is that in principle the skips require another $2n\lg u$ bits of space. If we do not store the skips at all, we can still compute them at each node by extracting the corresponding substrings for the leftmost and rightmost leaves of the node subtree, and checking in how many more bits they coincide [24]. This can be obtained in time $O((\lceil \ell/\lg\sigma'\rceil + h)\lg n/\lg\lg n)$ using the extract operation, where ℓ is the skip value obtained. The total search time is thus $O(m\lg n/\lg\lg n + mh\lg\sigma'\lg n/\lg\lg n) = O(mh\lg\sigma'\lg n/\lg\lg n)$, since the $O(\lceil \ell/\lg\sigma'\rceil \lg n/\lg\lg n)$ terms cannot add up to more than $O(m\lg n/\lg\lg n)$ as one cannot skip more than $m\lg\sigma'$ overall, and the term $O(h\lg n)$ can be paid for every bit in the pattern if all skips are 1, obtaining the second term $O((m\lg\sigma')(h\lg n/\lg\lg n))$.

Instead, we can use b bits for the skips, so that skips in the range $[1, 2^b-1]$ can be represented, and a skip value zero means $\geq 2^b$. Now we need to extract the leftmost and rightmost descendants only when the edge length is $\ell \geq 2^b$, and we will work $O((\lceil(\ell - 2^b)/\lg\sigma'\rceil + h)\lg n/\lg\lg n)$ time. Although the $\ell - 2^b$ terms can still add up to $O(m\lg\sigma')$ (e.g., if all the lengths are $\ell = 2^{b+1}$), the h terms can be paid only $O(1 + m\lg\sigma'/2^b)$ times. Hence the total search cost is $O((m+h+\frac{mh\lg\sigma'}{2^b})\lg n/\lg\lg n)$, at the price of at most $2nb$ extra bits of space. We must also account for the $m\lg\sigma'$ tree node traversals and for the final Patricia tree check due to skipped characters, but these checks add only $O((m+h)\lg n/\lg\lg n)$ time. For example, using $b = \lg h + \lg\lg\sigma'$ we get $O((m+h)\lg n/\lg\lg n)$ time and $2n(\lg h + \lg\lg\sigma') = 2n\lg h + o(n\lg n)$ extra bits of space.

As we carry out $m-1$ searches for prefixes and suffixes of P, as well as $m-1$ range searches,

plus *occ* extractions of occurrences, we have the following result. We call *fixed locating time* the time required to obtain the primary occurrences of the pattern.

Lemma 4.1. *Let $T[1,u]$ be a text over alphabet $[1,\sigma]$ represented by an SLP of n rules and height h. Then there exists a representation of T using $n(\lg u + 3\lg n + 2\lg h + o(\lg n)) + o(\sigma)$ bits, such that any substring $T[l,r]$ can be extracted in time $O((r-l+h)\lg n/\lg\lg n)$, and the positions of the occurrences of a pattern $P[1,m]$ in T can be located in a fixed time $O(m(m+h)\lg n/\lg\lg n)$ plus $O(h\lg n)$ time per occurrence reported. By removing the $2\lg h$ term in the space, the fixed locating time increases to $O(m(m+h)\lg^2 n/\lg lgn)$. The existence problem is solved within the fixed locating time.*

Compared with the $2n\lg n$ bits of the plain SLP representation, ours requires at least $4n\lg n + o(n\lg n)$ bits, that is, roughly twice the space. More generally, as long as $u = n^{O(1)}$, our representation uses $O(n\lg n)$ bits, of the same order as required by the SLP itself. Otherwise, our representation can be superlinear in the size of the SLP (almost quadratic in the extreme case $n = O(\lg u)$). Yet, if $n = o(u/\lg_\sigma u)$, our representation takes $o(u\lg\sigma)$ bits, asymptotically smaller than the *original* text. Any parsing of T into distinct phrases, for example with a LZ78 grammar [119], achieves at most $u/\lg_\sigma u$ phrases, even for incompressible texts T, thus $n = o(u/\lg_\sigma u)$ is roughly equivalent to saying that T is asymptotically grammar-compressible. Also, since the LZ78 parsing takes $O(u)$ time, we can ensure that within this optimal time one can find an SLP that at least guarantees $O(u\lg\sigma)$ size for our self-index.

Combining both methods. We can combine the two previous approaches as follows. We sample one string out of k lexicographically consecutive ones, for a parameter k. We build the Patricia tree for the sampled set of strings. After finding the range of sampled strings that are prefixed with a pattern, we must conclude with a binary search on the unsampled range preceding the first sampled result, and another on the range following the last sampled result. (If the Patricia range is empty we must binary search the range preceding the first leaf larger than the search pattern; this leaf is easily found by reentering the tree after the final check determines the point where the pattern and the followed path differ.) We store the Patricia tree skips using b bits of precision. The total cost is the $O((m+h)\lg n/\lg\lg n)$ time of the Patricia tree search, plus the $O((m+h)\lg k\lg n/\lg\lg n)$ time required by the binary searches. In exchange, the extra space is $\frac{2n\lg h}{k} + o(n\lg n)$ bits. This leads to our main theorem on indexing SLPs.

Theorem 4.5. *Let $T[1,u]$ be a text over alphabet $[1,\sigma]$ represented by an SLP of n rules and height h. Then there exists a representation of T using $n(\lg u + 3\lg n + \frac{2}{k}\lg h + o(\lg n)) + o(\sigma)$ bits, for any parameter $1 \le k \le \lg h$, such that any substring $T[l,r]$ can be extracted in time $O((r-l+h)\lg n/\lg\lg n)$, and the positions of the occurrences of a pattern $P[1,m]$ in T can be located in a fixed time $O(m(m+h)\lg(k+1)\lg n/\lg\lg n)$ plus $O(h\lg n/\lg\lg n)$ time per occurrence reported. The existence problem is solved within the fixed locating time.*

By setting $k = 1$ and $k = \alpha(h)$ (the inverse Ackermann function [1]) we obtain the two most relevant space/time tradeoffs.

Corollary 4.1. *In Theorem 4.5 we can achieve fixed locating time $O(m(m+h)\lg n/\lg\lg n)$ and $n(\lg u + O(\lg n)) + o(\sigma)$ bits of space. We can also achieve $O(m(m+h)\lg n \lg\alpha(h)/\lg\lg n)$ fixed locating time and $n(\lg u + 3\lg n + o(\lg n)) + o(\sigma)$ bits of space.*

4.4.4 Construction

We discuss now how to carry out the construction of our index given the SLP.

Let us start with the binary relation that represents the grammar. Assume we have already computed the proper direct and reverse lexicographical orderings for X_r and X_l, respectively. We reorder once again the rules $X_i \to X_l X_r$ by their X_r component. Now we create one list per X_l, and traverse the rules $X_i \to X_l X_r$ in X_r order, adding pair (X_r, X_i) to the end of list X_l. Then we traverse the lists in X_l order, adding the X_r components to S_B and the X_is to $S_{\mathcal{L}}$. All this takes $O(n\lg n)$ time, dominated by the ordering of rules. Bit vectors X_A and X_B are easily built in $O(n)$ time, including their *rank*/*select* structures. The permutation π, including its extra structures for computing π^{-1}, is built in $O(n)$ time once the lexicographical orderings of the rules is found.

Building the wavelet trees for S_B and $S_{\mathcal{L}}$ takes $O(n\lg n)$ additional time within the space bounds of our construction (see Chapter 2).

The lengths $|\mathcal{F}(X_i)|$ are easily obtained in $O(n)$ time, by performing a bottom-up traversal of the DAG of the grammar (going first top-down, marking the already traversed nodes to avoid re-traversing, and assigning the lengths in the return of the recursion).

The remaining cost is that of lexicographically sorting the strings $\mathcal{F}(X_r)$ and $\mathcal{F}(X_l)^{rev}$, or alternatively, building the tries. In principle this can take as much as $\sum_{i=1}^{n} |\mathcal{F}(X_i)|$, which can be even $\omega(u)$. Let us focus on sorting the direct phrases $\mathcal{F}(X_i)$, as the reversed ones can be handled identically.

Our solution is based on the fact that all the phrases are substrings of $T[1,u]$. We first build the *suffix array* [83] of T in $O(u)$ time [69]. This is an array $A[1,u]$ pointing to all the suffixes of $T[1,u]$ in lexicographic ordering, $T[A[i],u] < T[A[i+1],u]$. As it is a permutation, we can also build its inverse $A^{-1}[1,u]$ in $O(u)$ time. Next, we build the LCP array in $O(u)$ time [72]: $LCP[k]$ is the length of the longest common prefix between $T[A[k-1],u]$ and $T[A[k],u]$. On top of this array, we build in $O(u)$ time an $O(u)$-bits data structure that answers range minimum queries in constant time [49]. With this data structure we compute $\textsc{rmq}(i,j) = \min_{i<k\le j} LCP[k]$, which is the longest common prefix between $T[A[i],n]$ and $T[A[j],n]$, in constant time.

To sort the phrases, we start by simulating the expansion of T using the grammar and recording one starting text position p_i for each string $\mathcal{F}(X_i)$. Now, comparing $A^{-1}[p_i]$ with $A^{-1}[p_j]$ would give us an ordering between p_i and p_j if none of them were a prefix of the other. Instead, if one is a prefix of the other, the prefix must be regarded as smaller than the other string. We know that $\mathcal{F}(X_i)$ is a prefix of $\mathcal{F}(X_j)$ if $|\mathcal{F}(X_i)| \le \textsc{rmq}(A^{-1}[p_i], A^{-1}[p_j])$, and vice versa. Then the phrases can be sorted in $O(n\lg n)$ time.

To build the Patricia trees, instead, we build the *suffix tree* of T in $O(u)$ time [41]. This can be seen as a Patricia tree built on all the suffixes of T. We find each of the n suffix tree leaves corresponding to phrase beginnings (that is, the $A^{-1}[p_i]$-th leaves), and create new leaves at depth

$|\mathcal{F}(X_i)|$ which are ancestors of the original suffix tree leaves. The points to insert these n new leaves are found by binary searching the string depths $|\mathcal{F}(X_i)|$ with level ancestor queries [14] from the original suffix tree leaves. These binary searches take $O(n \lg u)$ time in the worst case. Finally, the desired Patricia tree is formed by collecting the ancestor nodes of the new leaves while collapsing unary paths again, which yields $O(n)$ nodes. The Patricia tree is converted to binary by translating skip values and replacing each node having s children by a small Patricia tree where we insert the s strings of $\lg \sigma'$ bits corresponding to the s characters. This adds $O(n \lg \sigma')$ time overall.

The whole process takes $O(u + n \lg u)$ time, and $O(u \lg u)$ bits of working space.

We can reduce the construction space by using compressed suffix arrays and trees, which slightly increases construction time. Instead of a classical suffix array, we build a Compressed Suffix Array (CSA) [105], within $O(u \lg \lg \sigma)$ time and $O(u \lg \sigma)$ bits of space [65]. Similarly, we build a Compressed Suffix Tree (CST) [106] within time $O(u \lg^{\epsilon} u)$ and the same $O(u \lg \sigma)$ bits [65]. Among other operations the CST can, in constant time, determine if a node is an ancestor of another node, count the number of nodes and leaves below any node, move to the parent, first child and next sibling of a node, to the ancestor of any depth ("level ancestor") of a node, and find the lowest common ancestor between any two nodes. In addition, we can obtain the preorder number of any node, the node of any preorder number, the left-to-right rank of any leaf, and the ith left-to-right leaf.

The CSA supports the query $A^{-1}[p]$ within time $O(\lg^{\epsilon} u)$, for any constant $0 < \epsilon < 1$. Hence, except for the prefix problem, ordering the strings takes time $O(n(\lg n + \lg^{\epsilon} u))$, by first storing the $A^{-1}[p_i]$ values for the rules and then sorting them. To find out if $\mathcal{F}(X_i)$ is a prefix of $\mathcal{F}(X_j)$ we see if the suffix tree node corresponding to $\mathcal{F}(X_i)$ is an ancestor of that of $\mathcal{F}(X_j)$. Thus we first compute and store the nodes for all the phrases $\mathcal{F}(X_i)$ and then complete the sorting. To compute the node for any $\mathcal{F}(X_i)$ we first find the $A^{-1}[p_i]$th tree leaf in constant time. Now we compute its ancestor representing a string of length $|\mathcal{F}(X_i)|$. Although level ancestor queries are also supported in constant time, knowing the length of the string corresponding to a suffix tree node takes $O(\lg^{\epsilon} u)$ time. Thus we binary search the correct ancestor in $O(\lg^{1+\epsilon} u)$ time. Overall, the sorting takes $O(n \lg^{1+\epsilon} u)$ time.

For constructing the Patricia trees we also use the CST. We set up a bitmap $M[1, O(u)]$, so that $M[i]$ will be a mark for the node with preorder number i. As we can map from preorder numbers to nodes and back in constant time, we will refer to nodes and their preorder numbers indistinctly. We mark in M the suffix tree nodes corresponding to the phrases $\mathcal{F}(X_i)$ found as explained in the previous paragraph. If these end on an edge, we mark their child node in M. Now we traverse the marked nodes from left to right in M (in overall time $O(u)$), and for each consecutive pair of marked nodes, we also mark its lowest common ancestor in M. This process marks all the nodes that should belong to our Patricia tree. Finally, we traverse the marked nodes of M left to right again, which is equivalent to traversing the marked tree nodes in preorder, and create the Patricia tree with those nodes. Note that a single marked node may correspond to several strings whose insertion point is at the edge leading to the marked node. Since a preorder traversal of marked nodes corresponds to a left-to-right traversal of the suffix tree leaves $A^{-1}[p_i]$, all the strings to consider are next in the left-to-right traversal, so it is not

hard to delimit them, sort them by $|\mathcal{F}(\cdot)|$, and create the successive Patricia tree nodes, all within the current time bounds.

The overall cost is dominated by $O(u \lg^{\epsilon} u + n \lg^{1+\epsilon} u)$ time and $O(u \lg \sigma + n \lg u)$ bits of space. Since $u + n \lg u = O(u + n \lg n)$, we have the result.

Theorem 4.6. *Let $T[1,u]$ be a text over alphabet $[1,\sigma]$ represented by an SLP of n rules. Our representation can be built in $O(u + n \lg n)$ time and $O(u \lg u)$ bits of space. Alternatively, it can be built in $O((u + n \lg n) \lg^{\epsilon} u)$ time and $O(u \lg \sigma + n \lg n)$ bits of space.*

We remind the reader that $n \lg u = O(u \lg \sigma)$ for many simple grammar-based compression methods, for example LZ78 [119].

4.4.5 Faster Locating and Counting

We locate occurrences individually, even if they share the same phrase (albeit with different offsets). We show now that, if one uses some extra space for the query process, the $O(h\,occ \lg n / \lg \lg n)$ time needed for the occ occurrences can be transformed into $O(\min(h\,occ, n) \lg n + occ)$, thus reducing the time when there are many occurrences to report.

We set up a min-priority queue H, where we insert the phrases X_i which contain primary occurrences. We do not yet propagate these phrases to the secondary occurrences. The priority of X_i will be $|\mathcal{F}(X_i)|$. For each such X_i, with $X_i \to X_l X_r$, we store l and r; the minimum and maximum offset of the primary occurrences found in X_i; left and right *pointers*, initially null and later pointing to the data for X_l and X_r, if they are eventually inserted in H; and left and right offsets associated to those pointers. The data of those X_i will be kept in a fixed memory position across all the process, so we can set pointers to them, which will be valid even after we remove them from H (H just contains pointers to those memory areas). The left and right pointers point to those areas as well. Separately, we store a balanced binary search tree that, given i, gives the memory position of X_i, if it exists (this tree permits, in particular, freeing all the memory areas at the end).

Now, we repeatedly extract an element X_i with smallest $|\mathcal{F}(X_i)|$ from H, and using our binary relation data structures find all the other X_js that mention X_i in their rule. We use the balanced tree to determine whether X_j is already in H (and where is its memory area) or not (X_j could already be in H, e.g., if it has its own primary occurrences). If it is not, we allocate memory for X_j and insert it into H. Now, if $X_j \to X_i X_r$, then we set the left pointer of X_j (1) to the left pointer of X_i if X_i does not have primary occurrences nor a right pointer, setting the left offset of X_j to that of X_i; (2) to the right pointer of X_i if X_i does not have primary occurrences nor a left pointer, setting the left offset of X_j to the right offset of X_i; (3) to X_i itself otherwise, setting the left offset of X_i to zero. If $X_j \to X_l X_i$, we assign the right pointer and offset of X_j in the same way, except that we add $|\mathcal{F}(X_l)|$ to the right offset of X_j. Note that the priority queue ordering implies that all the occurrences descending from X_j are already processed when we process X_j itself.

The process finishes when we extract the initial symbol from H and H becomes empty. At this point we are ready to report all of the occurrences with a recursive procedure starting at the initial symbol. Moreover, we can report them in text order: To report $X_i \to X_l X_r$, we first report the occurrences at the left pointer of X_i (if not null), shifting their values by the left offset of X_i; then the primary occurrences of X_i (if any); and then the occurrences at the right pointer of X_i (if not null), shifting their values by the right offset of X_i. Those shifts accumulate as recursion goes down the tree, and become the true occurrence positions at the end.

To display all the primary occurrences of a node knowing only the *first* and *last* positions, we notice that these occurrences must overlap, thus we know the full text content of the area where the primary occurrences other than the first and the last may appear. By preprocessing the pattern we can obtain those occurrences in constant time each: Let $last - first = d$, so $last - d$ is the *first* primary occurrence. This means that $P[d+1, m] = P[1, m-d]$, thus P occurs at positions 1 (*first*) and $d+1$ (*last*) of string $X = P[1,d] \cdot P = P \cdot P[m-d+1, m]$. We wish to know which is the occurrence of P in X that precedes that at position $d+1$. We can search for P in $X[1, m+d-1]$ in time $O(m)$ using algorithm KMP [75] and store the position $d' < d$ of the last occurrence in a table $O[d]$. For the second previous occurrence, we have already that P occurs at position $d'+1$ of string $X' = P[1,d'] \cdot P$, thus it corresponds to $O[d']$.

Therefore, it is enough to precompute all those $O[1,m]$ values in $O(m^2)$ time beforehand. Later, given *first* and *last*, we report each primary occurrence in constant time by doing $d \leftarrow last - first$, reporting $last - d$, then $d \leftarrow O[d]$, reporting $last - d$, and so on until $d = 0$, where we report *last*.

Let us now analyze the algorithm. Although each occurrence can trigger h insertions into H, nodes are not repeated in H, and thus there are at most $O(\min(h\,occ, n))$ elements in H. Thus the space is in the worst case $O(\min(h\,occ, n) \lg u + m \lg m)$ bits (the second part is for O). As for the time, we spend $O(\lg n)$ time to insert each primary occurrence into H and compute its associated data, $O(\lg n)$ time to extract it from H, and $O(\lg n)$ time to find each of its parents and insert them into H (each parent $X_i \to X_l X_r$ is processed at most twice, from X_l and from X_r). Thus the overall cost of filling and emptying H is $O(\min(h\,occ, n) \lg n)$.

As for the process of reporting once H is emptied, note that the left and right pointers can be traversed in constant time and, because in the tree induced by the left/right pointers each pointed node either has at least one distinct primary occurrence, or it has two children, it follows that the total traversal time is $O(occ)$. Reporting all the primary occurrences can also be done in time $O(occ)$.

Overall, the time is $O(m^2 + \min(h\,occ, n) \lg n + occ)$, provided we can afford the extra space at search time. Note that the $O(m^2)$ part (to build $O[1,m]$) is dominated by higher terms in the search complexity.

Theorem 4.7. *Under the same conditions as those in Theorem 4.5, we can locate the occ occurrences of $P[1,m]$ within the fixed locating time reported in that theorem plus $O(\min(h\,occ, n) \lg n + occ)$, by using $O(\min(h\,occ, n) \lg u + m \lg m)$ extra bits of space at search time.*

A simplification of this technique lets us count the number of occurrences of $P[1,m]$ more

efficiently than by locating them all. We follow the same process of detecting the primary occurrences and using a heap to process the nonterminals by increasing length. We store, for each nonterminal, the number of occurrences of P inside it (initially zero). Each primary occurrence adds 1 to the counter of the corresponding nonterminal. Each nonterminal we extract from the heap adds its counter value to that of all the nonterminals that use it. When we finally extract the initial symbol, its counter is the number of occurrences of P. It is easy to see that the overall additional counting time is $O(\min(h\,occ, n)\lg n)$, which is interesting when $occ = \Omega(n/h)$ (otherwise it is better to locate the occurrences one by one).

Theorem 4.8. *Under the same conditions of Theorem 4.5, we can count the occurrences of $P[1,m]$ within the fixed locating time reported in those theorems plus $O(n \lg n)$, by using $O(n \lg u)$ extra bits of space at search time.*

Incidentally, this result provides an improved solution to a recently proposed problem on SLPs [66].

Corollary 4.2. *Given a text $T[1,u]$ over an alphabet of size σ, and an SLP of n rules generating T, the problem of finding the most repeated substring of T of length at least two can be solved using $O(n \lg u)$ bits of space and $O(\sigma^2 n \lg n / \lg\lg n)$ time.*

Proof. Clearly the most repeated substring if of length 2 exactly, as longer ones cannot be more frequent. We first build our self-index from the SLP and then count the occurrences of all the σ^2 possible pairs of characters. In principle the construction takes $O(u + n \lg n)$ time. However, the $O(u)$ term comes from sorting the strings $\mathcal{F}(X_r)$ and $\mathcal{F}(X_l)^{rev}$. For the purpose of this problem, these can be just sorted by their first/last two symbols. The first/last two symbols of all the phrases are easily obtained in $O(n)$ time with a recursive traversal of the grammar (marking traversed nodes to avoid re-traversing them). Then the phrases can be sorted in $O(n \lg n)$ time. Once the self-index is built, we apply Theorem 4.8 for each possible pair of symbols, using our fastest SLP representation of Theorem 4.5 and $m = 2$ to obtain the fixed locating time $O(m(m + h)\lg n/\lg\lg n) = O(n \lg n/\lg\lg n)$. □

The best previous result [66] needs $O(\sigma^2 n^2)$ time and $O(n^2)$ words of space, thus we significantly improve it in both aspects.

4.5 More General Grammar Compressors

Until now we have considered the case where the grammar-based compressor generates a single non-terminal symbol that represents the text. Many grammar-based compressors [119, 78, 96] output instead a *set of rules* (which can be seen as a forest of parse trees) and a *sequence* of terminals and nonterminals, whose expansion, using the rules, leads to the original text. This is captured by the following definition.

Definition 4.3 (Relaxed Straight-Line Program (RSLP))**.** *A* Relaxed Straight-Line Program (RSLP) $\mathcal{G} = (X = \{X_1, \ldots, X_n\}, \Sigma, \mathcal{C} = C_1 C_2 \ldots C_c)$ *is a tuple where* (X, Σ) *is a grammar on an alphabet* $\Sigma = [1, \sigma]$ *of terminals, such that each* X_i *generates a single finite string* $\mathcal{F}(X_i)$*, and can be of two types, as follows:*

- $X_i \to \alpha$*, where* $\alpha \in \Sigma$*. It generates string* $\mathcal{F}(X_i) = \alpha$*.*
- $X_i \to X_l X_r$*, where* $l, r < i$*. It generates string* $\mathcal{F}(X_i) = \mathcal{F}(X_l)\mathcal{F}(X_r)$*.*

Moreover, $\mathcal{C}$ *is a sequence of terminals and nonterminals* $C_i \in X \cup \Sigma$*, and* $\mathcal{G}$ *represents the text* $T[1, u] = \mathcal{F}(C_1)\mathcal{F}(C_2)\ldots\mathcal{F}(C_c)$*, assuming* $\mathcal{F}(\alpha) = \alpha$ *for* $\alpha \in \Sigma$*.*

This is basically an SLP relaxed to allow the first rule to produce more than just two nonterminals.

It is clear that an RSLP can be converted into an SLP by adding $c - 1$ new rules that derive $\mathcal{C}$ from an initial symbol I, and then the symbols of $\mathcal{C}$ expand as usual. The new rules can be balanced, thus adding only $\lg c$ to the height of the grammar. We might also need to introduce nonterminals for any terminal that could be mentioned in $\mathcal{C}$.

Definition 4.4. *Given an RSLP* $\mathcal{G} = (X = \{X_1, \ldots, X_n\}, \Sigma, \mathcal{C})$*, let* $n' \leq n + \min(c, \sigma)$ *be the number of rules in* X *once we add those of the form* $X_i \to \alpha$ *for the terminals* α *mentioned in* $\mathcal{C}$ *and not in* X*.*

Definition 4.5. *The height of RSLP* $\mathcal{G} = (X = \{X_1, \ldots, X_n\}, \Sigma, \mathcal{C})$ *is* $\texttt{height}(\mathcal{G}) = \max\{\texttt{height}(X_i), 1 \leq i \leq n\}$*. We will refer to* $\texttt{height}(\mathcal{G})$ *as* h *when the referred grammar is clear from the context.*

Corollary 4.3. *Let* $T[1, u]$ *be a text over alphabet* $[1, \sigma]$ *represented by an RSLP of* n *rules and height* h*, and a sequence of* c *nonterminal symbols. Then* T *can be represented using Theorem 4.5, using an SLP of* $n' + c - 1$ *rules and height* $h + \lceil \lg c \rceil + 1$*. For example, it can be represented using* $(n' + c)(\lg u + 3\lg(n' + c)) + o(\sigma)$ *bits, such that any substring* $T[l, r]$ *can be extracted in time* $O((r - l + h + \lg c)\lg(n' + c)/\lg\lg(n' + c))$*, and the positions of the occurrences of a pattern* $P[1, m]$ *in* T *can be located in a fixed time* $O(m(m + h + \lg c)\lg(n' + c)\lg\alpha(h + \lg c)/\lg\lg(n' + c))$ *plus* $O((h + \lg c)\lg(n' + c)/\lg\lg(n' + c))$ *time per occurrence reported.*

We now propose a more sophisticated scheme that can achieve better results.

1. We use our binary relation data structure to represent the forest of rules X. Thus it will require $n'(\lg u + 3\lg n' + o(\lg n')) + o(\sigma)$ bits of space, according to Section 4.4.
2. The sequence $\mathcal{C}$ is represented with the structure by Golysnki et al. This will require $c(\lg n' + o(\lg n'))$ bits of space and carry out *access* in $O(\lg\lg n')$ time and *select* in $O(1)$ time.
3. We store a bitmap $B[1, u]$ marking the positions of T where the symbols of $\mathcal{C}$ begin. It can be represented such that it uses $c \lg \frac{u}{c} + O(c + \lg\lg u)$ bits as we will only need constant-time $select_1$ queries on it (See ID in Chapter 1).

4. We store another labeled binary relation of c rows and n' columns. Value $1 \leq i \leq c$ is related to $1 \leq j \leq n'$ with label $2 \leq k \leq c$ if the suffix of T that starts at the k-th symbol of $\mathcal{C}$ is at lexicographical position i among all such suffixes, and the lexicographic position of $\mathcal{F}(\mathcal{C}_{k-1})^{rev}$, among all the distinct reversed nonterminals (and terminals) $\mathcal{F}(X_i)^{rev}$, is j. We wish to carry out range searches on this binary relation. Yet, as there is exactly one point per row, we do not need the bitmap X_B. We choose the representation of Golynski et al. [54] that achieves constant-time *access* for $S_{\mathcal{L}}$. This binary relation takes $(c + o(c))(\lg n' + \lg c + o(\lg c) + O(1)) + o(n') = c(\lg n' + \lg c + o(\lg(n' + c))) + o(n')$ bits of space, according to Theorem 4.3.

The total space is $(c + n')\lg u + (2c + 3n')\lg n' + o((c + n')\lg(c + n'))$. This can be up to 1/4 the space of Corollary 4.3 if $c \gg n'$, and never asymptotically larger.

The search for P proceeds just like for SLPs, within time $O((m(m + h)\lg\alpha(h)\lg n'/\lg\lg n' + h\,occ\lg n'/\lg\lg n')$ if using the most compact variant offered by Corollary 4.1, which would not change the asymptotic space formula given above. However, this will only find occurrences inside dictionary symbols. To complete the search, for each occurrence with offset o within symbol X_i, we look for all the positions $p_j = select_{\mathcal{C}}(X_i, j)$, for $j = 1, 2, \ldots$, and report the text position $select_1(B, p_j) + o$, within overall time $O(occ)$. This includes the cases where X_i does not occur in $\mathcal{C}$, as in this case the occurrence will still appear in T and thus we can charge the search cost to it.

It remains to find the occurrences that overlap two or more entries in $\mathcal{C}$. To find each of them just once, we will find the partitions $P_l P_r$ such that P_l is the suffix of a single entry in $\mathcal{C}$ and P_r is the prefix of a concatenation of entries in $\mathcal{C}$. Our second binary relation will let us find the positions $\mathcal{C}_{k-1}\mathcal{C}[k\ldots]$ where P_l appears at the end of C_{k-1} and P_r at the beginning of $\mathcal{C}[k\ldots]$. We already know the lexicographical range of each P_l^{rev} within the $\mathcal{F}(X_i)^{rev}$s. We can now binary search each corresponding P_r within the c suffixes starting at phrase beginnings. The content of the t-th lexicographical suffix is obtained by accessing $\mathcal{C}[S_{\mathcal{L}}[t]\ldots]$ and expanding each symbol of $\mathcal{C}$ using the binary relation that represents the rules. This gives $O((m+h)\lg n'/\lg\lg n')$ time per access (note the h overhead applies only to the last, partially expanded, symbol, as the rest are fully expanded). The binary search can be sped up with a partial Patricia tree, just as done in Theorem 4.5, as bitmap B lets us know exactly which offset from which symbol of $\mathcal{C}$ to extract. So the overall time is $O(m(m+h)\lg n'\lg\alpha(h)/\lg\lg n')$ and the extra space for the Patricia trees is $o(c\lg h) = o(c\lg n')$. Now, given the m lexicographical ranges of the suffixes, we carry out the m range searches in the second binary relation in $O(m\lg n'/\lg\lg n')$ time, and extract each occurrence in $O(\lg n'/\lg\lg n')$ time. We must map each position in $\mathcal{C}$ to the corresponding position in T via bit vector B, and subtract $|P_l|$ to yield the final offset.

Overall, the search time is $O(m(m + h)\lg\alpha(h)\lg n'/\lg\lg n' + h\,occ\lg n'/\lg\lg n')$. This can be up to $O(\lg c)$ times faster than Corollary 4.3 (if $c \gg n'$), and never worse.

Finally, to extract $T[l, r]$, we first binary search, using $select_1$ on B, the symbols of $\mathcal{C}$ to extract, and then expand them one by one using the grammar, in overall time $O((r-l+h)\lg n'/\lg\lg n + \lg c)$.

Theorem 4.9. *Let $T[1, u]$ be a text over alphabet $[1, \sigma]$ represented by an RSLP of n rules, height h, and a sequence of c nonterminal symbols. Let n' the number of rules after expanding them to*

contain the explicit terminals in the sequence. Then T can be represented using $(c+n')\lg u + (2c+3n')\lg n' + o((c+n')\lg(c+n'))$ bits of space. Any substring $T[l,r]$ can be extracted in time $O((r-l+h)\lg n'/\lg\lg n' + \lg c)$, and the positions of the occurrences of a pattern $P[1,m]$ in T can be located in a fixed time $O(m(m+h)\lg\alpha(h)\lg n'/\lg\lg n')$ plus $O(h\lg n'/\lg\lg n')$ time per occurrence reported.

4.5.1 Applications

One example where Theorem 4.9 applies straightforwardly is for the Re-Pair compression algorithm [78]. Re-Pair is a grammar-based compression method based on repeatedly replacing the most frequent pair of (terminal or nonterminal) symbols in the text by a new nonterminal, until the most frequent pair appears once. The result of Re-Pair compression is a set of n rules (essentially in SLP form) plus a sequence of c terminal or nonterminal symbols. It runs in $O(u)$ time and $O(u\lg u)$ bits of space over a text $T[1,u]$ [78]. It is also possible to select the rules in a balanced fashion [108] so as to guarantee that $h = O(\lg n)$, thus we achieve a practical index for this particular algorithm.

A practical implementation of the Re-Pair-based self-index was compared against state-of-the-art indexes for highly repetitive DNA sequences [27]. In particular, when the mutation rate from one sequence to another in the collection is near 0.01%, the Re-Pair self-index improves upon the RLCSA [109], the best alternative self-index obtained so far for this setting. Such mutation rates are realistic when the database contains genomes of different individuals of the same species [40].

A less straightforward application is the LZ78 compression algorithm, where we obtain a self-index that is competitive with previous work.

Consider the LZ78-parsing [119] of a string T of length u, drawn over an alphabet Σ of size σ. The text is processed left-to-right and, at each step, a new *phrase* is produced from the longest possible prefix of the remaining text which is formed by a previous phrase plus a character. The process produces n phrases X_i, corresponding to a grammar of the form $X_i \to X_j\alpha$, where $j < i$ and $\alpha \in \Sigma$. The text is obtained by expanding the sequence $\mathcal{C} = X_1X_2\ldots X_n$.

Much research has been carried out to obtain self-indexes for this compression method [89, 5, 102], usually called the *LZ-Index*. The first proposal [89] achieves $4n\lg n + 2n\lg\sigma + o(n\lg n)$ bits of space, and is able of locating the *occ* occurrences of $P[1,m]$ in time $O(m^3\lg\sigma + (m+occ)\lg n)$. In order to report true text positions of occurrences (and not just phrase positions), $n\lg\frac{u}{n} + O(n) + o(u)$ additional bits are necessary, for a total of $n\lg u + 3n\lg n + 2n\lg\sigma + o(u + n\lg n)$ bits of space.

A verbatim application of Theorem 4.9 could almost double this space. We show now that, by a slight adaptation of our general technique to the specificities of the LZ78 grammar, we can achieve a result that is competitive with the previous proposals.

Let us first consider the first binary relation of Theorem 4.9. Because the right-hand side of rules is always a character, our binary relation actually has σ columns. The rules can always be ordered by $\mathcal{F}(X_l)^{rev}$ (that is, their row order), and permutation π serves as a tool to know their original identifier (which coincides with their only occurrence position in $\mathcal{C}$); π^{-1} is needed only

for extracting $T[l,r]$. We do not need to store the lengths $|\mathcal{F}(X_l)|$, as we always descend to the left rule knowing that the length of the child is one less than its parent. Finally, $n' = n$ since $\mathcal{C}$ mentions only nonterminals. This makes the space $n(2\lg n + \lg\sigma + o(\lg n))$ bits (σ is assumed to be $o(n)$ in LZ78 self-indexes, so we do the same for comparison). The $n\lg\sigma$ bits are for S_B and the $2n\lg n$ for π and $S_{\mathcal{L}}$. The operations on S_B run in $O(\lg\sigma/\lg\lg n)$ time and those on $S_{\mathcal{L}}$ run in $O(1)$ time for *select* and $O(\lg\lg n)$ time for *access*.

Sequence $\mathcal{C} = X_1X_2\ldots X_n$ does not need to be stored. The bitmap B is stored as in the LZ78 proposal, requiring $n\lg\frac{u}{n} + O(n) + o(u)$ bits and doing the mapping in constant time [101].

For the second binary relation of Theorem 4.9, we do not require $S_{\mathcal{L}}$, as we know that the element at column j is $X_{\pi(j)}$ and thus the label is $\pi(j)+1$. Therefore the structure requires $n(\lg n + o(\lg n))$ bits. Furthermore, we do not need X_A because there is only one point per column in the binary relation.

Thus the total space is $n\lg u + 2n\lg n + n\lg\sigma + o(u + n\lg n)$ bits, which is less by an $n\lg n + n\lg\sigma$ term than the original LZ78 proposal [89].

The search for P starts by locating the occurrences within the nonterminals. The only possible partition of $P[1,m]$ is $P = P[1,m-1]P[m]$. The second part is easily searched for in constant time, whereas the first part requires $O(m\lg\sigma\lg\alpha(h))$ time because we search the reversed rules, which are unrolled in right-to-left order and thus the height we need to descend is bounded by m. The $O(\lg\sigma/\lg\lg n)$ cost is for accessing the rules and the $O(\lg\alpha(h))$ cost corresponds to the binary search speeded up with the partial Patricia trees. Once the searches are finished, each occurrence within rules is extracted in $O(\lg\sigma/\lg\lg n + \lg\lg n)$ time. These are mapped to $\mathcal{C}$ using π and B in additional constant time.

To spot the occurrences that span more than one phrase, we split $P = P_lP_r$ in all the $m-1$ possible ways and search for P_l in the columns in time $O(m\lg\sigma\lg\alpha(h)/\lg\lg n)$ (using the first binary relation as done for $P[1,m-1]$) and for P_r in the rows of the second binary relation in time $O((m\lg n + (m+h)\lg\sigma)\lg\alpha(h)/\lg\lg n)$. The latter time is because for the extraction of the first m symbols of a row it is necessary to (1) find in time $O(\lg n/\lg\lg n)$ the only column j related to the row; (2) use $\pi^{-1}(\pi(j)+s)$ to find the row in the first binary relation corresponding to the s-th phrase to extract, in $O(\lg n/\lg\lg n)$ time per phrase; (3) once that row is located, spending $O(\lg\sigma/\lg\lg n)$ time to find each symbol of each phrase in the first binary relation. Finally, we use a partial Patricia tree to speed up the binary search. Overall, as $\sigma = o(n)$, the m searches add up to $O(m(m\lg n + h\lg\sigma)\lg\alpha(h)/\lg\lg n)$ time, plus a negligible $O(m\lg n/\lg\lg n)$ time for the range searches. Each occurrence within the ranges is found in time $O(\lg n/\lg\lg n)$.

Overall, the search time is $O(m(m\lg n + h\lg\sigma)\lg\alpha(h)/\lg\lg n + occ\lg n/\lg\lg n)$. This is not comparable with the original work [89], but under the usual assumption for random texts $h = O(\lg_\sigma n)$, the time is usually dominated by $O(m^2\lg n\lg\alpha(h)/\lg\lg n + occ\lg n/\lg\lg n)$. This compares favorably with $O(m^3\sigma + occ\lg n)$ of the original work for long enough m. Although more recent developments [5] essentially achieve $(2+\epsilon)n\lg n$ bits of space and $O(m^2 + (m+occ)\lg n)$ time.

4.6 Indexing General Grammars

We can further extend our approach to more general grammars that generate a single text. In this scenario, we avoid paying time proportional to the height for the find operation, improving the bounds of the previous results presented in this chapter.

4.6.1 Preprocessing and Representing the Grammar

We will work with a grammar $\mathcal{G}$ that generates a single string $T[1..u]$, formed by n (terminal and nonterminal) symbols. The $\sigma \leq n$ terminal symbols come from an alphabet $\Sigma = [1, \sigma]$, and then $\mathcal{G}$ contains $n - \sigma$ rules of the form $X_i \to \alpha_i$, one per nonterminal. The main difference is that α_i, called the *right-hand side* of the rule, corresponds now to the sequence of terminal and nonterminal symbols generated by X_i (without recursively unrolling rules). We call $N = \sum |\alpha_i|$ the *size* of $\mathcal{G}$. Note it holds $\sigma \leq N$, as the terminals must appear in the right-hand sides. We assume all the nonterminals are used to generate the string; otherwise unused rules can be found and dropped in $O(N)$ time.

We preprocess $\mathcal{G}$ as follows. First, for each terminal symbol $a \in \Sigma$ present in $\mathcal{G}$ we create a rule $X_a \to a$, and replace all other occurrences of a in the grammar by X_a. As a result, the grammar contains exactly n nonterminal symbols $\mathcal{X} = \{X_1, \ldots, X_n\}$, each associated with a rule $X_i \to \alpha_i$, where $\alpha_i \in \Sigma$ or α_i is a sequence of elements in X. We assume that X_n is the start symbol.

Any rule that generates just one single nonterminal $X_i \to \alpha_i$ where $|\alpha_i| \leq 1$ is removed by replacing X_i by α_i everywhere, decreasing n and without increasing N.

We further preprocess $\mathcal{G}$ to enforce the property that any nonterminal X_i, except X_n and those $X_i \to a \in \Sigma$, must be mentioned in at least two right-hand sides. We traverse the rules of the grammar, count the occurrences of each symbol, and then rewrite the rules, so that only the rules of those X_i appearing more than once (or the excepted symbols) are rewritten, and as we rewrite a right-hand side, we replace any (non-excepted) X_i that appears once by its right-hand side α_i. This transformation takes $O(N)$ time and does not alter N (yet it may reduce n). Algorithm 4 shows the high-level pseudo-code for this process.

Note n is now the number of rules in the transformed grammar $\mathcal{G}$. We will still call N the size of the original grammar (the transformed one has size $\leq N + \sigma$; similarly its number of rules is at most $n + \sigma$).

We call $\mathcal{F}(X_i)$ the single string generated by X_i, that is $\mathcal{F}(X_i) = a$ if $X_i \to a$ and $\mathcal{F}(X_i) = \mathcal{F}(X_{i_1}) \ldots \mathcal{F}(X_{i_k})$ if $X_i \to X_{i_1} \ldots X_{i_k}$. $\mathcal{G}$ generates the text $T = \mathcal{L}(\mathcal{G}) = \mathcal{F}(X_n)$.

Our last preprocessing step, and the most expensive one, is to renumber the nonterminals so that $i < j \Leftrightarrow \mathcal{F}(X_i)^{rev} < \mathcal{F}(X_j)^{rev}$, where S^{rev} is string S read backwards (usefulness of this will be apparent later). The sorting can be done in time $O(u + n \lg n)$ and $O(u \lg u)$ bits of space following the same approach as in Section 4.4.4, which dominates the previous time complexities. Let us say that X_n became X_s after the reordering.

We now define a structure that will be the key to our index.

```
Data: G
Result: modifies G
cnt[1..n] ← 0
for i ∈ [1,...,n] do
    if X_i generates a terminal symbol then
        continue
    rules[] ← rules generated by X_i
    for j ∈ rules do
        cnt[j] ← cnt[j] + 1
for i ∈ [1,...,n−1] do
    if X_i generates a terminal symbol then
        continue
    rules[] ← rules generated by X_i
    for j ∈ rules do
        if cnt[j] = 1 then
            Rewrite X_i replacing X_j with its content
            Remove X_j
```

Algorithm 4: Preprocessing of a general grammar.

Definition 4.6. *The* grammar tree *of $\mathcal{G}$ is a general tree $\mathcal{T}_\mathcal{G}$ with nodes labeled in $\mathcal{X}$. Its root is labeled X_s. Let $\alpha_s = X_{s_1} \dots X_{s_k}$. Then the root has k children labeled $X_{s_1}, \dots, X_{s_k}$. The subtrees of these children are defined recursively, left to right, so that the first time we find a symbol X_i in the parse tree, we define its children using α_i. However, in the future when we find a symbol X_i in our recursive left-to-right traversal, we leave it as a leaf of the grammar tree (if we expanded it, the resulting tree would be the* parse tree *of T with u leaf nodes). Also symbols $X_a \to a$ are not expanded but left as leaves. We say that X_i is* defined *in the only internal node of $\mathcal{T}_\mathcal{G}$ labeled X_i.*

Since each right-hand side $\alpha_i \notin \Sigma$ is written once in the tree, plus the root X_s, the total number of nodes in $\mathcal{T}_\mathcal{G}$ is $N + 1$.

Figure 4.3 shows the reordering and grammar tree for a grammar generating the string *"al-abaralalabarda"*.

The grammar tree partitions T in a way that is useful for finding occurrences, using the same approach as before.

Definition 4.7. *Let $X_{l_1}, X_{l_2}, \dots$ be the nonterminals labeling the consecutive leaves of $\mathcal{T}_\mathcal{G}$. Let $T_i = \mathcal{F}(X_{l_i})$, then $T = T_1 T_2 \dots$ is a partition of T according to the leaves of $\mathcal{T}_\mathcal{G}$. We call occurrences of a pattern P* primary *relative to the given partition if $|P| > 1$ and spans more than one T_i. Other occurrences are called* secondary.

This definition matches the concept of primary and secondary occurrences presented earlier in this chapter, but is defined in terms of the parse tree of the grammar.

$$
\begin{array}{ll}
\bar{X}_1 \to a & \\
\bar{X}_2 \to b & \\
\bar{X}_3 \to d & \\
\bar{X}_4 \to l & \\
\bar{X}_5 \to r & \\
\bar{X}_6 \to \bar{X}_1\bar{X}_5 & \\
\bar{X}_7 \to \bar{X}_1\bar{X}_4\bar{X}_1\bar{X}_2 & \\
\bar{X}_8 \to \bar{X}_7\bar{X}_6 & \\
\bar{X}_9 \to \bar{X}_8\bar{X}_1\bar{X}_4\bar{X}_8\bar{X}_3\bar{X}_1 &
\end{array}
\quad \Rightarrow \quad
\begin{array}{ll}
X_1 \to a & a \\
X_2 \to X_9X_1X_6X_9X_5X_1 & alabaralalabarda \\
X_3 \to b & b \\
X_4 \to X_1X_6X_1X_3 & alab \\
X_5 \to d & d \\
X_6 \to l & l \\
X_7 \to r & r \\
X_8 \to X_1X_7 & ar \\
X_9 \to X_4X_8 & alabar
\end{array}
$$

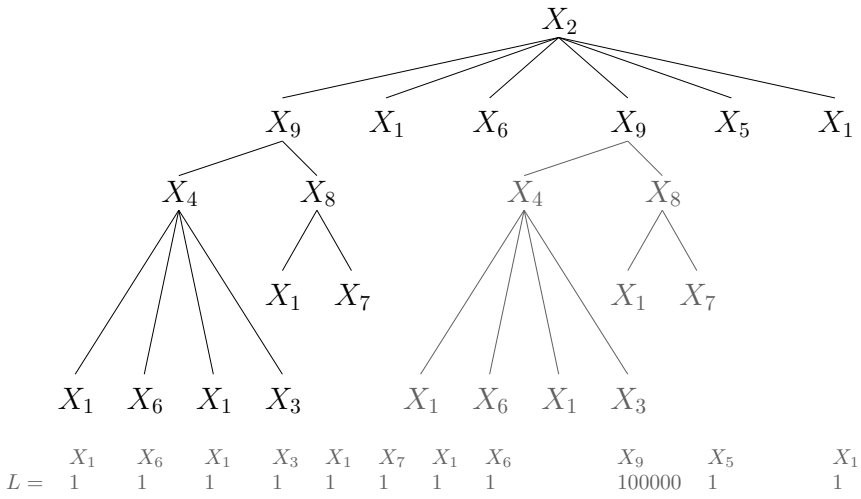

Figure 4.3: On top left, a grammar $\mathcal{G}$ generating string *"alabaralalabarda"*. On top right, our reordering of the grammar and strings $\mathcal{F}(X_i)$. In the middle, the grammar tree $\mathcal{T}_\mathcal{G}$ in black; the whole parse tree includes also the grayed part. On the bottom we show our bitmap L (Section 4.6.2).

Our self-index will represent $\mathcal{G}$ using two main components. One represents the grammar tree $\mathcal{T}_\mathcal{G}$ using a DFUDS representation (see Chapter 1) and a sequence of labels (see also Chapter 1). This will be used to extract the text and decompress rules. When augmented with a secondary trie $\mathcal{T}_S$ storing leftmost/rightmost paths in $\mathcal{T}_\mathcal{G}$, the representation will expand any prefix/suffix of a rule in optimal time [51].

The second component in our self-index corresponds to a labeled binary relation, where $B = \mathcal{X}$ and A is the set of proper suffixes starting at positions $j+1$ of rules α_i: $(\alpha_i[j], \alpha_i[j+1..])$ will be related for all $X_i \to \alpha_i$ and $1 \le j < |\alpha_i|$. The labels are numbers in the range $[1, N+1]$; we specify their meaning later. This binary relation will be used to find the primary occurrences of the search pattern. Secondary occurrences will be tracked in the grammar tree.

4.6.2 Extracting Text

We first describe a simple structure that extracts the prefix of length ℓ of any rule in $O(\ell + h)$ time. We then augment this structure to support extracting any substring of length ℓ in time $O(\ell + h \lg(N/h))$, and finally augment it further to retrieve the prefix or suffix of any rule in optimal $O(\ell)$ time. This last result is fundamental for supporting searches, and is obtained by extending the structure proposed by Gasieniec et al. [51] for SLPs to general context-free grammars generating one string. The improvement does not work for extracting arbitrary substrings, as in that case one has to first find the nonterminals that must be expanded. This subproblem is not easy, especially in little space.

As said, we represent the grammar tree $\mathcal{T}_\mathcal{G}$ using DFUDS. The sequence of labels associated to the tree nodes is stored in preorder in a sequence $X[1..N+1]$, using the representation of Golynski et al., where we choose constant time for $access(X, i) = X[i]$ and $O(\lg \lg n)$ time for $select_a(X, j)$.

We also store a bitmap $Y[1..n]$ that marks the rules of the form $X_i \to a \in \Sigma$ with 1-bit. Since the rules have been renumbered in (reverse) lexicographic order, every time we find a rule X_i such that $Y[i] = 1$, we can determine the terminal symbol it represents as $a = rank_1(Y, i)$ in constant time. This is the same Y bitmap as before.

Expanding Prefixes of Rules

Expanding a rule X_i that does not correspond to a terminal is done as follows. By the definition of $\mathcal{T}_\mathcal{G}$, the first left-to-right occurrence of X_i in sequence X corresponds to the definition of X_i; all the rest are leaves in $\mathcal{T}_\mathcal{G}$. Therefore, $v = node_{\mathcal{T}_\mathcal{G}}(select_{X_i}(X, 1))$ is the node in $\mathcal{T}_\mathcal{G}$ where X_i is defined. We traverse the subtree rooted at v in DFS order. Every time we reach a leaf u, we compute its label $X_j = X[preorder_{\mathcal{T}_\mathcal{G}}(u)]$, and either output a terminal if $Y[j] = 1$ or recursively expand X_j. This is in fact a traversal of the *parse tree* starting at node v, using the grammar tree instead. Such a traversal takes $O(\ell + h_v)$ steps, where $h_v \le h$ is the height of the parsing subtree rooted at v. In particular, if we extract the whole rule X_i we pay $O(\ell)$ steps, since we have removed unary paths in the preprocessing of $\mathcal{G}$ and thus v has $\ell > h_v$ leaves in the parse tree. The

only obstacle to having constant-time steps are the queries $select_{X_i}(X,1)$. As these are only for the position 1, we can have them precomputed in a sequence $F[1..n]$ using $n\lceil \lg N \rceil = n\lg n + O(N)$ further bits of space.

The total space required for $\mathcal{T}_\mathcal{G}$, considering the DFUDS representation, sequence X, bitmap Y, and sequence F, is $N\lg n + n\lg n + o(N\lg n)$ bits. We reduce the space to $N\lg n + \delta n\lg n + o(N\lg n)$, for any $0 < \delta \leq 1$, as follows. Form a sequence $X'[1..N-n+1]$ where the first position of every symbol X_i in X has been removed, and mark those first positions in X in a bitmap $Z[1..N+1]$, with a 1. Replace the sequence F by a permutation $\pi[1..n]$ so that $select_{X_i}(X,1) = F[i] = select_1(Z,\pi[i])$. We can still access any $X[i] = X'[rank_0(Z,i)]$ if $Z[i] = 0$. For the case $Z[i] = 1$ we have $X[i] = \pi^{-1}[rank_1(Z,i)]$. Similarly, $select_{X_i}(X,j) = select_0(Z, select_{X_i}(X', j-1))$ for $j > 1$. So, use Z, π, and X' instead of F and X.

All the operations retain the same times except for the access to π^{-1}. For π we use the representation by Munro et al. [86] that takes $(1+\delta)n\lg n$ bits and computes any $\pi[i]$ in constant time and any $\pi^{-1}[j]$ in time $O(1/\delta)$, which will be the cost to access X. Although this will have an impact later, we note that for extraction we only access X at leaf nodes, where it always takes constant time.[1]

Extracting Arbitrary Substrings

In order to extract any given substring of T, we add a bitmap $L[1..u+1]$ that marks with a 1 the first position of each T_i in T. We can then compute the starting position of any node $v \in \mathcal{T}_\mathcal{G}$ as $select_1(L, leafrank_{\mathcal{T}_\mathcal{G}}(v)+1)$.

To extract $T[p, p+\ell-1]$, we binary search the starting position p from the root of $\mathcal{T}_\mathcal{G}$. If we arrive at a leaf not representing a terminal, we go to its definition in $\mathcal{T}_\mathcal{G}$, translate position p to the area below the new node v, and continue recursively. At some point we reach position p, and from there on we extract the symbols rightwards. Just as before, the total number of steps is $O(\ell + h)$. Yet, the h steps require binary searches. As there are at most h binary searches among the children of different tree nodes, and there are $N+1$ nodes, at worst the binary searches cost $O(h\lg(N/h))$. The total cost is $O(\ell + h\lg(N/h))$.

The number of ones in L is at most N. Since we only need $select_1$ on L, we can use an ID representation (see Chapter 1), requiring $N\lg(u/N) + O(N + \lg\lg u) = N\lg(u/N) + O(N)$ bits (since $N \geq \lg u$ in any grammar). Thus the total space becomes $N\lg n + N\lg(u/N) + \delta n\lg n + o(N\lg n)$ bits.

Optimal Expansion of Rule Prefixes and Suffixes

Our improved version builds on the proposal by Gasieniec et al. [51]. We extend their representation using succinct data structures in order to handle general grammars instead of only

[1] Nonterminals $X_a \to a$ do not have a definition in $\mathcal{T}_\mathcal{G}$, so they are not extracted from X nor represented in π, thus they are accessed in constant time. They can be skipped to from $\pi[1..n]$ with bitmap Y, so that in fact π is of length $n - \sigma$ and is accessed as $\pi[rank_0(Y,i)]$; for π^{-1} we actually use $select_0(Y, \pi^{-1}[j])$.

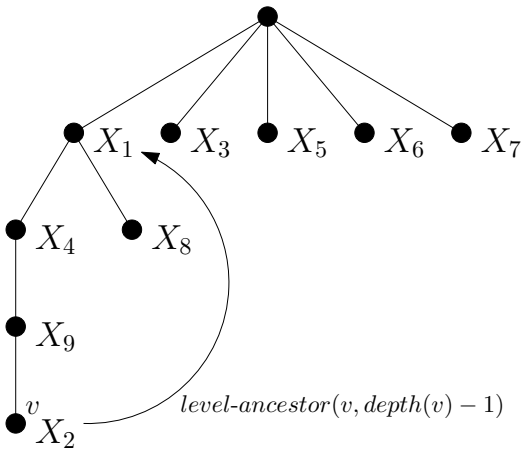

Figure 4.4: Example of the trie of leftmost paths for the grammar of Figure 4.3. The arrow pointing from X_2 to X_1 ilustrates the procedure to determine the first terminal symbol generated by X_2.

SLPs. Using their notation, call $S(X_i)$ the string of labels of the nodes in the path from any node labeled X_i to its leftmost leaf in *the parse tree* (we take as leaves the nonterminals $X_a \in \mathcal{X}$, not the terminals $a \in \Sigma$). We insert all the strings $S(X_i)^{rev}$ into a trie $\mathcal{T}_S$. Note that each symbol appears only once in $\mathcal{T}_S$ [51], thus it has n nodes. Again, we represent the topology of $\mathcal{T}_S$ using DFUDS. Yet, its sequence of labels $X_S[1..n]$ turns out to be a permutation in $[1..n]$. We represent it once again with the structure [86] that takes $(1+\epsilon)n \lg n$ bits and computes any $X_S[i]$ in constant time and any $X_S^{-1}[j]$ in time $O(1/\epsilon)$.

We can determine the first terminal in the expansion of X_i, which labels node $v \in \mathcal{T}_S$, as follows. Since the last symbol in $S(X_i)$ is a nonterminal X_a representing some $a \in \Sigma$, it follows that X_i descends in $\mathcal{T}_S$ from X_a, which is a child of the root. This node is $v_a = level\text{-}ancestor_{\mathcal{T}_S}(v, depth_{\mathcal{T}_S}(v) - 1)$. Then $a = rank_1(Y, X_S[preorder_{\mathcal{T}_S}(v_a)])$.

Figure 4.4 shows an example of this particular query in the trie for the grammar presented in Figure 4.3.

A prefix of X_i is extracted as follows. First, we obtain the corresponding node $v \in \mathcal{T}_S$ as $v = X_S^{-1}[X_i]$. Then we obtain the leftmost symbol of v as explained. The remaining symbols descend from the second and following children, in the parse tree, of the nodes in the upward path from a node labeled X_i to its leftmost leaf, or which is the same, of the nodes in the downward path from the root of $\mathcal{T}_S$ to v. Therefore, for each node w in the list $level\text{-}ancestor_{\mathcal{T}_S}(v, depth_{\mathcal{T}_S}(v) - 2), \ldots, parent_{\mathcal{T}_S}(v), v$, we map w to $x \in \mathcal{T}_G$, $x = node_{\mathcal{T}_G}(select_{X_j}(X, 1))$ where $X_j = X_S[preorder_{\mathcal{T}_S}(w)]$. Once x is found, we recursively expand its children, from the second onwards, by mapping them back to $\mathcal{T}_S$. Charging the cost to the new symbol to be expanded, and since there are no unary paths, it follows that we carry out $O(\ell)$ steps to extract the first ℓ symbols, and the extraction is in real-time [51]. All costs per step are $O(1)$ except for the $O(1/\epsilon)$ to access X_S^{-1}.

For extracting suffixes of rules in $\mathcal{G}$, we need another version of $\mathcal{T}_S$ that stores the rightmost paths. This leads to our first result (choosing $\delta = o(1)$).

Lemma 4.2. *Let a sequence $T[1..u]$ be represented by a context free grammar with n symbols, size N, and height h. Then, for any $0 < \epsilon \leq 1$, there exists a data structure using at most $N \lg n + N \lg(u/N) + (2+\epsilon)n \lg n + o(N \lg n)$ bits of space that extracts any substring of length ℓ from T in time $O(\ell + h \lg(N/h))$, and a prefix or suffix of length ℓ of the expansion of any nonterminal in time $O(\ell/\epsilon)$.*

4.6.3 Locating Patterns

A secondary occurrence of the pattern P inside a leaf of $\mathcal{T}_\mathcal{G}$ labeled by a symbol X_i occurs as well in the internal node of $\mathcal{T}_G$ where X_i is defined. If that occurrence is also secondary, then it occurs inside a child X_j of X_i, and we can repeat the argument with X_j until finding a primary occurrence inside some X_k. Thus, to find all the secondary occurrences, we can first spot the primary occurrences, and then find all the copies of the nonterminal X_k that contain the primary occurrences, as well as all the copies of the nonterminals that contain X_k, recursively.

As before, we base our approach on the strategy proposed by Kärkkäinen [68] to find the primary occurrences of $P = p_1 p_2 \ldots p_m$. Kärkkäinen considers the $m-1$ partitions $P = P_1 \cdot P_2$, $P_1 = p_1 \ldots p_i$ and $P_2 = p_{i+1} \ldots p_m$, for $1 \leq i < m$. In our case, for each partition we will find all the nonterminals $X_k \to X_{k_1} X_{k_2} \ldots X_{k_r}$ such that P_1 is a suffix of some $\mathcal{F}(X_{k_i})$ and P_2 is a prefix of $\mathcal{F}(X_{k_{i+1}}) \ldots \mathcal{F}(X_{k_r})$. This finds each primary occurrence exactly once. The secondary occurrences are then tracked in the grammar tree $\mathcal{T}_\mathcal{G}$. We handle the case m=1 by finding all occurrences of X_{p_1} in $\mathcal{T}_\mathcal{G}$ using select over the labels, and treat them as primary occurrences.

Finding Primary Occurrences

As anticipated at the end of Section 4.6.1, we store a binary relation $\mathcal{R} \subseteq A \times B$ to find the primary occurrences. It has n rows labeled X_i, for all $X_i \in \mathcal{X} = B$, and $N - n$ columns [2]. Each column corresponds to a distinct proper suffix $\alpha_i[j+1..]$ of a right-hand side α_i. The labels belong to $[1..N+1]$. The relation contains one pair per column: $(\alpha_i[j], \alpha_i[j+1..]) \in \mathcal{R}$ for all $1 \leq i \leq n$ and $1 \leq j < |\alpha_i|$. Its label is the preorder of the $(j+1)$st child of the node that defines X_i in $\mathcal{T}_\mathcal{G}$. The space for the binary relation is $(N-n)(\lg n + \lg N) + O(N)$ bits.

Recall that, in our preprocessing, we have sorted $\mathcal{X}$ according to the lexicographic order of $\mathcal{F}(X_i)^{rev}$. We also sort the suffixes $\alpha_i[j+1..]$ lexicographically with respect to their expansion, that is $\mathcal{F}(\alpha_i[j+1])\mathcal{F}(\alpha_i[j+2]) \ldots \mathcal{F}(\alpha_i[|\alpha_i|])$. This can be done in $O(u + N \lg N)$ time in a way similar to how $\mathcal{X}$ was sorted: Each suffix $\alpha_i[j+1..]$, labeled p, can be associated to the substring $T[select_1(L, rankleaf_{\mathcal{T}_\mathcal{G}}(node_{\mathcal{T}_\mathcal{G}}(p)) + 1) \ldots select_1(L, rankleaf_{\mathcal{T}_\mathcal{G}}(v) + 1 + numleaves_{\mathcal{T}_\mathcal{G}}(v)) - 1]$, where v is the parent of $node_{\mathcal{T}_\mathcal{G}}(p)$. Then we can proceed as in our construction for SLPs.

Figure 4.5 illustrates how $\mathcal{R}$ is used for the grammar presented in Figure 4.3.

[2] Recall $\mathcal{F}(X_i) \leq \mathcal{F}(X_j)$ iff $i \leq j$

Given P_1 and P_2, we first find the range of rows whose expansions finish with P_1, by binary searching for P_1^{rev} in the expansions $\mathcal{F}(X_i)^{rev}$. Each comparison in the binary search needs to extract $|P_1|$ terminals from the suffix of $\mathcal{F}(X_i)$. According to Lemma 4.2, this takes $O(|P_1|/\epsilon)$ time. Similarly, we binary search for the range of columns whose expansions start with P_2. Each comparison needs to extract $\ell = |P_2|$ terminals from the prefix of $\mathcal{F}(\alpha_i[j+1])\mathcal{F}(\alpha_i[j+2])\ldots$. Let r be the column we wish to compare to P_2. We extract the label p associated to the column in constant time. Then we extract the first ℓ symbols from the expansion of $node_{\mathcal{T}_\mathcal{G}}(p)$. If $node_{\mathcal{T}_\mathcal{G}}(p)$ does not have enough symbols, we continue with $nextsibling_{\mathcal{T}_\mathcal{G}}(p)$, and so on, until we extract ℓ symbols or we exhaust the suffix of the rule. According to Lemma 4.2, this requires time $O(|P_2|/\epsilon)$. Thus our two binary searches require time $O((m/\epsilon)\lg N)$.

This time can be further improved by using the technique of building a trie of sampled phrases. We sample phrases at regular intervals and store the sampled phrases in a Patricia tree [84]. We first search for the pattern in the Patricia tree, and then complete the process with a binary search between two sampled phrases (we first verify the correctness of the Patricia search by checking that our pattern is actually within the range found). By sampling every $\lg u \lg\lg n/\lg n$ phrases, the resulting time for searching becomes $O\left(m \lg\left(\frac{\lg u}{\lg n}\right)\right)$ and we only require $o(N \lg n)$ bits of extra space, as the Patricia tree needs $O(\lg u)$ bits per node.

Once we identify a range of rows $[a_1, a_2]$ and of columns $[b_1, b_2]$, we retrieve all the k points in the rectangle and their labels in time $O((k+1)\lg n/\lg\lg n)$. The parents of all the nodes $node_{\mathcal{T}_\mathcal{G}}(p)$, for each point p in the range, correspond to the primary occurrences. In Section 4.6.3 we show how to report primary and secondary occurrences starting directly from those $node_{\mathcal{T}_\mathcal{G}}(p)$ positions.

We have to carry out this search for $m-1$ partitions of P, whereas each primary occurrence is found exactly once. Calling occ the number of primary occurrences, the total cost of this part of the search is $O\left((m^2/\epsilon)\lg\left(\frac{\lg u}{\lg n}\right) + (m + occ)\lg n/\lg\lg n\right)$.

Tracking Occurrences Through the Grammar Tree

The remaining problem is how to track all the secondary occurrences triggered by a primary occurrence, and how to report the positions where these occur in T. Given a primary occurrence for partition $P = P_1 \cdot P_2$ located at $u = node_{\mathcal{T}_\mathcal{G}}(p)$, we obtain the starting position of P in T by moving towards the root while keeping count of the offset between the beginning of the current node and the occurrence of P. Initially, for node u itself, this is $l = -|P_1|$. Now, while u is not the root, we set $l \leftarrow l + select_1(L, rankleaves_{\mathcal{T}_\mathcal{G}}(u)+1) - select_1(L, rankleaves_{\mathcal{T}_\mathcal{G}}(parent_{\mathcal{T}_\mathcal{G}}(u))+1)$, then $u \leftarrow parent_{\mathcal{T}_\mathcal{G}}(u)$. When we reach the root, the occurrence of P starts at l.

It seems like we are doing this h times in the worst case, since we need to track the occurrence up to the root. In fact we might do so for some symbols, but the total cost is amortized. Every time we move from u to $v = parent_{\mathcal{T}_\mathcal{G}}(u)$, we know that $X[v]$ appears at least once more in the tree. This is because of our preprocessing (Section 4.6.1), where we force rules to appear at least twice or be removed. Thus v defines $X[v]$, but there are one or more leaves labeled $X[v]$, and we have to report the occurrences of P inside them all. For this sake we carry out $select_{X[v]}(X, i)$

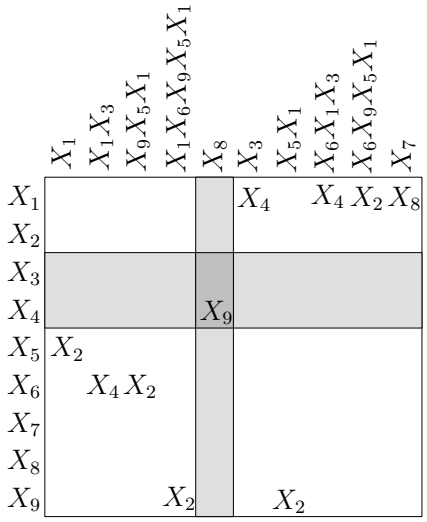

Figure 4.5: Relation $\mathcal{R}$ for the grammar presented in Figure 4.3. The highlighted ranges correspond to the result of searching for $b \cdot ar$, where the single primary occurrence corresponds to X_9.

for $i = 1, 2\ldots$ until spotting all those occurrences (where P occurs with the current offset l). We recursively track them to the root of $\mathcal{T}_\mathcal{G}$ to find their absolute position in T, and recursively find the other occurrences of all their ancestor nodes. The overall cost amortizes to $O(1)$ steps per occurrence reported, as we can charge the cost of moving from u to v to the other occurrence of v. If we report occ secondary occurrences we carry out $O(occ)$ steps, each costing $O(\lg\lg n)$ time. We can thus use $\delta = O(1/\lg\lg n)$ (Section 4.6.2) so that the cost to access $X[v]$ does not impact the space nor time complexity.

4.6.4 Resulting Index

By adding up the space of Lemma 4.2 with that of the labeled binary relation, and adding up the costs, we have our central result,

Theorem 4.10. *Let a sequence $T[1..u]$ be represented by a context free grammar with n symbols, size N and height h. Then, for any $0 < \epsilon \leq 1$, there exists a data structure using at most $2N\lg n + N\lg u + \epsilon\, n\lg n + o(N\lg n)$ bits that finds the occ occurrences of any pattern $P[1..m]$ in T in time $O\left((m^2/\epsilon)\lg\left(\frac{\lg u}{\lg n}\right) + (m + occ)\lg n/\lg\lg n\right)$. It can extract any substring of length ℓ from T in time $O(\ell + h\lg(N/h))$. The structure can be built in $O(u + N\lg N)$ time and $O(u\lg u)$ bits of working space.*

This is the first grammar-based index whose search time does not depend on the height of the grammar.

Our result, combined with the one by Bille et al [16], allows us to state the following corollary:

Corollary 4.4. *Let a sequence $T[1..u]$ be represented by an SLP of n symbols, there exists an index requiring $O(n \lg u)$ bits, that:*

- *Finds the occ occurrences of any pattern $P[1..m]$ in T in time $O\left(m^2 \lg\left(\frac{\lg u}{\lg n}\right) + (m + occ) \lg n\right)$.*
- *Extracts any substring of length ℓ from T in time $O(\ell + \lg u)$.*

This solves the problem of representing a grammar in $O(n)$ words while supporting search and extract independent of the height and in sublinear time with respect to the size of the text.

4.7 Concluding Remarks

We have presented the first compressed indexed text representation based on Straight-Line Programs (SLPs), which are as powerful as context-free grammars. It achieves space proportional to that of the bare SLP representation in most relevant cases and, in addition to just uncompressing, it permits extracting arbitrary substrings of the text, as well as carrying out pattern searches, in time usually sublinear on the grammar size. We also showed how to extend this to a more general class of grammars, and gave some by products: extracting prefixes of rules in constant time per symbol and representing labeled binary relations.

There are many interesting open problems left in this work. The main ones are:

- The space required by our indexes has an $n \lg u$ term. This can dominate the space for highly compressible text. It is quite challenging to avoid this term without significantly increasing the time complexity.
- We also have an $\tilde{O}(m^2)$ term in the search. This is unappealing for very long patterns. It would be desirable to reduce this to $\tilde{O}(m)$.
- Improving the construction to achieve time in terms of n and not u is also very interesting for highly compressible text, where u can be much larger than n.

It is remarkable that when compared to the original LZ-Index, we get close to such carefully engineered work with our general approach, even beating the original proposal [89].

5 Document Retrieval

Highly repetitive collections are becoming more and more common. We have a lot of versioned information on the Web; good examples of this are software repositories and Wikipedia[1]. It is also expected that in the future we will have to provide storage for genome sequences of many individuals of the same species [2]. This last scenario is interesting because within the same species, the sequences share close to 99.99%, making the collection extremely repetitive [90].

Being capable of storing archive data with historic information on how documents evolve is an interesting task by itself, but we also need to provide searching capabilities to make this information easily available. In this work we focus on the document listing problem for such collections.

Definition 5.1 (Document Listing)**.** *The document listing problem is defined as follows: Given a collection of documents $\mathcal{D} = \{T_1, T_2, \ldots, T_d\}$, and a query P, retrieve the documents that contain P as a substring.*

One important point to clarify is the distinction between standard text-indexing and the document listing problem. Usually text indexes allow you to search for a pattern in a text and report the positions where the pattern occurs. If we concatenate all the documents and use a classical index, we can retrieve the documents that contain the pattern. The drawback is that we are forced to iterate over all occurrences of the pattern, which means we can pay a huge overhead for just one document if it contains the pattern many times. This difference renders classical text indexes unsuitable for certain instances of the problem.

For natural language, the problem has been usually simplified by using an inverted index. For every word w in the language, we have a list $L[w] = \{T_{i_1}, T_{i_2}, \ldots, T_{i_\ell}\}$ giving all documents that contain w, plus extra information to compute the ranking function. Answering a query $\mathcal{Q} = \{P_1, P_2, \ldots, P_q\}$ corresponds to obtaining a subset of the elements in $\cap_{i=1}^{q} L[P_i]$.

This solution has been shown to be effective in space, retrieval time, and quality, but it lacks the freedom we would expect in other domains. For example, if we consider a collection of DNA sequences, the concept of a word is not well defined. For this reason, solutions in one domain may be completely useless in others. We also see this phenomenon in languages in which the separation between words is not clearly defined, or hard to determine automatically.

This chapter presents a practical index for document listing derived mainly from the results from Chapters 3 and 4. We include some definitions and revisit the index, and also go modify the structures to obtain a more practical index.

The main idea behind our proposal is as follows: Given a collection of documents, we compress the whole set of texts using a grammar-compressor [78, 119, 22]. The resulting file is indexed using the result of Theorem 4.10. Then we augment the structure with a set of inverted

[1] http://wikipedia.org

[2] This is already happening with the 1000genomes project, http://www.1000genomes.org/

lists for non-terminal symbols. These inverted lists store the documents that contain each non-terminal. The queries are answered by first asking the text index to produce the minimum set of non-terminals that match the query pattern. Once we have the inverted lists for the appropriate nonterminals, we compute the union of these lists. This generates the final result, an inverted list for the query pattern.

The main contributions of this chapter:

- We show how to extend the grammar-compressed index from Chapter 4 to support document listing in a simple and clean way. The index also supports access to any document of the collection, verbatim, so it completely replaces the original input. Building our index on top of any grammar-compressor allows us to achieve a good space bound for repetitive sequences, including versioned documents. In addition to achieving good space bounds [27, 28], a straight-forward grammar representation allows fast decompression, and therefore, access to the content being indexed [31, 27, 28].
- The resulting structure supports retrieving the inverted list for an arbitrary pattern. This is particularly interesting, since all the algorithms developed for plain posting lists can be applied to the output of our searches. This allows us to easily extend our result to support conjunctive queries.
- We can apply the same result for words in natural language, allowing an interesting new index. This index does not support full-text document listing, but solves the problem of searching for phrases, a problem that is also hard to handle with traditional inverted indexes.
- We discuss how to extend our final index to compute other interesting pieces of information commonly used by ranking functions:
  - Term frequencies for each document.
  - Positional information on where patterns occur inside each document.

5.1 Related Work

Most of the items in our index are built using grammar compression and indexes. Recall the definition from Chapter 4; we have a set of non- terminals $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$ and each non-terminal $X_i \to \alpha_i$ generates either a single terminal symbol, or a sequence of non- terminals, with no circular dependencies. $\mathcal{F}(X_i)$ is the result of recursively replacing all non-terminals until a sequence of terminal symbols is obtained. We also refer to $\mathcal{F}(X_i)^{rev}$ as $\mathcal{F}(X_i)$ reversed (i.e., read from right to left).

We say that $\mathcal{G}$ compresses $T = t_1 t_2 \ldots t_u$, iff $\mathcal{F}(X_s) = T$. (X_s is the starting symbol.)

We denote by N the sum of the sizes of all the right sides in the grammar, that is

$$N = \sum_{i=1}^{n} |\alpha_i|$$

The *height of the grammar* is the length of the longest path from the start symbol to any terminal symbol in the parse tree.

The grammar-based index from Theorem 4.10 is used to support one of the steps in our search procedure. In Section 5.1.1, we explain in neater detail the pieces required.

5.1.1 Grammar Indexes

The original index takes as input a free-context grammar that generates a single sequence. We call $\mathcal{G}$ the grammar, composed of a set of non-terminals $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, and an initial symbol X_s.

First the grammar is preprocessed to remove duplicate rules, and to embed rules that are mentioned only once inside the rule that mentions them. This does not increase the size of the grammar, but allows us to bound some of the running times further (see Chapter 4).

For the construction of the index, we first preprocess the grammar and re-assign the identifiers of each non-terminal so that they are sorted lexicographically by the reverse of the string they generate, i.e., $\mathcal{F}(X_i)^{rev}$. We number the non-terminals in sorted order, that is, $\mathcal{F}(X_i)^{rev} \leq \mathcal{F}(X_j)^{rev}$ iff $i \leq j$. We then create a bitmap Y where we assign a 1 to position i iff X_i generates just a single terminal symbol.

By using Y we can tell whether a rule generates more non-terminals or just a single terminal symbol. Given X_i, if $access_Y(i) = 1$, then we know it generates a terminal symbol. Furthermore, if we assume terminal symbols are contiguous, we know that X_i generates $rank_Y(1, i)$. It is also possible to obtain the non-terminal X_j that generates symbol a by computing $j = select_Y(1, a)$. (See Chapters 1 and 4.)

In addition, for each proper suffix of each rule, we assign an id, and then relabel them according to the lexicographical order of the strings generated by those proper suffixes. We will call this SuffPerm. In other words, SuffPerm stores at position i the i-th proper suffix of a rule in lexicographical order.

Finally, we create a labeled binary relation $\mathcal{R}$ that maps $\texttt{SuffPerm}[i]$ to j through a label k if rule j appears before the suffix represented by $\texttt{SuffPerm}[i]$ in rule k.

We want to support range searching in $\mathcal{R}$, and this can be done in $O(\lg n / \lg \lg n)$ time within $O(n \lg N)$ bits of space (see Chapter 3).

In Chapter 4, the grammar is represented as a tree, with $N - n$ leaves. In this case a simple representation of the grammar is enough, we do not need to navigate the parse tree upwards, and in practice the theoretical solution for fast access works slower than traversing a plain representation.

To find the primary occurrences of a pattern $P = p_1 p_2 \dots p_m$, we try the m possible partitions: $p_1 \cdot p_2 \dots p_m$, $p_1 p_2 \cdot p_3 \dots p_m$, up to $p_1 \dots p_{m-1} \cdot p_m$. For each partition $P = P_l \cdot P_r$, we perform a binary search on the rules to determine which ones finish with P_l. Then we perform a binary search over the suffixes of rules, `SuffPerm`, to find suffixes of rules that begin with P_2. Finally, using the binary relation $\mathcal{R}$, we can perform a range search to retrieve the non-terminals that contain elements that start with P_2 preceded by elements that end with P_1.

Secondary occurrences are obtained by following up the primary occurrences in the parse tree. As we will explain later, we only care about primary occurrences in this work, so we do not deal with an efficient representation for the parse tree to track secondary occurrences.

Chapter 4 shows how to represent `SuffPerm` in little space other than that of the binary relation, and also how to extract prefixes of suffixes of rules in linear time. We do not need the technical details of these results, it suffices to know the running time of each step. The binary search for P_1 requires $O(m \lg n)$ time. The binary search for P_2 requires $O(m \lg N)$ time. Finally, retrieving the primary occurrences requires $O(\lg n / \lg \lg n)$ time per element retrieved.

Retrieving all $\mathrm{occ_p}$ primary occurrences requires $O(m^2 \lg N + \mathrm{occ_p} \lg n / \lg \lg n)$ time. As we will see later, this is the only thing we need, since we do not care about secondary occurrences.

5.1.2 Re-Pair

Due to its simplicity, we chose Re-Pair as the grammar compression [78] for evaluating our index. It is important to point out that other grammar compressors may achieve better results, but may cause difficulty when implemented on a large scale. Algorithm 5 shows the high-level pseudo-code for Re-Pair. Surprisingly, this can be implemented in linear time [78], and it is also possible to trade compression speed and space for compression ratio using an approximate version [31].

We post-process the result of Re-Pair to make the final grammar smaller. For each rule X_i that generates a set of non-terminals, if it is mentioned only once in the grammar by rule X_j, we expand X_i where X_j mentions it, and then remove X_i. We repeat this process until each remaining rule is mentioned at least twice in the grammar.

This is required by the index, but it also has the nice property that matches the dictionary compression algorithm proposed by González and Navarro [57], that has shown to improve the final result considerably (see [57, 31]).

5.2 The Index

In this section we describe how to build the index, augment it to support document listing, and how queries are answered.

```
Result: G = (X, σ, Γ, s)
G ← new Grammar
G.σ ← max{T}
create σ rules generating each terminal symbol
next ← σ + 1
while true do
    find the most frequent pair of symbols, p, in T
    if p appears less than 2 times in T then
        replace all terminals in T by the non-terminal that generates it
        create X_{next−σ} and add it to G.X
        add X_{next−σ} → T
        return G
    replace all occurrences of p in T by next
    create X_{next−σ} and add it to G.X
    add X_{next−σ} → p
    next ← next + 1
return G
```

Algorithm 5: Re-Pair algorithm adapted to produce a grammar

5.2.1 Construction for Primary Occurrences

From the whole collection $\mathcal{D} = \{T_1, T_2, \ldots, T_d\}$, we generate a single sequence

$$T = \$_0 T_1 \$_1 T_2 \$_2 \ldots \$_{d-2} T_{d-1} \$_{d-1} T_d,$$

where $\$_i$ are symbols that do not appear anywhere else in the collection.

When we compress this sequence with Re-Pair, we are sure that no rule spans from one document to the other, since the $\$_i$ symbols cannot form pairs that appear twice.

We then remove the $\$_i$ elements, and generate one rule per document, containing all the elements left between the \$ symbols in X_s. After that, we replace X_s by a new rule that generates the new rules we just created, in order. This allows us to have direct access to a rule that generates the whole content for any document.[3]

Our grammar, after this preprocessing, has the following form:

- X_s generates d non-terminals, $X_{t_1} X_{t_2} \ldots X_{t_d}$, where $\mathcal{F}(X_{t_i}) = T_i$.
- X_{t_i} generates the symbols between $\$_{i-1}$ and $\$_i$ in the original X_s generated by Re-Pair.

[3] We could modify the grammar generation algorithm so it does not generate rules that contain one unique \$ symbol. This would avoid increasing too much the alphabet size during compression.

When building the index, we leave X_s outside the permutation `SuffPerm`. This not only saves space, but makes sure that whenever we find a primary occurrence, it is contained inside a single document.

To access the i-th document in the collection, we just expand the i-th non-terminal generated by s. This allows us to retrieve documents in time proportional to their lengths.

Note that we can adapt other grammar-based compressors to this scheme. An interesting option is to just simply compress each document separately with a compressor that generates an SLP, and then apply the merge algorithm of Wan [114]. This will generate a grammar that satisfies the conditions above, and by applying the same preprocessing before constructing the index, we can optimize the output even further.

5.2.2 Adding Inverted Lists

For each non-terminal, we store an inverted list of the documents containing that non-terminal. Note that this requires at most $n \times d$ bits, and we expect n to be small. Yet this is still not satisfactory. The key point is that if two versions share much of their content, they will appear in a very similar set of lists, since they will be formed by the same non-terminals.

To exploit this, we again use grammar-compression on the sequence of lists. We could use any space-efficient representation of lists, but for repetitive ones, this particular solution has proven to work well in practice [27, 28].

Let L denote the set of inverted lists, where $L[X_i]$ is the list of documents containing non-terminal X_i. We represent the inverted lists in the same way as we represent the documents, this allows to access an entire list in time proportional to its length.

It is interesting to relate the size of these inverted lists to the size of the original sequence. It turns out that these lists can be represented space efficiently. We see the inverted lists as a grid, where coordinate (i, j) is a 1 iff non-terminal i is contained in document j. Let t be the number of points in this grid. We need $t \lg \frac{nd}{t} + O(t)$ bits to represent the grid[4].

We know that $n \leq t$, therefore, the space is bounded by $t \lg d$, which is as much as the solution by Välimaki and Mäkinen requires for the document array [112]. We can further bound the space by considering the worst possible space for the grid. The space is maximized when $t = \frac{nd}{e}$. In this case, the total space required by the grid is $O(t)$ bits.

On the other hand, we can also bound the length of the text in terms of t. We know that each point in the grid represents at least one occurrence of a rule in the collection, therefore, $u \geq t$. This means that the total extra space for the grid is bounded by the length of the collection in bits, in other words, $\frac{\mathcal{D}}{\lg \sigma}$ bits.

[4]This is a simple information theoretic lower bound, there exist representations that achieve this [43], and some that do better on repetitive cases [28], as in our case. We could also use the representations from Chapter 3, but the binary relation might be too dense.

5.2.3 Full-Text Document Listing

Having built the grammar-index, and the inverted lists, the searching becomes quite straightforward. The high-level version of the algorithm is shown in Algorithm 6.

Result: L list of documents that contain P
Find the set of primary occurrences using the result of Theorem 4.10
Compute the union of the lists associated with the primary occurrences (using Algorithm 7)

Algorithm 6: High-level description of how to find the documents containing a pattern.

It is interesting that at this stage we need to compute the union of sets, in contrast with the usual operation we encounter between inverted lists, which is the intersection. There are several ways of computing the union, for example:

- Sort the lists by length and compute the union following this order.
- Emulating a Huffman-like merge, where we put all lists (including intermediate results) into a priority queue. Then we always merge the two smallest ones.

Our case is a bit more complicated. We have a grammar-compressed version of the lists, and thus we want to make use of this fact, both to keep the space low, and to improve the query time.

Given a set of non-terminals representing the primary occurrences of the pattern, we will create a dynamic dictionary containing those elements, called *seen*, and a queue containing the same elements, we call this queue *remaining*. The merge procedure generates a dictionary containing all the elements, and is shown in Algorithm 7.

The worst case running time of this algorithm is $O(\mathrm{occ_p} \times output)$. Section 5.3 shows that in practice $\mathrm{occ_p}$ is in general small, and also that our heuristic of keeping track of previously seen non-terminals allows us to save processing time; it exploits the regularities seen between the lists. If two lists contain basically the same elements, we will only explore one of them, since we will encounter a non-terminal we have already seen.

The reason for finding only primary occurrences is quite straight-forward. Secondary occurrences contain documents we already reported as primary occurrences, thus they would only increase the processing time, but will not add anything new to the resulting set of documents.

5.2.4 Word-Based Document Listing

An interesting twist of our index, supported by the previous machinery, is that we can index word identifiers instead of symbols. Now we do not support searching for arbitrary patterns, but we can search for phrases by just following the same procedure as the one described in 5.2.3.

```
Data: Set V = {v_1, v_2, ..., v_n}, Lists G = (X, Γ, σ, s)
Result: R = (d_{i_1}, d_{i_2}, ..., d_{i_k})
remaining ← ∅
seen ← ∅
R ← ∅
for v ∈ V do
    remaining ← remaining ∪ {X_v}
    seen ← seen ∪ {X_v}
while remaining ≠ ∅ do
    x ← GetMax(remaining)
    remaining ← remaining − {x}
    if x is terminal then
        R ← R ∪ {x}
    for x^j in Γ(x) do
        if x^j ∉ seen then
            seen ← seen ∪ {x^j}
            remaining ← remaining ∪ {x^j}
return L
```

Algorithm 7: Computing the union of the lists for a set of non-terminals.

Words identifiers are terminal symbols, therefore, we can access the inverted list of a word without performing a search operation. It suffices to find the non-terminal symbol that generates the word we want and retrieve/decompress its list. It is even possible to augment the lists representation to speedup intersections to support conjunctive queries among words [28]. We can retrieve the inverted list of a word w, $I[w]$, in constant time per element retrieved (amortized in practice) as shown before ($I[w] = L[select_Y(1, w)]$).

The resulting structure is basically the standard inverted lists plus some extra information to support phrase searches. Retrieving the inverted list of a single word can be achieved in constant time per element retrieved, since we actually store those lists! We also have a tokenized representation of the collection embedded in the index.

5.2.5 Adding Ranking Information

The index can be augmented with extra information in a similar way to that used on inverted lists, with a couple of restrictions. We can augment the inverted lists that associate each non-terminal symbol with the documents that contain it with score values. In particular, frequencies offer an interesting property that is easy to exploit here.

When we augment the lists L with frequencies, if we can just add up all the values associated with primary occurrences of a certain document and we will obtain exactly the number of occurrences of the pattern in the entire document. We next formalize this before commenting on some other interesting properties that can be exploited within our structure.

Assume a function freq that gives the number of times a symbol in the grammar appears in a document. We can compute the number of occurrences of each element by simply adding these values for each primary occurrence. This is shown in Algorithm 8. We use $L[v][i]$ to denote accessing the i-th position of the inverted list for non-terminal v. We also make use of the function $\text{freq}(v,i)$ to retrieve the frequency for the i-th document associated with non-terminal v. We can see that Algorithm 8 is a simple merge-like algorithm that accumulates the frequencies together with the result. We use freq_R to accumulate the partial frequencies for the resulting set R.

```
   Data: Set V = {v_1, v_2, ..., v_k}, Lists G = (X, Γ, σ, s)
   Result: R = (d_{i_1}, d_{i_2}, ..., d_{i_k})
 1 R ← ∅
 2 for v ∈ V do
 3 |   i ← 1
 4 |   j ← 1
 5 |   while i ≤ len(L[v]) and j ≤ len(R) do
 6 |   |   if L[v][i] = R[j] then
 8 |   |   |   freq_R[j] ← freq_R[j] + freq(v, i)
 9 |   |   |   i ← i + 1
10 |   |   |   j ← j + 1
11 |   |   else if L[v][i] < R[j] then
12 |   |   |   insert L[v][i], freq(v, i) at position j in R
13 |   |   |   i ← i + 1
14 |   |   else
15 |   |   |   j ← j + 1
16 return R
```

Algorithm 8: Computing the union of the lists for a set of non-terminals together with their accumulated frequencies.

A simple bound on the running time is $O(output \times \text{occ}_\text{p})$, where occ_p is the number of primary occurrences. This bound is the same as that of the previous algorithm, yet in this case we cannot easily apply our heuristic in Algorithm 7 that allows to save work through identifying repeated non-terminals.

We still have to prove that Algorithm 8 actually retrieves the correct frequencies for each document.

Lemma 5.1. *Algorithm 8 correctly computes the frequency at which the pattern P appears in each document containing it.*

Proof. To prove the equality, we will show that for any element $r \in R$, with frequency f_r, $\text{freq}_R[r] \leq f_r$ and $\text{freq}_R[r] \geq f_r$.

- $\text{freq}_R[r] \leq f_r$: By contradiction, assume $\text{freq}_R[r] > f_r$. That means at least one occurrence is being counted twice. This has to happen in line 8, so there is an occurrence of the

pattern reported by two non-terminals as primary occurrence. Since every occurrence is associated only with one primary occurrence, this is a contradiction.

- $\text{freq}_R[r] \geq f_r$: Also by contradiction, assume $\text{freq}_R[r] < f_r$. For this to happen, there is at least one occurrence of P that is not covered by a primary occurrence. This is also a contradiction, since every occurrence of the pattern is associated with a primary occurrence.

This concludes the proof, therefore, $f_r = \text{freq}_R[r]$. □

It is interesting to note that we may not need to store the frequencies for each possible occurrence of a document in the inverted lists. We could store an approximation to the frequency, allowing Algorithm 8 to approximate the term frequency and save space, by storing values from a smaller universe. For example, storing the logarithms of each frequency.

We can also use the result from Chapter 4 to support locating the occurrences of the pattern in the collection. This allows to obtain positional information for the query when required. Another interesting option here is to approximate the locations of multiple patterns depending on the primary occurrences. However, this line of work is beyond our scope.

5.3 Experimental Results

5.3.1 Practical Considerations

For the practical implementation, we did not implement the real-time access to prefixes/suffixes of rules as described in Chapter 4. We just store the grammar as a set of arrays describing each rule. Furthermore, we do not need the tree, since we are not tracking occurrences upwards.

The binary relation is represented using a wavelet tree, as implemented in Libcds[5]. We also make use of the arrays implemented in the library. We use Navarro's implementation of Re-Pair[6], which runs in linear time.

As containers we use the standard C++ STL containers. For sets we use `set`, and for unsorted sequences, we use `vector`.

5.3.2 Experimental Setup

To test our index we downloaded the first part of the English version of Wikipedia[7], and sampled documents from it uniformly at random. For each document selected, we extracted all its versions. This was done using the `go-wikiparse` library[8].

[5] Available at http://libcds.recoded.cl

[6] Available at http://www.dcc.uchile.cl/gnavarro/software/

[7] enwiki-20110722-pages-meta-history1.xml

[8] Available at https://github.com/dustin/go- wikiparse.

Dataset	size	# docs	versions/doc (avg)	mutation rate	Re-Pair
Wiki1	69MB	8	582	-	0.36MB
Wiki2	600MB	20	772.85	-	3.45MB
Wiki3	1.5GB	36	831.08	-	5.50MB
DNA1	1000MB	1	1000	0.01%	4.5MB
DNA2	1000MB	1	1000	0.005%	2.09MB
DNA3	1000MB	1	1000	0.0026%	1.17MB

Table 5.1: Datasets

We also generated synthetic collections composed of symbols A, C, G and T. This is to mimic the compression of genome databases. The process of generation is the following:

- Generate a random sequence T_1 of length n.
- Generate $d-1$ copies of T_1 and mutate x% of it.

Table 5.1 shows the main characteristics of our datasets. The compression ratio may not be very descriptive given that the sequences are highly repetitive. For this reason, we include the compression ratio achieved by Navarro's Re-Pair implementation. This does not include any post-processing, and just represents the original sequences, therefore, it is only a guideline on how much the text could be compressed.

We generated queries by taking a version uniformly at random, and then choosing a substring uniformly at random from that particular version.

The machine used for generating the indexes and measuring time has 2 Intel(R) Xeon(R) CPU X5660 processors running at 2.80GHz, 11TB of hard drive and 24GB of RAM. The machine is running Ubuntu Linux 11.04 with kernel `2.6.38-13-generic` for `x86_64`. All our code is implemented and `C++` and was compiled using `gcc` version 4.5.2 with flags `-O3 -DNDEBUG`. Our code is available for download from `http://fclaude.recoded.cl/projects`.

5.3.3 Full-Text Document Listing

Table 5.2 shows the sizes of our index for the different collections. We can see that our indexes, for the Wikipedia samples and the DNA synthetic data, are approximately 4 to 4.5 times the size of the collection when we compress it using Re-Pair. This means, within this space, we are replacing the collection and supporting search operations on top of it.

Table 5.3 shows the time in microseconds per element retrieved. This was averaged over 10,000 queries. Table 5.4 shows the time per query for our index.

Our index shows very good performance for the Wikipedia samples, but not as good for DNA. This is because the DNA grammar forces the algorithm to visit more symbols per element

Collection	T	Lists	SuffPerm	$\mathcal{R}$	Total	Compr.
Wiki1	0.39MB	0.49MB	0.39MB	0.39MB	1.66MB	2.43%
Wiki2	1.75MB	2.14MB	1.69MB	1.71MB	7.29MB	1.22%
Wiki3	3.19MB	4.37MB	3.12MB	3.06MB	13.73MB	0.90%
DNA1	3.21MB	4.76MB	2.94MB	3.03MB	13.95MB	1.40%
DNA2	1.99MB	2.80MB	1.78MB	1.91MB	8.47MB	0.85%
DNA3	1.26MB	1.59MB	1.15MB	1.23MB	5.23MB	0.52%

Table 5.2: Space required for our index for each dataset, separated by components.

Collection	$m = 4$	$m = 8$	$m = 16$	$m = 32$
Wiki1	0.60	1.36	3.37	7.38
Wiki2	0.51	0.72	1.72	4.03
Wiki3	0.54	0.83	2.40	6.23
DNA1	20.03	1.86	3.05	6.05
DNA2	12.42	1.35	2.17	4.06
DNA3	8.05	1.06	1.59	2.90

Table 5.3: Time per element retrieved in microseconds for patterns of length $m = 4, 8, 16, 32$, averaged over $10,000$ queries.

Collection	$m = 4$	$m = 8$	$m = 16$	$m = 32$
Wiki1	1.42	1.26	2.22	4.33
Wiki2	4.24	1.67	1.87	3.29
Wiki3	12.02	5.16	6.45	12.00
DNA1	20.12	1.86	3.04	6.04
DNA2	14.02	1.34	2.14	4.06
DNA3	9.19	1.05	1.59	2.91

Table 5.4: Time per query in milliseconds for patterns of length $m = 4, 8, 16, 32$, averaged over $10,000$ queries.

Collection	$m = 4$	$m = 8$	$m = 16$	$m = 32$
Wiki1	1.95	1.92	1.91	1.91
Wiki2	1.99	1.98	2.00	2.02
Wiki3	1.94	1.92	1.91	1.91
DNA1	26.86	2.50	1.99	1.98
DNA2	18.10	2.29	1.99	1.99
DNA3	12.83	2.17	2.00	1.99

Table 5.5: Elements visited per document reported by the index for different pattern lengths m. This is averaged over $10,000$ queries.

Collection	$m = 2$	$m = 4$	$m = 8$	$m = 16$	$m = 32$
Wiki1	0.22%	0.77%	1.57%	2.08%	2.68%
Wiki2	0.09%	0.45%	1.11%	1.87%	2.37%
Wiki3	0.05%	0.43%	1.16%	1.65%	2.00%

Table 5.6: Percentage of primary occurrences per occurrence in the collection for different pattern lengths m. This is averaged over 10,000 queries.

Dataset	Words	Length	Plain Size (MB)	Index (MB)	Compression
Wiki1	8,310	7,392,756	12.34	0.85	6.87%
Wiki2	20,983	68,771,392	122.97	4.14	3.37%
Wiki3	32,287	175,204,941	313.29	8.20	2.62%

Table 5.7: Space results for the word-based index.

reported. We summarize this in Table 5.5, where we count the number of symbols we visit in the grammar representation of the inverted list, and divide this by the size of the output.

Table 5.6 shows the average of the number of primary occurrences divided by the number of occurrences for each pattern on the Wikipedia samples. This shows that in fact, the number of primary occurrences that cover all occurrences of the pattern is quite small when compared to the total number of occurrences in the collection. This also backs up the intuition that shorter patterns, which tend to have more occurrences, are better captured by the grammar.

5.3.4 Word-Based Document Listing

To measure the word-based version of our index we considered phrases of lengths 2, 3, 4, 5, 6, 8 and 10. We used the same datasets from Wikipedia (Wiki1, Wiki2, and Wiki3), after taking all substrings composed only by alphabetic characters and applying the Snowball stemming algorithm [100]. The code and data are available at `http://fclaude.recoded.cl/projects/`. Table 5.7 shows the properties of this new dataset and the size of our index for each one of them.

The phrases were generated following the same procedure as for the symbol-based indexes. Table 5.8 shows the average time per element retrieved in the resulting set for 10,000 queries.

Dataset	$m = 2$	$m = 3$	$m = 4$	$m = 5$	$m = 6$	$m = 8$	$m = 10$
Wiki1	0.41	0.63	0.78	0.93	1.08	1.40	1.77
Wiki2	0.45	0.52	0.59	0.67	0.76	0.95	1.18
Wiki3	0.47	0.57	0.70	0.83	0.98	1.28	1.61

Table 5.8: Time per element retrieved in microseconds for different pattern lengths L.

Dataset	Document Array	Our index	Ratio
Wiki1	27MB	1.66MB	16.27
Wiki2-pref	223MB	3.58MB	62.29

Table 5.9: Sizes for the document array and our complete index. We also include the ration dividing the size of the document array by that of our index.

5.3.5 Comparison to related work

The solutions for repetitive collections focus on the more traditional sense, which is, building inverted lists for a fixed set of words, and allowing queries that only contain words from that set. The compression of those results are by far superior to traditional indexes. Some examples are the results by He et al. [61, 62] and the ones by Claude et al. [28]. They support conjunctive queries, yet they are not designed for searching arbitrary patterns or phrases.

The best result at the time of this writing is due to Navarro et al. [94]. It is hard to compare numbers with related work, since they are not designed to work on highly repetitive sequences. We created the indexes for the implementation due to Navarro et al. [94]. They only implement the construction of the so called document array, but this structure by itself is not enough to perform document listing. They require a (compact) suffix array or suffix trie in addition [92]. For this, the most natural choice is the CSA of Sadakane [104]. In our scenario, it is better to replace this by the Run-length CSA [109].

To compare our results, we actually ignore the space of the CSA or RLCSA, and only consider the document array. This structure by itself, requires enough space to argue that our space is in fact much better. Any structure depending on the document array using currently known methods requires at least the space shown in Table 5.9[9]. We compare the results against our complete index, that is, ours replaces the collection and allows searching, whereas the document array by itself does not provide any functionality.

When comparing query times, they report times close to ours, which allows us to conclude that we are competitive in time, while providing an index that is considerably smaller for repetitive collections.

5.4 Concluding Remarks

We presented a new index for representing highly repetitive collections. This index can be used in two different scenarios:

- Indexing a collection to support document listing of exact substrings.

[9]The implementation available cannot handle datasets bigger than 250MB, so we could only index Wiki1 and a prefix of Wiki2.

- Indexing a collection and support phrase searches for words existing in the collection.

The results show that while providing competitive time complexities, we achieve space considerably smaller than previous results. This opens an interesting new line for storing historic information on documents while supporting efficient search operations.

The algorithms developed for inverted lists play well with our index. In the symbol-based version, we can build the inverted index for any possible substring using our index. Furthermore, when we tokenize the text, and index the word identifiers, our index is just a grammar-compressed representation of the inverted lists, augmented with extra information to support phrase search operations on top of it, allowing to produce the inverted list of an arbitrary phrase.

Our work also leaves several open problems. First, the union of all non-terminals that represent primary occurrences has no good theoretical bound, yet is reasonable in practice. Is it possible to modify the structure or the grammar in order to provide a reasonable bound, say we do not visit more than k symbols per element in the resulting set?

Another interesting problem, not considered in this work, is whether we could support approximate searches, allowing to retrieve the phrases or substrings that are most similar to the query. This is interesting especially since typos may have a huge effect in the result.

Finally, one could explore ranking results by closeness. This may be achieved by first filtering using the parse tree of the grammar. Given two primary occurrences, the lowest common ancestor in the parse tree allows to estimate bounds on the distance between their occurrences.

6 Conclusions

In this thesis we have presented space efficient data structures for binary relations, text indexing, and document listing.

Our first result deals with constructing wavelet trees. This structure is used throughout the whole thesis, and is the main component in many of them. The result we obtain allows for fast and space-efficient construction of those.

Binary relation representations are a building block for most of the structures presented in this thesis, and also in general for many other problems in Computer Science. Our representations obtain the best time complexities for a wide set of operations within the space in which they are represented. It is still interesting to explore alternative representations that allow the same query times while obtaining space closer to the entropy. Another interesting problem is to obtain better measures on the size of binary relations that are somehow repetitive. A good example of this are Web graphs [26] and the inverted lists constructed in Chapter 5.

In Chapter 4, we present two grammar-based compressed indexes. One is the first compressed index aimed at such a wide class of compressors, SLPs in this case. The second index extends this result even further, allowing any grammar-compressed text as input, and also improves the dependency on the height of the grammar for searching.

Finally, in Chapter 5, we present a practical application of the index presented in Chapter 4 that allows to solve the document listing problem on highly repetitive or versioned collections. This result is a proof of concept that shows how the previous results can be used to solve problems in information retrieval.

The concluding remarks of each chapter discuss the more technical aspects of the problems left open. Below are some broader interesting open problems and potential directions to pursue:

- Dynamization: all the structures presented in this thesis are static, that is, once built, we cannot update the information again. This goes for the binary relations, text indexes and as a consequence, to the document listing index. It would certainly be useful and interesting to extend these results for the dynamic case, as this would allow to incrementally build indexes and also maintain them without having to rebuild everything from scratch after every change.

  There are dynamic solutions for wavelet trees, but this is not enough to extend it to binary relations, as the size of the alphabet may change, and that is harder to deal with. It is also hard to maintain and modify a grammar-compressed sequence, as the structure does not only add rules, but has to re-adjust them as the text or collection changes.

- Grammar-compressed indexes are an important part of this thesis. The chapters dealing with them do not quite solve the whole issue yet. There is still room for improvement in representations that are space-efficient and practical, that allow to extract and search in time that is independent from the height of the grammar.

- It is certainly fascinating to extend the index for document listing to perform top-k queries. The solution presented here allows to perform a pre-filtering for a given query, but does not rank the documents obtained. Including common measures to the index may be challenging, depending on whether we can generate the measure from partial results.

Much of the code required to build or use the structures presented in this thesis are available at http://libcds.recoded.cl/ [38]. This library was created, and has been maintained, by the author. There are still many interesting lines to pursue, particularly those related to use in variety of large scale applications. Most of the structures presented in this thesis have not been tested on large documents, and they require quite some work to become practical options in real applications. To be more concrete, the chapter on binary relations requires a thorough experimental study before using the structures in real data. The wavelet tree representation has proven to be efficient in terms of space and time in some cases (for example, Chapter 5). Indeed, it has been shown by Arroyuelo et al. [4] that in some applications wavelet trees are a viable option to classical inverted lists.

Bibliography

[1] W. Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99:118–133, 1928. ISSN 0025-5831. doi:10.1007/BF01459088. 77

[2] A. Apostolico and S. Lonardi. Some theory and practice of greedy off-line textual substitution. In *Proceedings of the 8th Data Compression Conference (DCC)*, pages 119–128. 1998. 67

[3] D. Arroyuelo, F. Claude, R. Dorrigiv, S. Durocher, M. He, A. López-Ortiz, J. I. Munro, P. K. Nicholson, A. Salinger, and M. Skala. Untangled monotonic chains and adaptive range search. *Theoretical Computer Science*, 412(32):4200–4211, 2011. 55, 57

[4] D. Arroyuelo, S. González, M. Marín, M. Oyarzún, and T. Suel. To index or not to index: time-space trade-offs in search engines with positional ranking functions. In *Proceedings of the 35th International ACM SIGIR conference on research and development in Information Retrieval (SIGIR)*, pages 255–264. ACM, 2012. 114

[5] D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 319–330. 2006. 11, 85, 86

[6] J. Barbay, L. C. Aleardi, M. He, and J. I. Munro. Succinct representation of labeled graphs. *Algorithmica*, 62(1-2):224–257, 2012. 31, 36, 37

[7] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 2013. To appear. 4, 5, 7, 12, 58

[8] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proceedings of the 21st Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 6507, pages 315–326. Springer, 2010. Part II. 4, 5, 12, 44, 58

[9] J. Barbay, A. Golynski, J. I. Munro, and S. S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theoretical Computer Science*, 387(3):284–297, 2007. xv, 7, 8, 31, 32, 36, 40, 60

[10] J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proceedings of the 18th Symposium on Discrete Algorithms (SODA)*, pages 680–689. 2007. xv, 7, 8, 31

[11] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 14, 2009. 36

[12] J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 111–122. Schloss Dagstuhl, Leibnitz Zentrum fuer Informatik, Germany, 2009. 7, 15, 55, 57, 58

[13] M. Beeler, R. Gosper, and R. Schroeppel. Hakmem. MIT Artificial Intelligence Lab Publications: AI Memos (1959 - 2004), February 1972. doi:http://hdl.handle.net/1721.1/6086. Http://hdl.handle.net/1721.1/6086. 25

[14] M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004. 79

[15] D. Benoit, E. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. 5, 6

[16] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. S. Rao, and O. Weimann. Random access to grammar-compressed strings. In *Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 373–389. 2011. 68, 95

[17] P. Boldi and S. Vigna. The WebGraph framework I: compression techniques. In *Proceedings of the 13th World Wide Web Conference (WWW)*, pages 595–602. 2004. 34

[18] P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *Proceedings of the 20th Symposium on Algorithms and Data Structures (WADS)*, pages 98–109. 2009. 15, 48, 64

[19] N. Brisaboa, S. Ladra, and G. Navarro. K2-trees for compact web graph representation. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 18–30. 2009. 31, 34

[20] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, 1994. 11

[21] S. Büttcher, C. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press, 2010. ISBN 0262026511, 9780262026512. 11

[22] 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. 67, 69, 97

[23] Y.-F. Chien, W.-K. Hon, R. Shah, and J. Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *Proceedings of the 18th Data Compression Conference (DCC)*, pages 252–261. 2008. 36

[24] D. Clark. *Compact Pat Trees*. Ph.D. thesis, University of Waterloo, 1996. 3, 4, 76

[25] D. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391. 1996. 6, 9, 15

[26] F. Claude. *Compressed Data Structures for Web Graphs*. Master's thesis, University of Chile, 2008. Advisor: Gonzalo Navarro. 31, 32, 113

[27] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proc. 10th IEEE Conference on Bioinformatics and Bioengineering (BIBE)*. 2010. 11, 68, 85, 98, 102

[28] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Indexes for highly repetitive document collections. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 463–468. 2011. 98, 102, 104, 110

[29] F. Claude and S. Ladra. Practical representations for web and social graphs. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management (CIKM)*, pages 1185–1190. 2011. 31

[30] F. Claude, J. I. Munro, and P. K. Nicholson. Range queries over untangled chains. In *Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 6393, pages 82–93. 2010. 58

[31] F. Claude and G. Navarro. A fast and compact Web graph representation. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 105–116. Springer, 2007. 98, 100

[32] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187. Springer, 2008. 15

[33] F. Claude and G. Navarro. Self-indexed text compression using straight-line programs. In *Proceedings of the 34th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, LNCS 5734, pages 235–246. 2009. 11

[34] F. Claude and G. Navarro. Extended compact Web graph representations. In T. Elomaa, H. Mannila, and P. Orponen, editors, *Algorithms and Applications (Ukkonen Festschrift)*, LNCS 6060, pages 77–91. Springer, 2010. 31

[35] F. Claude and G. Navarro. Fast and compact Web graph representations. *ACM Transactions on the Web (TWEB)*, 4(4):article 16, 2010. 31, 34

[36] F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2010. 31, 36, 40

[37] F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 180–192. 2012. 15, 17, 18

[38] F. Claude (Maintainer). Compact data structures library (Libcds). http://libcds.recoded.cl. 17, 114

[39] J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA)*, LNCS 6347, pages 194–205 (part II). 2010. 13

[40] D. Bentley et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, 2008. 85

[41] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000. 78

[42] A. Farzan. *Succinct Representation of Trees and Graphs*. Ph.D. thesis, University of Waterloo, 2009. 7

[43] A. Farzan, T. Gagie, and G. Navarro. Entropy-bounded representation of point grids. In *Proceedings of the 21st Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 6507, pages 327–338. 2010. Part II. 48, 102

[44] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithms*, 13, 2009. 30 pages. 15

[45] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–398. 2000. 11

[46] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of the 12th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 269–278. Philadelphia, PA, USA, 2001. 11

[47] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007. 4, 5, 11, 12, 16, 23, 32

[48] F. E. Fich, J. I. Munro, and P. V. Poblete. Permuting in place. *SIAM Journal on Computing*, 24:266, 1995. 20, 24

[49] J. Fischer. Optimal succinctness for range minimum queries. In *Proceedings of the 9th Symposium on Latin American Theoretical Informatics (LATIN)*, LNCS 6034, pages 158–169. 2010. 12, 49, 62, 78

[50] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426:25–41, 2012. 44, 45, 46, 48, 49, 52, 64

[51] L. Gasieniec, R. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Proceedings of the 15th Data Compression Conference (DCC)*, page 458. 2005. 90, 91, 92

[52] L. Gasieniec and I. Potapov. Time/space efficient compressed pattern matching. *Fundamenta Informaticae*, 56(1-2):137–154, 2003. 67

[53] A. Golynski. *Upper and Lower Bounds for Text Indexing Data Structures*. Ph.D. thesis, University of Waterloo, 2007. 3, 4

[54] A. Golynski, J. I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373. 2006. 4, 5, 12, 23, 44, 84

[55] A. Golynski, R. Raman, and S. Rao. On the redundancy of succinct data structures. In *Proceedings of the 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, LNCS 5124, pages 148–159. 2008. 54

[56] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: Pat trees and pat arrays. *Information retrieval*, pages 66–82, 1992. 9

[57] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227. 2007. 100

[58] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994. ISBN 0201558025. 5

[59] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850. 2003. 4, 5, 11, 12, 15, 23, 44

[60] R. Grossi, A. Orlandi, R. Raman, and S. S. Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. *CoRR*, abs/0902.2648, 2009. 3, 4

[61] J. He, H. Yan, and T. Suel. Compact full-text indexing of versioned document collections. In *Proceedings of the 18th ACM conference on Information and Knowledge Management (CIKM)*, pages 415–424. ACM, 2009. ISBN 978-1-60558-512-3. 110

[62] J. He, J. Zeng, and T. Suel. Improved index compression techniques for versioned document collections. In *Proceedings of the 19th ACM international conference on Information and Knowledge Management (CIKM)*, pages 1239–1248. ACM, 2010. 110

[63] M. He, J. I. Munro, and S. Rao. Succinct ordinal trees based on tree covering. In *Proceedings of the 34th International Colloquium Automata, Languages and Programming (ICALP)*, pages 509–520. Springer, 2007. 7

[64] W. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 713–722. IEEE, 2009. 12

[65] W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partial sums. In *Proceedings of the 14th Annual International Symposium on Algorithms and Computation (ISAAC)*, pages 505–516. 2003. 79

[66] S. Inenaga and H. Bannai. Finding characteristic substrings from compressed texts. In *Proceedings of the 14th Prague Stringology Conference (PSC)*, pages 40–54. Czech Technical University in Prague, 2009. 82

[67] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. 1989. 3, 4, 5, 6, 15

[68] J. Kärkkäinen. *Repetition-Based Text Indexing*. Ph.D. thesis, U. Helsinki, Finland, 1999. 36, 93

[69] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages, and Programming (ICALP)*, LNCS 2719, pages 943–955. 2003. 78

[70] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proceedings of the 3rd South American Workshop on String Processing (WSP)*, pages 141–155. Carleton University Press, 1996. 73

[71] M. Karpinski, W. Rytter, and A. Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nordic Journal of Computing*, 4(2):172–186, 1997. 68

[72] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 2089, pages 181–192. 2001. 78

[73] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003. 67

[74] J. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000. 67

[75] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977. 81

[76] R. Konow and G. Navarro. Faster compact top-k document retrieval. In *Proceedings of the 23rd Data Compression Conference (DCC)*. 2013. To appear. 13

[77] S. Kreft and G. Navarro. Self-indexing based on LZ77. In *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 6661, pages 41–54. 2011. 15, 45, 48, 64

[78] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000. 67, 82, 85, 97, 100

[79] E. Lehman and A. Shelat. Approximation algorithms for grammar-based compression. In *Proceedings of the 13th annual ACM-SIAM symposium on Discrete algorithms*, pages 205–212. Society for Industrial and Applied Mathematics, 2002. 67

[80] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387:332–347, 2007. Special issue on The Burrows-Wheeler Transform and its Applications. 7, 11, 36, 44

[81] V. Mäkinen and N. Välimäki. Constructing wavelet trees. Personal Communication. 20

[82] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 319–327. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1990. ISBN 0-89871-251-3. 9

[83] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993. 78

[84] D. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968. 76, 94

[85] J. I. Munro. Tables. In *Proceedings of the 16th Conference of Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume LNCS 1180, pages 37–42. 1996. 2, 3, 4, 23

[86] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 345–356. Springer, 2003. 7, 73, 91, 92

[87] J. I. Munro and S. S. Rao. Succinct representations of functions. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1006–1015. 2004. 6

[88] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc 13th Annual Symposium on Discrete Algorithms (SODA)*, pages 657–666. 2002. 12, 50

[89] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004. 11, 85, 86, 96

[90] G. Navarro. Indexing highly repetitive collections. In *Proceedings of the 23rd International Workshop on Combinatorial Algorithms (IWOCA)*, LNCS 7643, pages 274–279. 2012. 97

[91] G. Navarro. Wavelet trees for all. In *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7354, pages 2–26. 2012. Invited paper. 4

[92] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007. 10, 11, 15, 63, 110

[93] G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1066–1078. 2012. 13

[94] G. Navarro, S. Puglisi, and D. Valenzuela. Practical compressed document retrieval. In *Proceedings of the 10th international Conference on Experimental Algorithms (SEA)*, LNCS 6630, pages 193–205. 2011. 13, 110

[95] G. Navarro and D. Valenzuela. Space-efficient top-k document retrieval. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA)*, LNCS 7276, pages 307–319. 2012. 13

[96] C. Nevill-Manning, I. Witten, and D. Maulsby. Compression by induction of hierarchical grammars. In *Proceedings of the 4th Data Compression Conference (DCC)*, pages 244–253. 1994. 67, 82

[97] C. G. Nevill-Manning. *Inferring Sequential Structure*. Ph.D. thesis, 1996. 67

[98] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2007. 3, 4, 57

[99] M. Pătraşcu. Succincter. In *Proceeding of the 49th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313. 2008. 3, 4, 57, 61

[100] M. F. Porter. An Algorithm for Suffix Stripping. *Program*, 14(3):130–137, 1980. 109

[101] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242. 2002. ISBN 0-89871-513-X. 3, 4, 11, 15, 23, 29, 57, 86

[102] L. Russo and A. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Information Retrieval*, 11(4):359–388, 2008. 85

[103] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003. 67, 68, 69

[104] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceedings of the 11th International Conference on Algorithms and Computation (ISAAC)*, pages 410–421. Springer-Verlag, 2000. 10, 13, 110

[105] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003. 79

[106] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007. 12, 79

[107] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149. 2010. 6

[108] H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3:416–430, 2005. 85

[109] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 164–175. 2008. 85, 110

[110] B. Stinson. *The Bro Code*. A Fireside book. Simon & Schuster, 2008. ISBN 9781439110003.

[111] J. Storer and T. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982. 67

[112] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 205–215. 2007. 12, 102

[113] K. Wagner. Monotonic coverings of finite sets. *Elektron. Informationsverarb. Kybernet.*, 20:633–639, 1984. 55

[114] R. Wan. *Browsing and searching compressed documents*. Ph.D. thesis, The University of Melbourne, 2003. 102

[115] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11. 1973. 9

[116] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, 2nd edition, 1999. 31, 36

[117] B. Yang, J. Chen, E. Lu, and S. Q. Zheng. A comparative study of efficient algorithms for partitioning a sequence into monotone subsequences. In *Proceedings of the 4th International Conference on Theory and Applications of Models of Computation (TAMC)*, volume 4484 of *LNCS*, pages 46–57. Springer, 2007. ISBN 978-3-540-72503-9. 55

[118] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. 67

[119] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978. 67, 77, 80, 82, 85, 97