Succinct Indexes

by

Meng He

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, 2007

©Meng He, 2007

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.

Meng He

Abstract

This thesis defines and designs succinct indexes for several abstract data types (ADTs). The concept is to design auxiliary data structures that ideally occupy asymptotically less space than the information-theoretic lower bound on the space required to encode the given data, and support an extended set of operations using the basic operators defined in the ADT. As opposed to succinct (integrated data/index) encodings, the main advantage of succinct indexes is that we make assumptions only on the ADT through which the main data is accessed, rather than the way in which the data is encoded. This allows more freedom in the encoding of the main data. In this thesis, we present succinct indexes for various data types, namely strings, binary relations, multi-labeled trees and multi-labeled graphs, as well as succinct text indexes. For strings, binary relations and multi-labeled trees, when the operators in the ADTs are supported in constant time, our results are comparable to previous results, while allowing more flexibility in the encoding of the given data.

Using our techniques, we improve several previous results. We design succinct representations for strings and binary relations that are more compact than previous results, while supporting access/rank/select operations efficiently. Our high-order entropy compressed text index provides more efficient support for searches than previous results that occupy essentially the same amount of space. Our succinct representation for labeled trees supports more operations than previous results do. We also design the first succinct representations of labeled graphs.

To design succinct indexes, we also have some preliminary results on succinct data structure design. We present a theorem that characterizes a permutation as a suffix array, based on which we design succinct text indexes. We design a succinct representation of ordinal trees that supports all the navigational operations supported by various succinct tree representations. In addition, this representation also supports two other encodings schemes of ordinal trees as abstract data types. Finally, we design succinct representations of planar triangulations and planar graphs which support the rank/select of edges in counter clockwise order in addition to other operations supported in previous work, and a succinct representation of k-page graph which supports more efficient navigation than previous results for large values of k.

Acknowledgements

I would like to first thank my supervisor, Professor J. Ian Munro, for his guidance, support, encouragement and patience throughout my graduate program. Ian has supervised both my master's and doctoral studies, and working with Ian has been great experience for me. Not only has he taught me a great deal about research, about how to discover research problems and how to solve them, he has also offered me numerous advices on various aspects of academic life, such as academic writing and presentations. His friendliness and humor have made the collaboration with him even more enjoyable. For this thesis, he has spent a lot of time on proofreading it and and his comments have greatly improved its quality.

Many thanks to the members of my committee, Dr. Guy Blelloch, Dr. L. Bruce Richmond, Dr. Alejandro López-Ortiz and Dr. Jérémy Barbay for their time. They have given me valuable comments to further improve the presentation of my thesis. Thanks to Wendy Rush for helping me submit my thesis for review (when the final draft of this thesis was ready for the committee, I had already moved to Ottawa to start my postdoctoral job in Carleton University). I would also like to thank my collaborators during my research work at University of Waterloo (besides Ian and Jeremy), including Dr. S. Srinivasa Rao and Dr. Luca Castelli Aleardi, for providing me with new problems to work on and teaching me new problem-solving techniques.

During the several years that I have spent at Waterloo, there are many people and friends that have made my life fun and exciting. Special thanks to Marilyn Miller, Ian's wife, for her kindness and her help. She helped me a lot to make preparations for my wedding, which took place shortly before I started writing this thesis. Thanks for Zheng Qin for letting us stay in her apartment when my wife and I went to Waterloo for my thesis defense.

Among my family, I would like to thank my wife, Wenhan Yu, for her love and her care when I was busy writing my thesis. Although she had her own studies and exams, she spent much more time on housework than I did when I was writing. This had been great help. Last but not least, I would like to thank my father, Taiyi He, and my mother, Jianxin Xu, for their support and encouragement over the years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The rapid growth of large sets of text and the need for efficient searches of these sets, have led to a trend of succinct representation of text indexes as well as the text itself. *Succinct data structures* were first proposed by Jacobson [55] to encode bit vectors, (unlabeled) trees and planar graphs in space close to the information-theoretic lower bound, while supporting efficient navigational operations. This technique was successfully applied to various other abstract data types (ADTs), such as dictionaries, strings, binary relations [5] and labeled trees [36, 5]. In addition, succinct data structures have been proved to be very useful in practice. For example, Delpratt *et al.* [23] engineered the implementation of succinct trees and reported that their structure uses 3.12 to 3.81 bits per node to encode the structure of XML trees that have 57K to 160M nodes. Such space cost is merely a small percentage of that of an explicit, pointer-based tree representation.

In most of the previous work, researchers encode the given data (or assume that the data is encoded) in a specific format, and further construct auxiliary data structures on it. They then use both the encoded data and the auxiliary data structures to support various operations, e.g. [44, 36, 34, 52, 5]. Usually in this type of design, the auxiliary data structures do not work if the given data is encoded in a different format, and therefore, the encoding of the given data and the design of the auxiliary data structures are inseparable. We thus call this type of design *succinct integrated encodings* of data structures.

A different line of research concentrates on reducing the size of the traditional text indexes to allow fast text retrieval, without transforming the text (i.e. the given data) to store it in specific formats. Therefore, in such research work, the representation of the text indexes and the encodings of the text itself can be designed separately. For example, Clark and Munro [21] designed a compact PAT tree which takes much less space than the standard representation of a suffix tree, and used it to facilitate text retrieval.

The concept of separating the index and the given data was also used to prove the lower bounds [24, 66, 39] and to analyze the upper bounds [80] on the space required to encode some data structures: it limits the definition of the encoding to the index. For example, Demaine and López-Ortiz [24] proved that any text index supporting pattern search in time linear in the length of the pattern requires roughly the same amount of space as the text itself. Miltersen [66] proved a lower bound of the size of any index supporting rank/select operations on bit vectors, and Golynski [39] further improved his results. Sadakane and Grossi [80] analyzed the space cost of their data structure by proving that the auxiliary data structures occupy asymptotically less space than the given data.

In this thesis, we formulate the distinction between the index and the raw data, and apply it to the design of succinct data structures. Given an ADT, our goal is to design auxiliary data structures (i.e. *succinct indexes*) that ideally occupy asymptotically less space than the information-theoretic lower bound on the space required to encode the given data, and support an extended set of operations using the basic operators defined in the ADT. Succinct indexes and succinct integrated encodings are closely related, but they are different concepts: succinct indexes make assumptions only on the ADT through which the given data is accessed, while succinct integrated encodings represent data in specific formats. Succinct indexes are also more difficult to design: raw data plus a succinct index is a succinct integrated encoding, but the converse is not true.

Although the concept of succinct indexes was previously followed mainly to design space efficient text indexes, and was also presented as a technical restriction to prove lower/upper bounds, we argue that in fact succinct indexes are more appropriate to the design of a library of succinct tools for multiple usages than succinct integrated encodings, and that they are even directly required in certain applications. Some of the advantages of succinct indexes over succinct integrated encodings are:

1. A succinct integrated encoding requires the given data to be stored in a specific format. However, a succinct index applies to any encoding of the given data that

supports the required ADT. Thus when using succinct indexes, the given data can be either stored to achieve maximal compression or to achieve optimal support of the operations defined in the ADT.

2. The existence of two succinct integrated encodings supporting different operations over the same data type does not imply the existence of a single encoding supporting the union of the two sets of operations without storing the given data twice, because they may not store it in the same format. However, we can always combine two different succinct indexes for the same ADT to yield a single succinct index that supports the union of the two corresponding sets of operations in a straightforward manner.

3. In some cases, we need not store the data explicitly because it can be derived from some other information in a manner that efficiently supports the operations defined in the ADT. Hence a succinct index is the only additional memory cost.

In this thesis, we design succinct indexes for strings, binary relations, multi-labeled trees and multi-labeled graphs, as well as succinct text indexes. We then apply these techniques to various research problems, including the design of succinct integrated encodings to achieve maximum compression. In order to achieve these results, we also design succinct representations of unlabeled trees and graphs.

1.1 Organization of the Thesis

The thesis is organized as follows.

Chapter 2 deals with the background knowledge of the research area. First we present the notation and the word RAM machine model used throughout the thesis. We then introduce the ADT bit vector including the operations on it and the prior results achieving efficient implementation. This is a key structure for many succinct data structures and the results in our paper. We also introduce the information-theoretic lower bound and the notion of entropy, which are used to measure the space efficiency of our data structures.

Chapter 3 focuses on the design of the succinct indexes for strings and binary relations. We also apply these indexes to design high-order entropy-compressed succinct integrated encodings for strings, and a succinct integrated encodings for binary relations using essentially the information-theoretic minimum space. This chapter is based on part of the joint work with Jérémy Barbay, J. Ian Munro and S. Srinivasa Rao [6].

In Chapter 4, we design succinct index structures for a text string to support efficient pattern searching. Motivated by the fact that the standard representation of suffix arrays uses more space than the theoretical minimum, we present a theorem that characterizes a permutation as the suffix array of a binary string, and design a succinct representation of suffix arrays of binary strings based on the theorem. We also generalize our results to text strings drawn from larger alphabets, and apply the succinct indexes for strings to design high-order entropy compressed text indexes. Most of this chapter is based on the joint work with J. Ian Munro and S. Srinivasa Rao [52]. Section 4.7.3 is based on part of the joint work with Jérémy Barbay, J. Ian Munro and S. Srinivasa Rao [6].

In Chapter 5, we first design a succinct integrated encoding of ordinal trees that supports all the navigational operations independently supported by various succinct tree representations. We also show that our method supports two other encoding schemes of ordinal trees as abstract data types. We then apply the succinct indexes for binary relations to design succinct indexes for multi-labeled trees. This chapter is based on the joint work with J. Ian Munro and S. Srinivasa Rao [53], and part of the joint work with Jérémy Barbay, J. Ian Munro and S. Srinivasa Rao [6].

In Chapter 6, we use the previous results in designing succinct indexes for vertex labeled planar triangulations. We also apply succinct indexes for binary relations to design succinct representations of edge labeled planar graphs and the more general k-book embedded graphs. To achieve these results, we also improve some of the previous results on the succinct representation of unlabeled graphs. This chapter is based on the joint work with Jérémy Barbay, Luca Castelli Aleardi and J. Ian Munro [4].

Chapter 7 provides a brief summary, conclusions and some suggestions for future work. Appendix A lists the definitions of most of the terms introduced in this thesis.

Chapter 2

Preliminaries

This chapter introduces some concepts, results and definitions used throughout this thesis.

2.1 Notation

We use n to denote the sizes of various problems. For example, when we consider a string, we use n to denote the length of the string. When we consider a tree, we use n to denote the number of the nodes in the tree. We define n for other problems in later chapters.

We use $\log_2 x$ to denote the logarithm base 2 and $\lg x$ to denote $\lceil \log_2 x \rceil$. Occasionally this will matter.

We use $[i]$ to denote the set $\{1, 2, ..., i\}$.

2.2 Machine Model

The word RAM model is a popular variation of the classic *random-access machine* (*RAM*) model of Cook and Reckhow [22]. A RAM consists of an infinite number of memory cells with addresses $0, 1, 2, ...$, and a processor operating on them. The instruction set contains instructions commonly available in real computers, including arithmetic operations, computing the addresses of memory registers, data movement between registers, and control (subroutine calls, branch, etc.). The execution of each instruction takes constant time.

A word RAM is defined as a unit-cost random-access machine with word size w bits, for some w, and its instruction set is similar to that found in present-day computers [49]. It differs from the standard RAM mainly in the assumption that the contents of all the memory cells are integers in the range $\{0, ..., 2^w - 1\}$, which allows some new instructions that are natural on integers represented as strings of w bits. This model of computation has become very popular in a broad range of algorithms and data structures that deal with integer data and the structural information of combinatorial objects [49, 50, 30, 17, 36, 5, 80].

As with the RAM, there is no consensus on the set of arithmetic instructions available on a word RAM, and researchers have been using various sets of arithmetic instructions that are common in modern computers. In this thesis, we assume that these arithmetic instructions include integer addition, subtraction, multiplication and division, left and right shifts, and the bitwise Boolean operations (AND, OR and NOT). If we are dealing with trees on n nodes or strings of length n, we usually assume that the word size is $\Theta(\lg n)$ bits, i.e. $w = \Theta(\lg n)$. When all the n elements (for example, nodes or characters) are stored sequentially in a certain order, $\Theta(\lg n)$ is the minimum length of a word that allows the address of an element to be stored using a constant number of memory cells. Thus this is a common assumption. Hence we call our machine model a word RAM with word size $\Theta(\lg n)$ bits.

As can be seen above, we only make common assumptions in the machine model we use, and thus this model is a general model, adopted by a wide range of research work [30, 45, 17, 73, 36, 5, 80, 56].

2.3 Bit Vectors

A key structure for many succinct data structures, and for the research work in this thesis, is a bit vector B of length n that supports *rank* and *select* operations. We assume that the positions in B are numbered $1, 2, \ldots, n$. For $\alpha \in \{0, 1\}$, we consider the following operations:

- $\texttt{bin\_rank}_B(\alpha, x)$, the number of occurrences of α in $B[1 \ldots x]$;

- $\texttt{bin\_select}_B(\alpha, r)$, the position of the r^{th} α in B.

We omit the subscript B when it is clear from the context. Lemma 2.1 addresses the problem of succinctly representing bit vectors, in which part (a) is from Jacobson [55] and Clark and Munro [21], while part (b) is from Raman *et al.* [75].

Lemma 2.1. *A bit vector B of length n with v 1s can be represented using either: (a) $n+o(n)$ bits, or (b) $\lg \binom{n}{v} + O(n \lg\lg n / \lg n)$ bits, to support the access to each bit,* `bin_rank` *and* `bin_select` *in $O(1)$ time.*

A less powerful version of $\texttt{bin\_rank}(1, x)$, denoted $\texttt{bin\_rank}'(1, x)$, returns the number of 1s in $B[1 \ldots x]$ in the restricted case where $B[x] = 1$.

Lemma 2.2 ([75])**.** *A bit vector B of length n with v 1s can be represented using $\lg \binom{n}{v} + o(v) + O(\lg\lg n)$ bits to support the access to each bit, $\texttt{bin\_rank}'(1, x)$ and $\texttt{bin\_select}(1, r)$ in $O(1)$ time.*

2.4 Information-Theoretic Lower Bound and Entropy

There are several ways of measuring the space efficiency of succinct data structures (i.e. *succinctness*). The two most common approaches are to compare the space cost of a succinct data structure with the information-theoretic lower bound of representing the corresponding combinatorial object, and with the entropy of the corresponding combinatorial object, respectively.

Jacobson [55] initially measured the succinctness of a data structure by comparing its space cost to the information-theoretic minimum. Given a combinatorial object of n elements, the information-theoretic lower bound of representing it is $\lg u$ bits, where u is the number of different such combinatorial objects of size n. For example, there are 2^n different bit vectors of length n, thus the information-theoretic minimum of representing a bit vector of length n is $\lg(2^n) = n$ bits. Given a string of length n over alphabet $[\sigma]$, the information-theoretic minimum is $\lceil n \log_2 \sigma \rceil$ bits, as there are σ^n such strings.

Entropy is well-defined for strings, and has been extensively used to measure the text compression algorithms.

Definition 2.1. *The* **zeroth order empirical entropy** *of a string S of length n over alphabet $[\sigma]$ is*

$$H_0(S) = \sum_{\alpha=1}^{\sigma}(p_\alpha \log_2 \frac{1}{p_\alpha}) = -\sum_{\alpha=1}^{\sigma}(p_\alpha \log_2 p_\alpha),$$

where p_α is the frequency of the occurrence of character α, and $0 \log_2 0$ is interpreted as 0.

An ideal compressor that uses $\log_2 \frac{1}{p_\alpha}$ (or $-\log_2 p_\alpha$) bits to code the character α can compress the string S to $nH_0(S)$ bits. This is the maximum compression we can achieve using a uniquely decodable coding scheme in which each character is assigned a fixed code word. In the worst case, when the characters of S occur at the same frequency, we have $nH_0(S) = -n\log_2(\frac{1}{\sigma}) = n\log_2 \sigma$, which is the information-theoretic lower bound. Thus $nH_0(S) \leq n \log_2 \sigma$.

Definition 2.2. *Consider a string S of length n over alphabet $[\sigma]$. Given another string $w \in [\sigma]^k$, we define the string w_S to be a concatenation of all the single characters immediately following one of the occurrences of w in S. Then the* **$\mathbf{k}^{\mathbf{th}}$ order empirical entropy** *of S is*

$$H_k(T) = \frac{1}{|S|} \sum_{w \in [\sigma]^k} |w_S| H_0(w_S).$$

To illustrate the string w_S in Definition 2.2, consider the string $S = aabacabbabc$. For example, if $w = ab$, then $w_S = abc$.

$nH_k(S)$ is a lower bound in bits on the compression we can achieve using for each character a code that depends on only the k characters preceding it.

The notion of entropy has also been used to measure the succinctness of data structures supporting operations on strings [44, 63, 80, 6]. As the entropy for most other combinatorial objects such as binary relations and graphs are not well-defined, it is less common to use it to measure the succinctness for those data types. However, there are still definitions of the entropies of some combinatorial objects that are not strings. For example, Ferragina *et al.* [29] defined the entropy of labeled trees.

We use both methods to measure the succinctness of the data structures in this thesis.

Chapter 3

Strings and Binary Relations

This chapter deals with the problem of designing succinct indexes for strings and binary relations, two basic data types with applications to many research problems, including several described in later chapters. The chapter starts with a brief introduction in Section 3.1, followed by a brief review of previous work in Section 3.2, and a summary of the existing results we use in Section 3.3. In Sections 3.4 and Section 3.5, we design succinct indexes for strings and binary relations, respectively. Section 3.6 presents two applications of these results. Section 3.7 gives some conclusion remarks and suggestions for future work.

3.1 Introduction

The first data structure we consider is a string structure which supports efficient rank/select operations. The rank/select operations on bit vectors in Section 2.3 can be generalized to a string (or a sequence) S of length n over alphabet $[\sigma]$, with the operations:

- $\texttt{string\_rank}_S(\alpha, x)$, the number of occurrences of character α in $S[1..x]$;
- $\texttt{string\_select}_S(\alpha, r)$, the position of the r^{th} occurrence of character α in the string;
- $\texttt{string\_access}_S(x)$, the character at position x in the string.

We omit the subscript S when it is clear from the context.

These operations can be further generalized to binary relations. Consider a binary relation R between a set of objects, $[n]$, and a set of labels, $[\sigma]$, under which each object can be associated with zero or more labels. We use t to denote the number of object-label pairs, and thus R can be treated as t pairs from $[n]\times[\sigma]$. We consider the following operations:

- $\texttt{label\_rank}_R(\alpha, x)$, the number of objects labeled α up to (and including) x;
- $\texttt{label\_select}_R(\alpha, r)$, the position of the r^{th} object labeled α;
- $\texttt{label\_access}_R(x, \alpha)$, whether object x is associated with label α.

We omit the subscript R when it is clear from the context.

The above operations on strings and binary relations have a number of applications [44, 34, 52, 41, 5], and hence supporting them efficiently is a fundamental problem in the design of succinct data structures. We thus design succinct indexes for strings and binary relations to support these operations.

We introduce succinct indexes in two steps: we first define the ADTs and then design succinct indexes for these ADTs.

3.2 Previous Work

3.2.1 Strings

Grossi *et al.* [44] first generalized the `bin_rank` and `bin_select` operators to strings during their research on designing compressed suffix arrays. They originally stored a set of bit vectors and used the straightforward approach of encoding the bit vectors separately. However, this used more space than the information-theoretic minimum. Based on the fact that these bit vectors actually constitute a string (i.e. there is one and only one bit vector that has a 1 at any given location), they designed a data structure called *wavelet tree* to combine the bit vectors. A wavelet tree can be used to encode a string using $nH_0+o(n)\cdot\lg\sigma$ bits to support `string_access`, `string_rank` and `string_select` in $O(\lg\sigma)$ time.

To design a succinct integrated encoding for strings over large alphabets, Golynski *et al.* [41] gave another encoding that uses $n\,(\lg\sigma + o(\lg\sigma))$ bits and supports `string_rank` and `string_access` in $O(\lg\lg\sigma)$ time, and `string_select` in constant time. The $\lg\lg\sigma$ factor in the above running time is more scalable for large alphabets than the $\lg\sigma$ factor of wavelet trees. However, their encoding is not easily compressible.

3.2.2 Binary Relations

Based on a reduction from the support of rank/select on binary relations to that on strings, Barbay *et al.* [5] proposed an encoding of binary relations using $t(\lg\sigma + o(\lg\sigma))$ bits to support the operators `label_rank` and `label_access` in $O(\lg\lg\sigma)$ time, and `label_select` in constant time. They also considered the following operations:

- `label_nb`(α), the number of objects associated with label α;
- `object_rank`(x, α), the number of labels associated with object x preceding and including label α;
- `object_select`(x, r), the r^{th} label associated with object x;
- `object_nb`(x), the number of labels associated with object x.

Their encoding supports the operators `label_nb` and `object_nb` in $O(1)$ time, `object_rank` in $O((\lg\lg\sigma)^2)$ time, and `object_select` in $O(\lg\lg\sigma)$ time. They have an alternative encoding using also $t(\lg\sigma + o(\lg\sigma))$ bits that supports `label_nb`, `object_select` and `object_nb` in $O(1)$ time, `label_select`, `object_rank` and `label_access` in $O(\lg\lg\sigma)$ time, and `label_rank` in $O(\lg\lg\sigma\lg\lg\lg\sigma)$ time.

3.3 Preliminaries

3.3.1 Permutation

One important data structure we use in this chapter is a succinct representation of a permutations on $[n]$ that supports the efficient computations of the permutation and its inverse. It is fairly straightforward to represent a permutation π to support π and π^{-1} in $O(s)$ time for any parameter $s > 0$. We simply give the forward permutation and an auxiliary structure that gives for every s^{th} position in every cycle of length greater than s, the element s positions earlier in that cycle. Munro *et al.* [69] investigated this problem and trimmed the space required to $(1 + 1/s)n \log_2 n + O(n \lg\lg n / \lg n)$ bits.

To achieve this result, they explicitly encode the sequence $\pi(1), \pi(2), ..., \pi(n)$ in $n \log_2 n + o(n)$ bits, but only use the operator $\pi()$ to access the given data. Thus, this result can be rewritten in the form of designing succinct indexes:

Lemma 3.1 ([69]). *Given support for $\pi()$ (or $\pi^{-1}()$) in $g(n)$ time on a permutation on $[n]$, there is a succinct index using $n \lg n/s + O(n \lg\lg n/\lg n)$ bits that supports $\pi^{-1}()$ (or $\pi()$) in $O(s \cdot g(n))$ time for any parameter $s > 0$.*

3.3.2 y-fast Trie

Another important data structure we use is a y-fast trie, proposed by Willard [85] to encode a set E that consists of v distinct integers in the universe $[n]$ in $O(v \lg n)$ bits. It is an improvement upon the *stratified tree* proposed by Van Emde Boas *et al.* [26]. Given an integer x, the y-fast trie can be used to retrieve the largest integer in the set E that is less than or equal to x in $O(\lg\lg n)$ time.

If we treat the universe $[n]$ as a bit vector B of length n, and the v integers in the set as the positions of the 1s in B, the y-fast trie can be used to encode B in $O(v \lg n)$ bits and support the retrieval of the position of the last 1 in $B[1..x]$ in $O(\lg\lg n)$ time. As the integers in the set E are stored in the leaves of a y-fast trie, if we store their ranks explicitly in the leaf nodes, we can augment the y-fast trie to support $\texttt{bin\_rank}_B(1, x)$ in $O(\lg\lg n)$ time using additional $v \lg n$ bits. More precisely, to compute $\texttt{bin\_rank}_B(1, x)$, we first locate the last 1 in $B[1..x]$ using the y-fast trie in $O(\lg\lg n)$ time, and then retrieve the rank stored in the corresponding leaf of the y-fast trie in constant time. Thus:

Lemma 3.2 ([85]). *A bit vector B of length n with v 1s can be encoded using $O(v \lg n)$ bits to support $\texttt{bin\_rank}_B(1, x)$ in $O(\lg\lg n)$ time.*

3.3.3 Squeezing Strings into Entropy Bounds

Sadakane and Grossi [80] investigated the problem of encoding a string in its compressed form, while at the same time allowing efficient access to the string. Their main result is described in the following lemma:

Lemma 3.3 ([80]). *A string $S \in [\sigma]^n$ can be encoded using $nH_k(S) + O(\frac{n}{\log_\sigma n}(k \lg \sigma + \lg \lg n))$ bits*[1]*. When $k = o(\log_\sigma n)$, the above space cost is $nH_k(S) + \lg \sigma \cdot o(n)$ bits. This encoding can be used to retrieve any $O(\lg n)$ consecutive bits of the binary encoding of the string in $O(1)$ time.*

3.3.4 The Information-Theoretic Lower Bound of Representing Binary Relations

We showed in Section 2.4 that the information-theoretic lower bound of representing a string $S \in [\sigma]^n$ is $\lceil n \log_2 \sigma \rceil$ bits. Here we compute the information-theoretic lower bound of the representation of a binary relation.

To compute the number of distinct binary relations formed by t pairs from an object set $[n]$ and a label set $[\sigma]$, we observe that the set of these t pairs is a subset of the set $[n] \times [\sigma]$. Hence there are $\binom{n\sigma}{t}$ such binary relations. Thus the information-theoretic lower bound of representing a binary relation is $\lg \binom{n\sigma}{t}$ bits.

Barbay *et al.* [5] showed that when the average number of labels associated with each object is small (more precisely, if $t/n = \sigma^{o(1)}$), the above lower bound is $t(\lg \sigma - o(\lg \sigma))$ bits.

3.4 Strings

3.4.1 Definitions

We first design succinct indexes for a given string S of length n over alphabet $[\sigma]$. We adopt the common assumption that $\sigma \leq n$ (otherwise, we can reduce the alphabet size to the number of characters that occur in the string). We define the ADT of a string through the `string_access` operator that returns the character at any given position of the string.

[1]González and Navarro [43] noted that the term $(k \lg \sigma + \lg \lg n)$ appears erroneously as $(k + \lg \log_\sigma n)$ in [80]. Therefore, we use the correct formula in this chapter.

To generalize the operators on strings defined in Section 3.1 to include "negative" searches, we define a *literal* as either a character, $\alpha \in [\sigma]$, or its negation, $\bar{\alpha} \in [\sigma] - \{\alpha\}$ as follows (we use the array notation for strings to refer to its characters and substrings):

Definition 3.1. *Consider a string $S[1 \dots n]$ over the alphabet $[\sigma]$. A position $x \in [n]$* **matches literal** *$\alpha \in [\sigma]$ if $S[x] = \alpha$. A position $x \in [n]$* **matches literal** *$\bar{\alpha}$ if $S[x] \neq \alpha$. For simplicity, we define $[\bar{\sigma}]$ to be the set $\{\overline{1}, \dots, \overline{\sigma}\}$.*

With this definition, we can use `string_rank` and `string_select` to perform negative searches. For example, given the string *bbaaacdd*, we have that `string_rank`$(\overline{a}, 7) = 4$, as there are 4 characters that are not a in the string up to position 7. We also have `string_select`$(\overline{a}, 3) = 6$, as position 6 is the 3[rd] position whose character is not a.

We also consider the following operations on strings in addition to the three primary operations introduced in Section 3.1:

Definition 3.2. *Consider a string $S \in [\sigma]^n$, a literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ and a position $x \in [n]$ in S. The* **$\boldsymbol{\alpha}$-predecessor** *of position x, denoted by* `string_pred`(α, x)*, is the last position matching α before (and not including) position x, if it exists. Similarly, the* **$\boldsymbol{\alpha}$-successor** *of position x, denoted by* `string_succ`(α, x)*, is the first position matching α after (and not including) position x, if it exists.*

To illustrate the above two operations, consider the string *bbaaacdd*. We have that `string_pred`$(a, 7) = 5$, as position 5 is the last position in the string before position 7 whose character is a. We also have `string_pred`$(\overline{a}, 5) = 2$, as position 2 is the last position before position 5 whose character is not a. By allowing α to be possibly a literal in the set $[\bar{\sigma}]$, the α-predecessor/successor queries in fact generalize the colored predecessor/successor queries defined by Mortensen [67, 68].

3.4.2 Supporting Rank and Select

We now design a succinct index to support rank/select operations on strings. We have the following result.

Lemma 3.4. *Given support for* `string_access` *in $f(n, \sigma)$ time on a string $S \in [\sigma]^n$, there is a succinct index using $n \cdot o(\lg \sigma)$ bits that supports* `string_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$*

$$
\begin{array}{ll}
\texttt{S =} & \left[\begin{array}{cccc|cccc|cccc|cccc}
\texttt{a} & \texttt{b} & \texttt{a} & \texttt{a} & \texttt{d} & \texttt{c} & \texttt{b} & \texttt{d} & \texttt{b} & \texttt{c} & \texttt{a} & \texttt{a} & \texttt{a} & \texttt{d} & \texttt{b} & \texttt{b}
\end{array}\right] \\
\texttt{E =} & \left(\begin{array}{cccc|cccc|cccc|cccc}
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}\right) \\
 & \quad 1^{\text{st}}\ \texttt{chunk} \quad 2^{\text{nd}}\ \texttt{chunk} \quad 3^{\text{rd}}\ \texttt{chunk} \quad 4^{\text{th}}\ \texttt{chunk}
\end{array}
$$

Figure 3.1: A sample string for the proof of Lemma 3.4.

in $O((\lg\lg\lg\sigma)^2(f(n,\sigma)+\lg\lg\sigma))$ *time, and* `string_select` *for any character* $\alpha \in [\sigma]$ *in* $O(\lg\lg\lg\sigma(f(n,\sigma)+\lg\lg\sigma))$ *time.*

Proof. As $\texttt{string\_rank}(\overline{\alpha}, x) = x - \texttt{string\_rank}(\alpha, x)$ for $\alpha \in [\sigma]$, we only need show how to support `string_rank` and `string_select` for $\alpha \in [\sigma]$.

First we conceptually treat the given string S and portions of S in several ways. We treat S as an $n \times \sigma$ table E with rows indexed by $1, 2, \ldots, \sigma$ and columns by $1, 2, \ldots, n$. For any $\alpha \in [\sigma]$ and $x \in [n]$, entry $E[\alpha][x] = 1$ if $S[x] = \alpha$, and $E[\alpha][x] = 0$ otherwise. Reading E in row major order yields a conceptual bit vector A of length σn with exactly n 1s. We divide A into *blocks* of size σ. The *cardinality* of a block is the number of 1s in it. A *chunk* of S is a substring of length σ (we assume that n is divisible by σ for simplicity), so that for the i^{th} chunk C, we have $C[j] = S[(i-1)\sigma + j]$, where $i \in [n/\sigma]$ and $j \in [\sigma]$. Hence a chunk corresponds to a $\sigma \times \sigma$ segment of E, or σ equally spaced substrings of A. We denote the block corresponding to the α^{th} row of the segment of E corresponding to a chunk C by C_α, where $\alpha \in [\sigma]$. Figure 3.1 illustrates these concepts. In this example, let C be the 4^{th} chunk. Then we have $C_2 = 0011$.

We first construct a bit vector B which stores the cardinalities of all the blocks in unary (i.e. a block of cardinality l is stored as l 1s followed by a 0), in the order they appear in A, so that $B = 1^{l_1}01^{l_2}0\ldots1^{l_n}0$, where l_i is the cardinality of the i^{th} block of A. The length of B is $2n$, as there are exactly n 1s in A, and n blocks. We store it using Part (a) of Lemma 2.1 in $2n+o(n)$ bits. For the example in Figure 3.1, $B = 11100110101010101100101000110010$.

Using this bit vector B, the support for `string_rank` and `string_select` operations on S can be reduced, in constant time, to supporting these operations on a given chunk as suggested by Golynski *et al.* [41, Section 2]. To be specific, to compute $\texttt{string\_rank}_S(\alpha, x)$, let C be the chunk that position x is in (i.e. C is the u^{th} chunk, where $u = \lceil x/\sigma \rceil$). We observe that $\texttt{string\_rank}_S(\alpha, x) = \texttt{string\_rank}_S(\alpha, (u-1)\sigma) + \texttt{string\_rank}_C(\alpha, x \bmod \alpha)$. The first item on the right side of this equation can be computed using B as follows: Let a_1 and a_2 be the positions of the 0s in B that correspond to the last block in the $(\alpha-1)^{\text{st}}$ row of E, and the last block before block C_α in the α^{th} row of E, respectively. Then $a_1 = \texttt{bin\_select}_B(0, n(\alpha-1)/\sigma)$ and $a_2 = \texttt{bin\_select}_B(0, n(\alpha-1)/\sigma + u - 1)$. Thus by the definition of B, the following equation holds: $\texttt{string\_rank}_S(\alpha, (u-1)\sigma) = \texttt{bin\_rank}_B(1, a_2) - \texttt{bin\_rank}_B(1, a_1)$. Therefore, we only need compute $\texttt{string\_rank}_C(\alpha, x \bmod \alpha)$.

To compute $\texttt{string\_select}_S(\alpha, r)$, we first compute the position, v, of the 1 in B that corresponds to the r^{th} α in S. Let the number of 1s in $B[1..v]$ to be q. As there are $\texttt{bin\_rank}_B(1, a_1)$ 1s above the α^{th} row in E, we have $q = \texttt{bin\_rank}_B(1, a_1) + r$. As $v = \texttt{bin\_select}_B(1, q)$, we can compute v in constant time. Let the block that contains the q^{th} 1 in A be the y^{th} block in E in the row major order, and the chunk, C', that contains the r^{th} α be the w^{th} chunk of the string S. Then $y = \texttt{bin\_rank}_B(0, v) + 1$ and $w = y - (\alpha-1)n/\sigma$. Thus we have $\texttt{string\_select}_S(\alpha, r) = \texttt{string\_select}_{C'}(\alpha, q - \texttt{bin\_rank}_B(1, \texttt{bin\_select}_B(0, y-1))) + (w-1)\sigma$.

Hence we only need show how to support `string_rank` and `string_select` on a given chunk C.

We store the following data structures for each chunk C:

- We construct a bit vector X that stores the cardinalities of the blocks in C in unary from top to bottom, i.e. $X = 1^{l_1}01^{l_2}0 \ldots 1^{l_\sigma}0$, where l_α is the number of 1s in the block C_α. There are σ 1s in X, each corresponding to a character of the chunk, and σ 0s, each corresponding to a block of the chunk. Hence the length of X is 2σ. We store it in $2\sigma + o(\sigma)$ bits using Part (a) of Lemma 2.1.

- We construct an array R such that $R[j] = \texttt{bin\_rank}_D(1, j) \mod k$, where D is the block $C_{C[j]}$, and k is a parameter which we fix later. Each element of R is an integer in the range $[0, k-1]$, so R can be stored in $\sigma \lg k$ bits.

- We construct a conceptual permutation π on $[\sigma]$, defined later in the proof. We store an auxiliary structure P that takes $O(\sigma \lg \sigma / s + n \lg\lg n / \lg n)$ bits using Lemma 3.1, where s is a parameter which we fix later, and supports access to π in $O(s \cdot g(n,\sigma))$ time, given $O(g(n,\sigma))$-time access to π^{-1}.

- For each block C_α in a chunk C, let F_α be a "sparsified" bit vector for C_α, in which only every k^{th} 1 of C_α is present (i.e. $F_\alpha[j] = 1$ iff $C_\alpha[j] = 1$ and $\texttt{bin\_rank}(1, j)$ on C_α is divisible by k). We encode F_α using Lemma 3.2 in $O(\lg\sigma \times l_\alpha/k)$ bits to support $\texttt{bin\_rank}_{F_\alpha}(1, i)$ in $O(\lg\lg\sigma)$ time. All the F_α's in a given chunk thus occupy $O(\sigma\lg\sigma/k)$ bits in total.

We first show how to support $\texttt{bin\_rank}'(1, j)$ on block $D = C_{C[j]}$ (note that $D[j] = 1$; hence $\texttt{bin\_rank}'(1, j)$ is defined and equivalent to $\texttt{bin\_rank}(1, j)$). For this, we first compute $C[j]$ using $\texttt{string\_access}$ in $f(n,\sigma)$ time. Then we compute $\texttt{bin\_rank}(1, j)$ on $F_{C[j]}$ in $O(\lg\lg\sigma)$ time, which is equal to $\lfloor \texttt{bin\_rank}'_D(1, j)/k \rfloor$. Hence, we can compute $k\lfloor \texttt{bin\_rank}'_D(1, j)/k \rfloor$ in $O(\lg\lg\sigma)$ time. We also retrieve $R[j]$ in constant time, which is equal to $\texttt{bin\_rank}'_D(1, j) \mod k$. As $\texttt{bin\_rank}'_D(1, j) = k\lfloor \texttt{bin\_rank}'_D(1, j)/k \rfloor + \texttt{bin\_rank}'_D(1, j) \mod k$, we can compute $\texttt{bin\_rank}'_D(1, j)$ in $O(f(n,\sigma) + \lg\lg\sigma)$ time.

The permutation π for a chunk C is obtained by writing down the positions (relative to the starting position of the chunk) of all the occurrences of each character α in increasing order, if α appears in C, for $\alpha = 1, 2, \cdots, \sigma$. For example, in Figure 3.1, let C be the 4^{th} chunk. Then $\pi = 1, 3, 4, 2$. Using π^{-1} to denote the inverse of π (in the previous example, $\pi^{-1} = 1, 4, 2, 3$), we see that $\pi^{-1}(j)$ is equal to the sum of the following two values: the number of characters smaller than $C[j]$ in C, and $\texttt{bin\_rank}'(1, j)$ on block $D = C_{C[j]}$. The first value can be computed using X in constant time, as it is equal to $\texttt{bin\_rank}_X(\texttt{bin\_select}_X(0, \alpha - 1))$, and we have already shown how to compute the second value in $O(f(n,\sigma) + \lg\lg\sigma)$ time in the previous paragraph. Therefore, we can compute any element of π^{-1} in $O(f(n,\sigma) + \lg\lg\sigma)$ time. We can further use P to compute any element of π in $O(s(f(n,\sigma) + \lg\lg\sigma))$ time (note that the $f(n,\sigma) + \lg\lg\sigma$ term here comes from the time required to retrieve a given element of π^{-1}).

Golynski *et al.* [41, Section 2.2] showed how to compute $\texttt{string\_select}$ on a chunk C by a single access to π plus a few constant-time operations. This is achievable because π stores the positions of the occurrences of characters that appears in C. More precisely, to compute $\texttt{string\_select}_C(\alpha, r)$, we first use X to compute the number of the occurrences of α in C, which is $d = \texttt{bin\_rank}_X(1, \texttt{bin\_select}_X(0, \alpha)) - \texttt{bin\_rank}_X(1, \texttt{bin\_select}_X(0, \alpha-1))$. We return ∞ if $d < r$. Otherwise, we have that $\texttt{string\_select}_C(\alpha, r) = \pi(\texttt{bin\_select}_X(0, \alpha - 1) + r)$. When combined with our approach, we can support `string_select` for any character $\alpha \in [\sigma]$ in $O(s(f(n, \sigma) + \lg\lg\sigma))$ time.

Golynski *et al.* [41, Section 2.2] also showed how to compute `string_rank` by calling `string_select` $O(\lg k)$ times. To be specific, to compute $\texttt{string\_rank}_C(\alpha, x)$, let $r_1 = k\lfloor \texttt{string\_rank}_C(\alpha, x)/k \rfloor$ and $r_2 = r_1 + k - 1$. We can compute r_1 and r_2 in $O(\lg\lg\sigma)$ time, as $\lfloor \texttt{string\_rank}_C(\alpha, x)/k \rfloor$ is equal to $\texttt{bin\_rank}(1, x)$ on F_α (i.e. the "sparsified" bit vector for the block C_α). As $r_1 \leq \texttt{string\_rank}_C(\alpha, x) \leq r_2$, we then perform a binary search in the range $[r_1, r_2]$. In each phase of the loop, we use `string_select` to check whether we have found the answer. Thus we can support operator `string_rank` in $O(s \lg k(f(n, \sigma) + \lg\lg\sigma))$ time.

As there are n/σ chunks, the sum of the space costs of the auxiliary structures constructed for all the chunks is clearly $O(n \lg k + n \lg\sigma(1/s + 1/k))$ bits. Choosing $s = \lg\lg\lg\sigma$ and $k = \lg\lg\sigma$ makes the overall space cost of all the auxiliary structures to be $O(n(\lg\sigma/\lg\lg\lg\sigma)) = n \cdot o(\lg\sigma)$. The query times for `string_select` and `string_rank` would then be $O((\lg\lg\lg\sigma)^2(f(n, \sigma) + \lg\lg\sigma))$ and $O(\lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ respectively. □

3.4.3 Supporting α-Predecessor and α-Successor Queries

We now extend our succinct indexes to support α-predecessor and α-successor queries.

Lemma 3.5. *Using at most $2n + o(n)$ additional bits, the succinct index of Lemma 3.4 also supports* `string_pred` *and* `string_succ` *for any character $\alpha \in [\sigma]$ in $O((\lg\lg\lg\sigma)^2(f(n, \sigma) + \lg\lg\sigma))$ time, and these two operators for any literal $\alpha \in [\bar{\sigma}]$ in $O(f(n, \sigma) + \lg\lg\sigma)$ time.*

Proof. We only show how to support `string_pred`; `string_succ` can be supported similarly. For any $\alpha \in [\sigma]$, $\texttt{string\_pred}(\alpha, x) = \texttt{string\_select}(\alpha, \texttt{string\_rank}(\alpha, x) - 1)$. Thus the operators `string_pred` and `string_succ` can be supported for any character $\alpha \in [\sigma]$ in $O((\lg\lg\lg\sigma)^2(f(n, \sigma) + \lg\lg\sigma))$ time. Hence we only need show how to support $\texttt{string\_pred}(\alpha, x)$ when $\alpha \in [\bar{\sigma}]$.

For this we require another auxiliary structure. In the bit vector A, there are n 1s, so there are at most n runs of consecutive 1s. Assume that there are u runs and their lengths are $p_1, p_2, \ldots, p_u$, respectively. We store these lengths in unary using a bit vector U, i.e. $U = 1^{p_1}01^{p_2}0 \cdots 1^{p_u}0$. The length of U is $n + u \leq 2n$, and we store it using Part (a) of Lemma 2.1 in at most $2n + o(n)$ bits.

To support $\texttt{string\_pred}(\alpha, x)$ for $\alpha \in [\bar{\sigma}]$, let c be the character such that $\alpha = \bar{c}$. We first retrieve $S[x-1]$ using $\texttt{string\_access}$ in $f(n, \sigma)$ time. If $S[x-1] \neq c$, then we return $x-1$. Otherwise, we compute the number, j, of 1s up to position $(c-1)\sigma + x - 1$ in A (this position in A corresponds to the $(x-1)^{\text{th}}$ position in the c^{th} row in table E). Let C be the chunk that contains the $(x-1)^{\text{th}}$ position of S. As $j = \texttt{bin\_rank}_B(1, \texttt{bin\_select}_B(0, (c-1)n/\sigma + \lfloor (x-1)/\sigma \rfloor)) + \texttt{bin\_rank}'_{C_c}(1, (x-1) \mod \sigma)$, we can compute j in $O(f(n, \sigma) + \lg\lg\sigma)$ time (the proof of Lemma 3.4 shows how to compute $\texttt{bin\_rank}'_D(1, k)$ in $O(f(n, \sigma) + \lg\lg\sigma)$ time, for any block D and position k such that $D[k] = 1$). The position in U that corresponds to the $(x-1)^{\text{th}}$ position in the c^{th} row in table E is $v = \texttt{bin\_select}_U(1, j)$. Thus the number of consecutive 1s preceding and including position v in U is $q = v - \texttt{bin\_select}_U(0, \texttt{bin\_rank}_U(0, v))$. If $q \geq x - 1$, then there is no 0 in front of position $x-1$ in row c of table E, so we return $-\infty$. Otherwise, we return $x - q - 1$ as the result. All the above operations take $O(f(n, \sigma) + \lg\lg\sigma)$ time. Therefore, $\texttt{string\_pred}$ and $\texttt{string\_succ}$ can be supported for any literal $\alpha \in [\bar{\sigma}]$ in $O(f(n, \sigma) + \lg\lg\sigma)$ time. □

Combining Lemmas 3.4 and 3.5, we have our first main result:

Theorem 3.1. *Given support for* $\texttt{string\_access}$ *in* $f(n, \sigma)$ *time on a string* $S \in [\sigma]^n$, *there is a succinct index using* $n \cdot o(\lg\sigma)$ *bits that supports:*

- $\texttt{string\_rank}$ *for any literal* $\alpha \in [\sigma] \cup [\bar{\sigma}]$ *in* $O((\lg\lg\lg\sigma)^2(f(n, \sigma) + \lg\lg\sigma))$ *time;*
- $\texttt{string\_select}$ *for any character* $\alpha \in [\sigma]$ *in* $O(\lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ *time;*
- $\texttt{string\_pred}$ *and* $\texttt{string\_succ}$ *for any character* $\alpha \in [\sigma]$ *in* $O((\lg\lg\lg\sigma)^2(f(n, \sigma) + \lg\lg\sigma))$ *time, and these two operations for* $\alpha \in [\bar{\sigma}]$ *in* $O(f(n, \sigma) + \lg\lg\sigma)$ *time.*

3.4.4 Using string_select to Access the Data

We can alternatively define the ADT of a string through the $\texttt{string\_select}(\alpha, r)$ operator, where $\alpha \in [\sigma]$. Although this definition seems unusual, it has a useful application in Section 4.7.3. With this definition, we have:

Theorem 3.2. *Given support for* string_select *(for any character $\alpha \in [\sigma]$) in $f(n, \sigma)$ time on a string $S \in [\sigma]^n$, there is a succinct index using $n \cdot o(\lg \sigma)$ bits that supports* string_rank*,* string_pred *and* string_succ *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$, as well as* string_access*, in $O(\lg \lg \sigma f(n, \sigma))$ time.*

Proof. As in the proof of Lemma 3.4, we divide string S and its corresponding conceptual table E into chunks and blocks, and construct bit vector B for the entire string, and bit vector X and the auxiliary structure P for each chunk. We also store the same set of "sparsified" bit vectors, F_α's, for each chunk. With the $f(n, \sigma)$-time support for string_select on S, using the method described in the proof of Lemma 3.4, we can support string_rank on S in $O(\lg \lg \sigma + \lg k f(n, \sigma))$ time.

Now we provide support for string_access. We first design the data structures supporting the access to π and π^{-1} for any chunk C (see the proof of Lemma 3.4 for the definition of π and π^{-1}). We assume that C is the i^{th} chunk of S. From the definition of π we have that $\pi(j) = \texttt{bin\_select}(1, r)$ on the block C_α, where the r^{th} occurrence of α in C corresponds to the j^{th} 1 in X. As $\alpha = \texttt{bin\_rank}_X(0, \texttt{bin\_select}_X(1, j)) + 1$, and $r = \texttt{bin\_select}_X(1, j) - \texttt{bin\_select}_X(0, \alpha - 1)$, α and r can be computed in $O(1)$ time. As $\texttt{bin\_select}_{C_\alpha}(1, r) = \texttt{string\_select}(\alpha, r + z)$, where z is the number of 1s in the α^{th} row of E up to position $(i - 1)\sigma$, we only need show how to compute z. Let a_1 and a_2 be the positions of the 0s in B that correspond to the last block in the $(\alpha - 1)^{\text{th}}$ row of E, and the block in the α^{th} row of E that ends at position $(i - 1)\sigma$, respectively. Then $a_1 = \texttt{bin\_select}_B(0, (\alpha - 1)n/\sigma)$ and $a_2 = \texttt{bin\_select}_B(0, (\alpha - 1)n/\sigma + i - 1)$. As $z = \texttt{bin\_rank}_B(1, a_2) - \texttt{bin\_rank}_B(1, a_1)$, we can compute z in constant time. Thus we can compute $\pi(j)$ in $f(n, \sigma)$ time. With the auxiliary structure P, we can further compute any element of π^{-1} in $O(sf(n, \sigma))$ time by Lemma 3.1.

With the support for the access to π^{-1}, we can now use the method of Golynski *et al.* [41, Section 2.2] to compute $C[j]$ as follows. We first compute $\pi^{-1}(j)$ in $O(sf(n, \sigma))$

time. By the definition of π^{-1}, we observe that the $(\pi^{-1}(j))^{\text{th}}$ 1 in X corresponds to $C[j]$. Thus $C[j] = \texttt{bin\_rank}_X(0, \pi^{-1}(j)) + 1$. Therefore, we can compute $C[j]$ in $O(sf(n,\sigma))$ time. Finally, by the equation $C[j] = S[(i-1)\sigma + j]$ for the i^{th} chunk of S, `string_access` can be supported in $O(sf(n,\sigma))$ time.

To analyze the space cost of the auxiliary data structures used to support `string_rank` and `string_access` (B, X, y-fast tries, and P), we observe that the same data structures are defined in the proof of Lemma 3.1. Thus they occupy $O(n \lg \sigma(1/s + 1/k))$ bits. Choosing $s = \lg\lg\sigma$ and $k = \lg\sigma$ makes the overall space cost of all the auxiliary structures to be $O(n\lg\sigma/\lg\lg\sigma + n) = n \cdot o(\lg\sigma)$ bits. The query time for `string_rank` and `string_access` would be $O(\lg\lg\sigma f(n,\sigma))$.

Finally, we show how to support `string_pred` and `string_succ`. Same as the proof of Lemma 3.5, we only need show how to support `string_pred` for $\alpha \in [\bar{\sigma}]$. We construct the bit vector U as in the proof of Lemma 3.5, which occupies at most $2n + o(n)$ bits. To compute `string_pred`(α, x) for $\alpha \in [\bar{\sigma}]$, let c be the character such that $\alpha = \bar{c}$. We first retrieve $S[x-1]$ in $\lg\lg\sigma f(n,\sigma)$ time. If $S[x-1] \neq c$, then we return $x-1$. Otherwise, we compute the number, j, of 1s up to and including position $(c-1)\sigma + x - 1$ in A (this position in A corresponds to the $(x-1)^{\text{th}}$ position in the c^{th} row in table E). As $j = \texttt{bin\_rank}_B(1, \texttt{bin\_select}_B(0, (c-1)n/\sigma)) + \texttt{string\_rank}_S(c, x-1)$, we can compute j in $O(\lg\lg\sigma f(n,\sigma))$ time. The position in U that corresponds to the $(x-1)^{\text{th}}$ position in the c^{th} row in table E is $v = \texttt{bin\_select}_U(1, j)$. Thus the number of consecutive 1s preceding and including position v in U is $q = v - \texttt{bin\_select}_U(0, \texttt{bin\_rank}_U(0, v))$. If $q \geq x-1$, then there is no 0 in front of position $x-1$ in row c of table E, so we return $-\infty$. Otherwise, we return $x - q - 1$ as the result. All the above operations take $O(\lg\lg\sigma f(n,\sigma))$ time. Therefore, `string_pred` and `string_succ` can be supported in $O(\lg\lg\sigma f(n,\sigma))$ time. □

3.5 Binary Relations

3.5.1 Definitions

We consider a binary relation R, relating an object set $[n]$ and a label set $[\sigma]$, and containing t pairs. We adopt the assumption that each object is associated with at least one label (thus $t \geq n$), and $n \geq \sigma$ (the converse is symmetric). We show how to extend the results to other cases by simple techniques after the proof of each Theorem. We define the interface of the ADT of a binary relation through the operator `object_select` defined in Section 3.2.2 that can be used to obtain the labels associated with a given object.

We generalize the definition of literals to binary relations:

Definition 3.3. *Consider a binary relation formed by t pairs from an object set $[n]$ and a label set $[\sigma]$. An object $x \in [n]$* **matches literal** *$\alpha \in [\sigma]$ if x is associated with α. An object $x \in [n]$* **matches literal** *$\bar{\alpha}$ if x is not associated with α. For simplicity, we define $[\bar{\sigma}]$ to be the set $\{\overline{1}, \ldots, \overline{\sigma}\}$.*

We also generalize the definition of α-predecessor and α-successor to binary relations.

Definition 3.4. *Consider a binary relation formed by t pairs from an object set $[n]$ and a label set $[\sigma]$, a literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ and an object $x \in [n]$. The* **$\boldsymbol{\alpha}$-predecessor of object x**, *denoted by* `label_pred`(α, x), *is the last object matching α before (and not including) object x, if it exists. Similarly, the* **$\boldsymbol{\alpha}$-successor of object x**, *denoted by* `label_succ`(α, x), *is the first object matching α after (and not including) object x, if it exists.*

3.5.2 Succinct Indexes

Theorem 3.3. *Given support for* `object_select` *in $f(n, \sigma, t)$ time on a binary relation R formed by t pairs from an object set $[n]$ and a label set $[\sigma]$, there is a succinct index using $t \cdot o(\lg \sigma)$ bits that supports:*

- `label_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O((\lg \lg \lg \sigma)^2 (f(n, \sigma, t) + \lg \lg \sigma))$ time;*
- `label_select` *for any label $\alpha \in [\sigma]$ in $O(\lg \lg \lg \sigma (f(n, \sigma, t) + \lg \lg \sigma))$ time;*
- `label_pred` *and* `label_succ` *for any label $\alpha \in [\sigma]$ in $O((\lg \lg \lg \sigma)^2 (f(n, \sigma, t) + \lg \lg \sigma))$ time, and these two operations for any literal $\alpha \in [\bar{\sigma}]$ in $O(f(n, \sigma, t) + \lg \lg \sigma)$ time;*
- `object_rank` *and* `label_access` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg \lg \lg \sigma f(n, \sigma, t) + \lg \lg \sigma)$ time;*
- `label_nb` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ and* `object_nb` *in $O(1)$ time.*

$$E=\begin{pmatrix} 0&1&0&1&0\\ 0&0&0&1&0\\ 1&0&1&1&0\\ 1&1&0&0&1 \end{pmatrix}$$

```
COLUMNS = 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0
ROWS    = 3, 4,    1, 4,    3,    1, 2, 3,    4
```

Figure 3.2: An example of the encoding of a binary relation.

Proof. As with strings, we also conceptually treat a binary relation as an $n \times \sigma$ table E, and entry $E[\alpha][x] = 1$ iff object x is associated with label α. A binary relation on t pairs from $[n] \times [\sigma]$ can be stored as follows [5] (See Figure 3.2 for an example):

- a string `ROWS` of length t drawn from alphabet $[\sigma]$, such that the i^{th} label of `ROWS` is the label of the i^{th} pair in the column-major order traversal of E;
- a bit vector `COLUMNS` of length $n + t$ encoding the number of labels associated with each object in unary.

To design a succinct index for binary relations, we explicitly store the bit vector `COLUMNS` using Part (a) of Lemma 2.1 in $n + t + o(n + t)$ bits. We now show how to support `string_access` on `ROWS` using `object_select`. To compute the i^{th} character in `ROWS`, we need compute the corresponding object, x, and the rank, r, of the corresponding label among all the labels associated with x. The position of the 1 in `COLUMNS` corresponding to the i^{th} character in `ROWS` is $l = \texttt{bin\_select}_{\texttt{COLUMNS}}(1, i)$. Therefore, $x = \texttt{bin\_rank}_{\texttt{COLUMNS}}(0, l) + 1$, and $r = l - \texttt{bin\_select}_{\texttt{COLUMNS}}(0, x - 1)$ if $x > 1$ ($r = l$ otherwise). Thus with these additional operations, we can support `string_access` in $O(f(n, \sigma, t))$ time using one call to `object_select` in addition to some constant-time operations.

We store a succinct index for `ROWS` using Theorem 3.1 in $t \cdot o(\lg \sigma)$ bits. As we can support `string_access` on `ROWS` using `object_access`, the index can support `string_rank` for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O((\lg\lg\lg \sigma)^2(f(n, \sigma) + \lg\lg \sigma))$ time, `string_pred` and `string_succ` for any character $\alpha \in [\sigma]$ in $O((\lg\lg\lg \sigma)^2(f(n, \sigma) + \lg\lg \sigma))$ time, `string_pred` and `string_succ` for any literal $\alpha \in [\bar{\sigma}]$ in $O(f(n, \sigma) + \lg\lg \sigma)$ time, and `string_select` for any character $\alpha \in [\sigma]$ in $O(\lg\lg\lg \sigma(f(n, \sigma) + \lg\lg \sigma))$ time. With this, we can use the approach of Barbay *et al.* [5, Theorem 1] to support `label_rank`, `label_select` and `label_access` operations on binary relations using rank/select on `ROWS` and `COLUMNS` as follows.

To compute $\texttt{label\_rank}(\alpha, x)$, we observe that the position of the 0 in COLUMNS that corresponds to the x^{th} column of E is $j = \texttt{bin\_select}_{\texttt{COLUMNS}}(0, x)$. Then the position of the last label associated with object x in ROWS is $k = \texttt{bin\_rank}_{\texttt{COLUMNS}}(1, j)$. As $\texttt{label\_rank}(\alpha, x) = \texttt{string\_rank}_{\texttt{ROWS}}(\alpha, k)$, we can support label_rank in $O((\lg\lg\lg\sigma)^2(f(n, \sigma, t) + \lg\lg\sigma))$ time.

To compute $\texttt{label\_select}(\alpha, r)$, we first observe that the position of the r^{th} occurrence of α in ROWS is $u = \texttt{string\_select}_{\texttt{ROWS}}(\alpha, r)$. The position of the 1 that corresponds to this character in COLUMNS is $v = \texttt{bin\_select}_{\texttt{COLUMNS}}(1, u)$, which corresponds to object $\texttt{bin\_rank}_{\texttt{COLUMNS}}(0, v) + 1$. This object is the answer. Thus label_select can be supported in $O(\lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ time.

To compute $\texttt{object\_nb}(x)$, we observe that the result is the x^{th} number encoded in COLUMNS in unary. Thus $\texttt{object\_nb}(x) = \texttt{bin\_rank}_{\texttt{COLUMNS}}(1, \texttt{bin\_select}_{\texttt{COLUMNS}}(0, x)) - \texttt{bin\_rank}_{\texttt{COLUMNS}}(1, \texttt{bin\_select}_{\texttt{COLUMNS}}(0, x - 1))$, so we can support object_nb in constant time. We can support label_nb for $\alpha \in [\sigma]$ in the same manner by encoding the number of objects associated with each label in unary in another bit vector W. For the example in Figure 3.2, $W = 1101011101110$. W occupies $n + t + o(n + t)$ bits. To support label_nb for $\alpha \in [\bar{\sigma}]$, we use the equation $\texttt{label\_nb}(\alpha) = n - \texttt{label\_nb}(c)$, where c is the label such that $\alpha = \bar{c}$.

To support object_rank, we construct, for each object y, a bit vector G_y of length σ, in which $G_y[\beta] = 1$ iff object y is associated with label β and $\texttt{object\_rank}(y, \beta)$ is divisible by $\lg\lg\sigma$. We encode G_y using Lemma 3.2. Let l_x be the number of labels associated with x. Then the number of 1s in G_y is $\lfloor l_x / \lg\lg\sigma \rfloor$. Hence G_y occupies $O(\lfloor l_x / \lg\lg\sigma \rfloor \times \lg\sigma) = O(l_x \lg\sigma / \lg\lg\sigma)$ bits, and the total space cost of all the G_y's is $O(t \lg\sigma / \lg\lg\sigma)$ bits. To compute $\texttt{object\_rank}(x, \alpha)$, let $r_1 = \lg\lg\sigma \lfloor \texttt{object\_rank}(\alpha, x) / \lg\lg\sigma \rfloor$ and $r_2 = r_1 + \lg\lg\sigma - 1$. We can compute r_1 and r_2 in $O(\lg\lg\sigma)$ time, as $\lfloor \texttt{object\_rank}(\alpha, x) / \lg\lg\sigma \rfloor$ is equal to $\texttt{bin\_rank}(1, \alpha)$ on G_x. As $r_1 \leq \texttt{object\_rank}(\alpha, x) \leq r_2$, we then perform a binary search in the range $[r_1, r_2]$. In each phase of the loop, we use object_select to check whether we have found the answer. Thus we can support operator object_rank in $O(\lg\lg\lg\sigma f(n, \sigma, t) + \lg\lg\sigma)$ time.

To compute $\texttt{label\_access}(x, \alpha)$, we make use of the fact that object x is labeled α iff $\texttt{object\_rank}(x, \alpha) - \texttt{object\_rank}(x, \alpha - 1)$ is 1. Therefore, label_access can be supported in $O(\lg\lg\lg\sigma f(n,\sigma,t)+\lg\lg\sigma)$ time.

We now design algorithms to support `label_pred` and `label_succ`. We show how to compute `label_pred`(α, x); `label_succ` can be supported similarly. The position of the first label associated with x in `ROWS` is $p = \texttt{bin\_rank}_{\texttt{COLUMNS}}(1, \texttt{bin\_select}_{\texttt{COLUMNS}}(0, x-1)) + 1$. Thus the position of the last occurrence of character α in `ROWS`$[1..p-1]$ is $q = \texttt{string\_pred}_{\texttt{ROWS}}(\alpha, p)$, and the object associated with the label that corresponds to this occurrence is $\texttt{bin\_rank}_{\texttt{COLUMNS}}(0, \texttt{bin\_select}_{\texttt{COLUMNS}}(1, q)) + 1$. This object is the answer. Hence we can support `label_succ`(α, x) for $\alpha \in [\sigma]$ in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ time, and the same operation for $\alpha \in [\bar{\sigma}]$ in $O(f(n,\sigma,t)+\lg\lg\sigma)$ time.

The space of the index is the sum of space cost of storing `COLUMNS`, W, G_y's and the index for `ROWS`, which is at most $n+t+o(n+t)+n+t+o(n+t)+t\lg\sigma/\lg\lg\sigma+t\cdot o(\lg\sigma) = t\cdot o(\lg\sigma)$ bits. □

Note that the above approach also works without the assumption that each object is associated with at least one label, though we can not use the inequality $t \geq n$ to analyze the space cost. Thus without such an assumption, our succinct index occupies $t\lg\sigma/\lg\lg\sigma+n+o(n)$ bits. By the discussions in Section 3.3.4, the space cost of our succinct index is a lower order term of the information-theoretic minimum, when $t/n = \sigma^{o(1)}$ and $t \geq n$.

As we treat a binary relation as an $n \times \sigma$ boolean matrix with t 1s in the proof of Theorem 3.3, our result also applies to the problem of succinctly representing a boolean matrix to allow rank/select on rows and columns. By the analysis in the above paragraph, our solution is particularly space-efficient for sparse boolean matrices.

3.6 Applications

3.6.1 High-Order Entropy-Compressed Succinct Encodings for Strings

Given a string S of length n over alphabet $[\sigma]$, we now design a high-order entropy-compressed succinct encoding for it that supports `string_access`, `string_rank`, and `string_select` efficiently. Golynski *et al.* [5] considered the problem and suggested a method with space requirements proportional to the k^{th} order entropy of a different but related string. Here we solve the problem in its original form.

Theorem 3.4. *A string S of length n over alphabet $[\sigma]$ can be represented using $nH_k(S) + \lg\sigma \cdot o(n) + n \cdot o(\lg\sigma)$ bits for any $k = o(\log_\sigma n)$, to support:*

- `string_access` *in $O(1)$ time;*
- `string_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg\lg\sigma(\lg\lg\lg\sigma)^2)$ time;*
- `string_select` *for any character $\alpha \in [\sigma]$ $O(\lg\lg\sigma\lg\lg\lg\sigma)$ time;*
- `string_pred` *and* `string_succ` *for any literal $\alpha \in [\sigma]$ in $O(\lg\lg\sigma(\lg\lg\lg\sigma)^2)$ time, and these two operations for any literal $\alpha \in [\bar{\sigma}]$ in $O(\lg\lg\sigma)$ time.*

When $\sigma = O(\lg n/\lg\lg n)$, S can be represented using $nH_k(S) + \lg\sigma \cdot o(n) + O(n)$ bits to support the above operations in $O(1)$ time.

Proof. We use Lemma 3.3.3 to store S in $nH_k(S) + O(\frac{n}{\log_\sigma n}(k\lg\sigma + \lg\lg n))$ bits. When $k = o(\log_\sigma n)$, the above space cost is $H_k(S) + \lg\sigma \cdot o(n)$. This representation allows us to retrieve any $O(\lg n)$ consecutive bits of the string in $O(1)$ time. Thus we can use it to retrieve $S[i]$ in $O(1)$ time (i.e. `string_access` can be supported in $O(1)$ time).

We store a succinct index for S using Theorem 3.1, and the support for `string_rank`, `string_select`, `string_pred` and `string_succ` for arbitrary σ immediately follows. The overall space is $nH_k + \lg\sigma \cdot o(n) + O(n\lg\sigma/\lg\lg\lg\sigma) = nH_k(S) + \lg\sigma \cdot o(n) + n \cdot o(\lg\sigma)$.

When $\sigma = O(\lg n/\lg\lg n)$, instead of constructing the entire succinct index for S, we construct the following auxiliary structures. We conceptually divide the string into chunks and blocks, and construct the bit vector B as in the proof of Lemma 3.4. This reduces the support for `string_rank` and `string_select` on S, to the support for these two operations on any given chunk C (see the proof of Lemma 3.4).

Let $l = \lfloor \log_2 n/(2\log_2\sigma) \rfloor$. We construct a table L, in which for each character $\alpha \in [\sigma]$, each integer $i \in [l]$, and each possible string $D \in [\sigma]^l$, we store the results of queries $\texttt{string\_rank}_D(\alpha, i)$ and $\texttt{string\_select}_D(\alpha, i)$ (i.e. $L[D, \alpha, i]$ stores $\texttt{string\_rank}_D(\alpha, i)$ and $\texttt{string\_select}_D(\alpha, i)$). As there are $\sigma^l \leq \sigma^{\log_2 n/(2\log_2\sigma)} = \sigma^{\frac{1}{2}\log_\sigma n} = \sqrt{n}$ different strings of length l over alphabet $[\sigma]$, We can encode each possible string using $\lceil \log_2 n/2 \rceil$

bits, which fits in a constant number of words. We can store the result of each query above in $\lg(l+1)$ bits. Thus the table L occupies at most $\sigma \times l \times \sqrt{n} \times \lg(l+1) = O(\lg n/\lg\lg n \times \sqrt{n} \times l \lg l) = O(\sqrt{n}\lg^2 n) = o(n)$ bits. Using the table L, we can answer queries $\texttt{string\_rank}(\alpha, i)$ and $\texttt{string\_select}(\alpha, i)$ on any string $D \in [\sigma]^l$ in constant time by performing a table lookup on L (as $L[D, \alpha, i]$ stores the answers). We can also support `string_rank` and `string_select` on any string G whose length, h, is less than l. This can be done by first appending the string with the first character till its length is l (on a word RAM, this step can be performed using a right shift of the binary encoding of G in constant time), and then use the resulting string, F, as a parameter to perform table lookups. Finally, as $\texttt{string\_rank}_G(\alpha, i) = \texttt{string\_rank}_F(\alpha, i)$ for $i \leq h$, and $\texttt{string\_select}_G(\alpha, i) = \texttt{string\_select}_F(\alpha, i)$ if $\texttt{string\_select}_F(\alpha, i) \leq h$ ($\texttt{string\_select}_G(\alpha, i) = \infty$ otherwise), we can support `string_rank` and `string_select` on G in constant time.

To support `string_rank` and `string_select` on any chunk C, we observe that $l = \Omega(\lg n/\lg\lg n)$. Therefore, the length of a chunk is either shorter than l, or can be divided into a constant number of substrings of length l and a substring of length at most l. To handle the latter case (the first case is already supported in the above paragraph), when answering $\texttt{string\_rank}_C(\alpha, i)$, we using table L to compute the number of α's in the substrings that appear before position i, and using table L to compute $\texttt{string\_rank}(\alpha, i \bmod l)$ on the substring that contains position i, and the sum of these values is the result. To compute $\texttt{string\_select}_C(\alpha, i)$, we computer the number of occurrences of α in each substring from left to right, and compute the subset sum, till we locate the substring that contains the result. We then use table lookup to retrieve the result.

To support `string_pred` and `string_succ`, we construct the bit vector U using at most $2n + o(n)$ bits as in the proof of Lemma 3.5, and use the same algorithm to support `string_pred` and `string_succ` in constant time.

The auxiliary data structures B, L and U occupies $O(n)$ bits in total, so the overall space cost is $nH_k(S) + \lg\sigma \cdot o(n) + O(n)$ bits. □

Using similar approaches, we can design succinct encodings for binary relations based on our succinct indexes, and compress the underlying strings (recall that we reduce the operations on binary relations to rank/select on strings and bit vectors) to high-order entropies. Although there is no standard definition for the entropy of binary relations so that we cannot measure the compression theoretically, we can still achieve much compression in practice.

3.6.2 Binary Relations in Almost Information-Theoretic Minimum Space

Theorem 3.5. *A binary relation R formed by t pairs from an object set $[n]$ and a label set $[\sigma]$ can be represented using $\lg \binom{n\sigma}{t} + t \cdot o(\lg \sigma)$ bits to support:*

- `label_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg \lg \sigma (\lg \lg \lg \sigma)^2)$ time;*
- `label_select` *for any label $\alpha \in [\sigma]$ in $O(\lg \lg \sigma \lg \lg \lg \sigma)$ time;*
- `label_pred` *and* `label_succ` *for any label $\alpha \in [\sigma]$ in $O(\lg \lg \sigma (\lg \lg \lg \sigma)^2)$ time, and these two operations for any literal $\alpha \in [\bar{\sigma}]$ in $O(\lg \lg \sigma)$ time;*
- `object_rank` *and* `label_access` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg \lg \sigma)$ time;*
- `label_nb` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$,* `object_select` *and* `object_nb` *in $O(1)$ time.*

Proof. We construct the bit vector `COLUMNS` as in the proof of Theorem 3.3 using $n + t + o(n+t)$ bits. We also construct another bit vector `BR`, which lists the bits of the conceptual table E (see the proof of Theorem 3.3 for the definition of E) in the column-major order. For the example in Figure 3.2, `BR` $=$ 00111001001011100001. We store `BR` using Lemma 2.2 in $\lg \binom{n\sigma}{t} + o(t) + O(\lg \lg (n\sigma))$ bits.

We now show how to compute $\texttt{object\_select}(x, r)$ in constant time. To answer this query, we need locate the row that contains the r^{th} 1 in the x^{th} column of E. The total number of 1s in columns $1, 2, ..., x-1$ of E is $i = \texttt{bin\_rank}_{\texttt{COLUMNS}}(1, \texttt{bin\_select}_{\texttt{COLUMNS}}(0, x-1))$. Thus, the r^{th} 1 in the x^{th} column of E is the $(r+i)^{\text{th}}$ 1 in `BR`, whose position in `BR` is $j = \texttt{bin\_select}_{\texttt{BR}}(1, r+i)$, which is in the $(j-(x-1)\sigma)^{\text{th}}$ row of E. Hence $\texttt{object\_select}(x, r) = j - (x-1)\sigma$. Therefore, we can support `object_select` in constant time.

With the constant-time support for `object_select`, we can construct a succinct index for R using Theorem 3.3, and the support for the operations listed follows directly.

The overall space cost in bits is $n+t+o(n+t)+\lg\binom{n\sigma}{t}+o(t)+O(\lg\lg(n\sigma))+t\cdot o(\lg\sigma) = \lg\binom{n\sigma}{t} + t\cdot o(\lg\sigma)$, as $t \geq n \geq \sigma$. $\square$

Same as Theorem 3.3, the above approach also works without the assumption that each object is associated with at least one label, though we can not use the inequality $t \geq n$ to analyze the space cost. Thus without such an assumption, our succinct representation occupies $\lg\binom{n\sigma}{t} + t\cdot o(\lg\sigma) + n + o(n)$ bits. This is close to the information-theoretic minimum.

3.7 Discussion

In this chapter, we have designed succinct indexes for strings and binary relation that, given the support for the interface of the ADTs of these data types, support various useful operations efficiently. When the operators in the ADTs are supported in constant time, our results are comparable to previous results, while allowing more flexibility in the encoding of the given data. We also generalized the queries on characters or labels to literals, to support "negative" searches.

Using our techniques, we design a succinct encoding that represents a string of length n over an alphabet of size σ using $nH_k+\lg\sigma\cdot o(n)+n\cdot o(\lg\sigma)$ bits to support access/rank/select operations in $O((\lg\lg\sigma)^{1+\epsilon})$ time, for any fixed constant $\epsilon > 0$. This is the first succinct representation of strings supporting rank/select operations efficiently that occupies space proportional to the high-order entropies of strings. We also design a succinct encoding that represents a binary relation formed by t pairs from an object set $[n]$ and a label set $[\sigma]$ using $\lg\binom{n\sigma}{t} + t\cdot o(\lg\sigma)$ bits to support various types of rank/select operations efficiently. This space cost is close to the information-theoretic minimum.

There are some related open problems. First, we are not certain whether the space costs of our succinct indexes are optimal. Thus one open problem is to prove the lower bounds of succinct indexes of strings and binary relations, or to further improve the results. Second, the term $t\cdot o(\lg\sigma)$ of representing a binary relation in Theorem 3.5 is a second-order term only when $t/n = \sigma^{o(1)}$. Thus it is an open problem to reduce this term.

Chapter 4

Text Indexes

This chapter deals with the problem of designing succinct text indexes to facilitate text search. The chapter starts with an introduction in Section 4.1, followed by a brief review of previous work in Section 4.2, and a summary of the existing results we use in Section 4.3. In Sections 4.4, we present a theorem that characterizes a permutation as the suffix array of a binary string. Based on this theorem, we design succinct text indexes for binary strings in Section 4.5 and Section 4.6. We extend the above results to general alphabets in Section 4.7. Section 4.8 gives some conclusion remarks and suggestions for future work.

4.1 Introduction

As a result of the growth of the textual data in databases, the World Wide Web and applications such as bioinformatics, various indexing techniques have been developed to facilitate full text searching. Given a text string T of length n and a pattern string P of length m, whose symbols are drawn from the same fixed alphabet $[\sigma]$, the goal is to look for the occurrences of P in T. We consider three types of queries: existential queries, cardinality queries, and listing queries. An *existential query* returns a boolean value that indicates whether P is contained in T. A *cardinality query* returns the number, $\mathtt{occ}$, of occurrences of P in T. A *listing query* lists all the positions of occurrences of P in T. We define *pattern searching* to be the process of answering all the above three types of queries for a given pattern string.

Inverted files [59] have been the most popular indexes used in practice. An inverted file is a sorted list (index) of keywords, with each keyword having links to the records containing that keyword in the text [51]. They can be easily adapted to give very efficient indexes for texts that can be naturally parsed into a set of words, such as English text, but not for DNA data or texts in far-eastern languages. Therefore, they are categorized as *word-level indexes*. However, the search for an arbitrary pattern that does not necessarily start at the beginning of a word is inefficient on inverted files.

A suffix tree [84] is a search tree whose leaves correspond (refer) to all the suffixes of the text. The nodes of the tree are placed in lexicographic order of the suffixes to which they refer. The search tree structure enables us to perform a query by searching the suffixes of the text. Because suffix trees index each position in the text, they are categorized as *full text indexes*, and are more powerful than inverted files. Using a suffix tree, we can support existential and cardinality queries of an arbitrary pattern P in text T in $O(m \lg \sigma)$ time. We need additional $O(\mathtt{occ})$ time to answer listing queries. However, a standard representation of a suffix tree requires somewhere between $4n \lg n$ and $6n \lg n$ bits, which is impractical for many applications. Suffix arrays [65, 42] have been proposed to reduce the space cost of suffix trees. The idea is to organize the suffix offsets in a sorted list using the suffixes as sort keys instead of organizing them in a tree, which takes exactly $n \lg n$ bits. With a suffix array, one can answer existential and cardinality queries in $O(m \lg n)$ time, and listing queries in $O(\mathtt{occ})$ extra time. Additional information about the lengths of the (longest) common prefixes of pairs of suffixes of the text can be stored to speed up pattern search. By precomputing and storing such information for $2n - 1$ pairs of suffixes (see [48] for a detailed description of such pairs), one can answer existential and cardinality queries in $O(m + \lg n)$ time, and listing queries in $O(\mathtt{occ})$ extra time. Unfortunately, straightforward representation of such prefix length data takes $(2n - 1) \lg n$ bits. Perhaps as a consequence, suffix arrays are still less popular than inverted lists for large text collections.

The straightforward method to represent a suffix array is to treat it as a permutation of the set of integers $[n]$, the offsets of all the suffixes, and store it in $n \lg n$ bits. However, there are σ^{n-1} different texts of length n drawn from an alphabet of size σ (assume the last character is a special end-of-file symbol not in the alphabet), and so there are σ^{n-1} different suffix arrays associated with them. Therefore, there is a canonical way to represent suffix arrays in $O(n \lg \sigma)$ bits.

In this chapter, we provide a categorization theorem that lets us tell which permutations are suffix arrays and which are not. We further exploit this theorem to design space efficient full-text indexes for fast text searching. We also apply our succinct indexes for strings to make the index scalable for large alphabets, and to compress it.

The storage costs of the text indexes we design in this chapter are at least as much as the space required to encode the texts (or compressed versions of the texts). This saves a lot of space compared with the standard text indexing techniques mentioned in this section. In fact, Demaine and López-Ortiz [24] proved that a text index that supports pattern searching in time linear in the length of the pattern requires space proportional to that of the text. Thus our text indexes are space efficient. Our research also shows that with these succinct text indexes, the original texts no longer need be stored explicitly, which is different from our results on designing succinct indexes for other combinatorial objects. Thus the representations of the results of the succinct text indexes may be slightly different from those of the succinct indexes in other chapters.

4.2 Previous Work

Using some of the techniques of succinct data structures, Grossi and Vitter [45, 46] proposed the compressed suffix array structure, which is the first method that represents suffix arrays drawn from alphabet $[\sigma]$ in $O(n \lg \sigma)$ bits and supports access to any entry of the original suffix array in $O(\log_\sigma^\epsilon n)$ time, for any fixed constant ϵ, where $0 < \epsilon < 1$ (without computing the entire original suffix array). Based on compressed suffix arrays, they designed a full-text index that uses $O(n \lg \sigma)$ bits and answers existential and cardinality queries in $O(\frac{m}{\log_\sigma n} + \log_\sigma^\epsilon n)$ time. Listing queries can be answered in $O(\mathtt{occ} \log_\sigma^\epsilon n)$ additional time. Sadakane [78] proposed additional structures to make the compressed suffix array a *self-indexing data structure*, using which we can retrieve any substring of the text without storing the text itself. His structure uses $O(nH_0 + n)$ bits, where H_0 is the zeroth order entropy of the text, while supporting pattern searching in $O(m \lg n + \mathtt{occ} \lg^\epsilon n)$ time. Retrieving a part of the text of length l starting at any given position costs $O(l + \lg^\epsilon n)$ time. Grossi, Gupta and Vitter [44] further proposed a self-indexing data structure based on compressed suffix arrays that uses $nH_k + o(n) \cdot \lg \sigma$ bits, where H_k is the k^{th} order entropy of the text, while supporting pattern searching in $O(m \lg \sigma + \text{polylog}(n))$ time.

The FM-index [30, 31] proposed by Ferragina and Manzini is based on the Burrows-Wheeler compression [14]. It is a self-indexing data structure that encodes the text (drawn from an alphabet of constant size) in $O(nH_k) + o(n)$ bits, and supports pattern searching in $O(m + \texttt{occ} \lg^{\epsilon} n)$ time. By designing additional data structures to facilitate listing queries, they designed a full-text index that uses $O(nH_k \lg^{\epsilon} n) + o(n)$ bits and supports pattern searching in $O(m + \texttt{occ})$ time [32]. Ferragina, Manzini, Mäkinen and Navarro [34] proposed another variant of the FM-index which occupies $nH_k + O((n \lg\lg n)/\log_{\sigma} n)$ bits, and supports pattern searching in $O(m \lg \sigma + \texttt{occ} \lg \sigma (\lg^2 n / \lg\lg n))$ time.

4.3 Preliminaries

4.3.1 Orthogonal Range Searching on a Grid

Assume that there are n points in an $n \times n$ grid (i.e. the coordinate of each point is in the set $[n] \times [n]$). Given a query range which is a rectangle on the grid (i.e. the range is of the form $[a, b] \times [c, d]$, where $a, b, c, d \in [n]$), the *orthogonal range searching* is to report all the points (x, y) such that $a \leq x \leq b$ and $c \leq y \leq d$. Alstrup *et al.* [1] have the following result on this problem.

Lemma 4.1 ([1])**.** *Given n points in an $n \times n$ grid, there exists a data structure using $O(n \lg^{1+\delta} n)$ bits, for any constant $\delta > 0$, that supports orthogonal range searching in $O(\lg\lg n + k)$ time, where k is the number of the points in the given query range.*

4.3.2 The Burrows-Wheeler Transform

The *Burrows-Wheeler transform* (*BWT*)) was proposed by Burrows and Wheeler [14] to introduce a new class of text compression algorithms. To illustrate the BWT, we give a running example, using the classical text $T[1..n-1] = \textit{mississippi}$ (an example taken from [28]) as the input text. We use $T^{\texttt{BWT}}$ to denote the Burrows-Wheeler transformed string of T. It is performed in three steps:

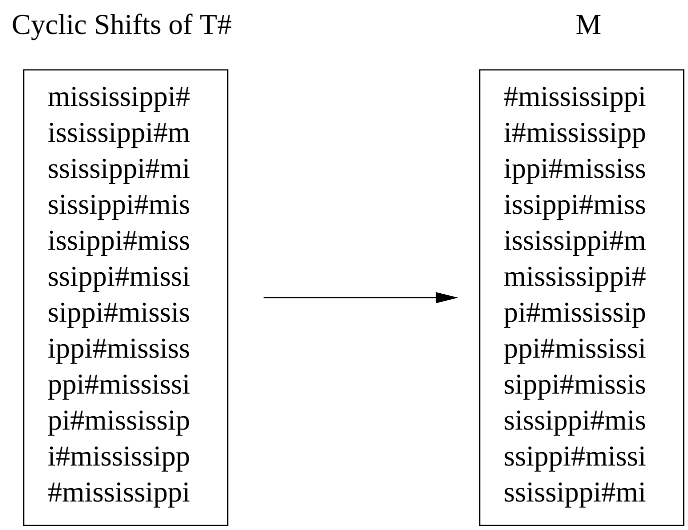


Figure 4.1: Sorting the cyclic shifts of $T\#$ to construct the matrix M for the text $T = mississippi$.

1. Append to the end of T an end-of-file symbol (denoted by #) smaller than any other alphabet symbol.
   In our example, we get $T\# = mississippi\#$.

2. Form a conceptual $n \times n$ matrix M whose elements are symbols, and whose rows are the cyclic shifts of $T\#$, sorted in lexicographic order.
   Please refer to Figure 4.1 for the processing of our example.

3. Return the last column of M, which is the transformed text $T^{\texttt{BWT}}$.
   In our example, $T^{\texttt{BWT}} = ipssm\#pissii$.

Note that the process of sorting the cyclic shifts of $T\#$ is equivalent to the process of sorting suffixes of $T\#$. This is because the symbol # is smaller than any other alphabet symbol, and no character occurring after the symbol # is compared.

The Burrows-Wheeler Transform is not a compression process by itself. However, when combined with other simple compression techniques, it can compress a text string effectively [14].

4.3.3 Compression Boosting

The concept of *compression boosting* [38, 33] was proposed to design BWT-based compression algorithms to achieve good guaranteed compression performance. Given a compression algorithm that can store a string using space proportional to its zeroth order entropy, a compression booster can use it to compress the string in space proportional to its k^{th} order entropy. This is based on the claim that compressing a string up to its k^{th} order entropy can be achieved by optimally partitioning its Burrows-Wheeler transformed string and using a zeroth-order compressor to compress each partition [38]. One variant of the compression boosting technique used in our research was proposed by Ferragina *et al.*:

Lemma 4.2 ([34]). *Consider a compression algorithm A that can store any string S of length p in less than $pH_0(S) + f(p)$ bits, where $f(p)$ is a non-decreasing concave function. Given a text string T of length n drawn from alphabet $[\sigma]$, there is a partition, $S_1, S_2, \cdots, S_z$, of $T^{\mathtt{BWT}}$, such that, for any $k \leq 0$, we have*

$$\sum_{i=1}^{z}(A(S_i)) \leq nH_k(T) + \sigma^k f(n/\sigma^k),$$

where $A(S_i)$ is the number of bits required to store S_i using algorithm A.

This partition can be computed in $O(n)$ time.

4.4 Permutations and Suffix Arrays

In this section, we compare a suffix array with an arbitrary permutation of integers $[n]$. We then present a categorization theorem by which we can determine whether a given permutation is a suffix array of a binary string. Based on the theorem, we give an efficient algorithm that checks whether a permutation is a suffix array of a binary string.

4.4.1 Valid and Invalid Permutations

We adopt the convention that the text T of length n is a string of $n-1$ symbols drawn from the binary alphabet $\Sigma = \{a, b\}$, followed by a special end-of-file symbol #. We assume that

$a < \# < b$. The suffix array SA of T is then a permutation of $[n]$ that corresponds to the lexicographic ordering of the suffixes of T, i.e. the suffix of T that starts at position $SA[i]$ is ranked the i^{th} among all the suffixes in lexicographic order. Based on our convention, there are 2^{n-1} different text strings of length n, so there are at most 2^{n-1} different suffix arrays associated with them. However, there are $n!$ different permutations of $[n]$. Therefore, not all of the $n!$ permutations are suffix arrays. We call those permutations that are suffix arrays *valid permutations*, and those that are not suffix arrays *invalid permutations*. For example, the permutation $4, 7, 5, 1, 8, 3, 6, 2$ is a valid permutation, because it is the suffix array of the text *abbaaba#*, but the permutation $4, 7, 1, 5, 8, 2, 3, 6$ is an invalid one because it is not a suffix array of any text string. Because there are at most 2^{n-1} different suffix arrays of length n, there is a canonical way to represent suffix arrays in $O(n)$ bits. Grossi and Vitter [45] gave the first non-trivial method to represent suffix arrays in $O(n)$ bits and support efficient searching (See Section 4.2). However, they did not provide a method to characterize a permutation as a suffix array, and indeed mentioned it as an open problem. We now address this problem.

4.4.2 A Categorization Theorem

As in Section 3.3.1, if M is a permutation, we denote its inverse by M^{-1}. Hence the inverse permutation of the suffix array SA is SA^{-1}. We find this notation very useful as $M^{-1}[i]$ simply says where i occurs in M, so $M^{-1}[i] < M^{-1}[j]$ simply means i comes before j in the permutation M. We first give two definitions on permutations.

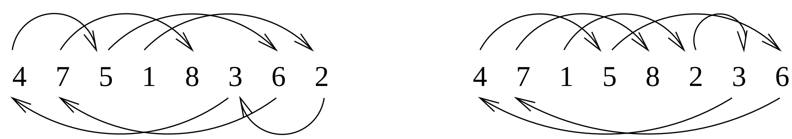
Definition 4.1. *Given a permutation $M[1..n]$ of $[n]$, we call it* **ascending-to-max** *iff for any integer i where $1 \le i \le n-2$, we have:*

(i) if $M^{-1}[i] < M^{-1}[n]$ and $M^{-1}[i+1] < M^{-1}[n]$, then $M^{-1}[i] < M^{-1}[i+1]$, and

(ii) if $M^{-1}[i] > M^{-1}[n]$ and $M^{-1}[i+1] > M^{-1}[n]$, then $M^{-1}[i] > M^{-1}[i+1]$.

Definition 4.2. *Given a permutation $M[1..n]$ of $[n]$, we call it* **non-nesting** *iff for any two integers i, j, where $1 \le i, j \le n-1$ and $M^{-1}[i] < M^{-1}[j]$, we have:*

(i) if $M^{-1}[i] < M^{-1}[i+1]$ and $M^{-1}[j] < M^{-1}[j+1]$, then $M^{-1}[i+1] < M^{-1}[j+1]$, and



(a) a valid permutation (b) an invalid permutation

Figure 4.2: Valid and invalid permutations.

(ii) if $M^{-1}[i] > M^{-1}[i+1]$ *and* $M^{-1}[j] > M^{-1}[j+1]$*, then* $M^{-1}[i+1] < M^{-1}[j+1]$*.*

Figure 4.2 shows the valid and invalid permutations presented in Section 4.4.1. In each we draw an arrow from i to $i+1$ for $i = 1, 2, ..., n-1$, i.e. from position $M^{-1}[i]$ to $M^{-1}[i+1]$, and we denote it as arrow $(i, i+1)$. Arrows pointing to the right (or *right links*) are drawn above the permutations, and arrows pointing to the left (or *left links*) are drawn below the permutations. In an ascending-to-max permutation, all the arrows that do not enclose the maximum value are in the direction that points towards the maximum value in the permutation. In a non-nesting permutation, no arrow encloses another arrow in the same direction. From Figure 4.2, we can see that (a) is both ascending-to-max and non-nesting, but neither is true of (b), because arrow $(2, 3)$ is in the direction away from the maximum value, and right link $(5, 6)$ encloses right link $(2, 3)$.

We can now state our categorization theorem.

Theorem 4.1. *A permutation is a suffix array of a binary string iff it is both ascending-to-max and non-nesting.*

Proof. In this proof, given two strings α and β, we use $\alpha \prec \beta$ ($\alpha \succ \beta$) to denote that string α is lexicographically smaller (larger) than β. First, we prove that a suffix array is ascending-to-max and non-nesting. Assume that we have a suffix array SA of length n. Lemma 4.3 immediately follows from the definition of a suffix array.

Lemma 4.3. *Given an integer* i*, where* $1 \leq i \leq n-1$*, if* $SA^{-1}[i] < SA^{-1}[n]$*, then* $T[i] = a$*. If* $SA^{-1}[i] > SA^{-1}[n]$*, then* $T[i] = b$*.*

To prove the ascending-to-max feature, given an integer i where $1 \leq i \leq n-2$, we first consider the case when $SA^{-1}[i] < SA^{-1}[n]$ and $SA^{-1}[i+1] < SA^{-1}[n]$. By Lemma 4.3,

$T[i] = T[i+1] = a$. Therefore, $T[i,n] = aT[i+1,n] \prec T[i+1,n]$. By the definition of the suffix array, we have $SA^{-1}[i] < SA^{-1}[i+1]$. By similar reasoning, we can prove that if $SA^{-1}[i] > SA^{-1}[n]$ and $SA^{-1}[i+1] > SA^{-1}[n]$, then $SA^{-1}[i] > SA^{-1}[i+1]$. This proves the ascending-to-max feature.

To prove the non-nesting feature, assume we have two integers i, j, where $1 \leq i, j \leq n-1$ and $SA^{-1}[i] < SA^{-1}[j]$. We first consider the case when $SA^{-1}[i] < SA^{-1}[i+1]$ and $SA^{-1}[j] < SA^{-1}[j+1]$. By the definition of the suffix array, we have the following three inequalities: (i) $T[i,n] \prec T[i+1,n]$, (ii) $T[j,n] \prec T[j+1,n]$, and (iii) $T[i,n] \prec T[j,n]$. $T[i] \neq \#$ because $i < n$. We conclude that $T[i] = a$, because otherwise if $T[i] = b$, then $T[i,n] = bT[i+1,n] \succ T[i+1,n]$, which is a contradiction. Similarly, we conclude that $T[j] = a = T[i]$. Because $T[i,n] = aT[i+1,n] \prec T[j,n] = aT[j+1,n]$, the inequality $T[i+1,n] \prec T[j+1,n]$ holds, and the inequality $SA^{-1}[i+1] < SA^{-1}[j+1]$ follows immediately. By similar reasoning, we can prove that if $SA^{-1}[i] > SA^{-1}[i+1]$ and $SA^{-1}[j] > SA^{-1}[j+1]$, then $SA^{-1}[i+1] < SA^{-1}[j+1]$. This proves the non-nesting feature.

Second, we prove that any ascending-to-max and non-nesting permutation is a suffix array. We first describe an algorithm [45] that constructs a text from its suffix array. Given a suffix array SA of length n, we need find its corresponding text T. First, we assign # to $T[n]$. We then scan SA to find the position v such that $SA[v] = n$. By Lemma 4.3, for the i^{th} entry in SA, where $1 \leq i < v$, we assign a to $T[SA[i]]$. For the j^{th} entry in SA, where $v < j \leq n$, we assign b to $T[SA[j]]$.

The above algorithm can construct a text string for any given input permutation M. However, if M is not a suffix array, the suffix array of the text constructed is different from M. We must prove that if M is ascending-to-max and non-nesting, it is the same as the suffix array SA of the constructed text T. Assume that $M[v] = n$. Then in the text string T, there are $(v-1)$ a's and $(n-v)$ b's. In SA, the first $(v-1)$ entries point to suffixes starting with an a, the v^{th} entry points to suffix #, and the last $(n-v)$ entries point to suffixes starting with a b. Therefore, $SA[v] = n = M[v]$. Now we must prove that all the other entries in M and SA are the same. We give a proof by contradiction. First we give the following definition.

Definition 4.3. *A* **reverse pair** *on two given permutations π_1 and π_2 is a pair of integers (i,j), where $1 \leq i, j \leq n$, such that $\pi_1^{-1}[i] < \pi_1^{-1}[j]$ but $\pi_2^{-1}[i] > \pi_2^{-1}[j]$, i.e. the relative* *positions of i and j in π_1 and π_2 are different.*

Assume, contrary to what we are going to prove, that M is different from SA. Then there exists at least one reverse pair on M and SA. We have the following lemma on reverse pairs.

Lemma 4.4. *For any reverse pair (i, j) on M and SA, one of the following two conditions holds:*

(i) $M^{-1}[i] < M^{-1}[j] < v$ *and* $SA^{-1}[j] < SA^{-1}[i] < v$*;*

(ii) $M^{-1}[j] > M^{-1}[i] > v$ *and* $SA^{-1}[i] > SA^{-1}[j] > v$*.*

To prove this lemma, we first consider the case when $M^{-1}[i] < M^{-1}[j] < v$. In this case, according to the algorithm that generates T, we have $T[i] = T[j] = a$. By the definition of suffix arrays, we immediately have $SA^{-1}[j] < SA^{-1}[i] < v$. The case when $M^{-1}[j] > M^{-1}[i] > v$ is similar. We only need consider the case when $M^{-1}[i] < v < M^{-1}[j]$. In this case, $T[i] = a$ and $T[j] = b$, so $SA^{-1}[i] < SA^{-1}[j]$, which is a contradiction. Therefore $M^{-1}[i] < v < M^{-1}[j]$ never holds. □

With this lemma, we can continue the proof of the theorem.

There exists one reverse pair (g, h) such that g is the greatest among the first items of all the reverse pairs. We observe that both g and h are less than n because neither $M^{-1}[g]$ or $M^{-1}[h]$ is v. Therefore, the inequality $1 < g + 1, h + 1 \leq n$ holds. We first consider the case when pair (g, h) satisfies Condition (i) of Lemma 4.4. In this case, we observe that $M^{-1}[g] < M^{-1}[g+1]$ and $M^{-1}[h] < M^{-1}[h+1]$, because otherwise, M is not ascending-to-max. Because M is non-nesting, we have $M^{-1}[g+1] < M^{-1}[h+1]$. By similar reasoning, we can prove that $SA^{-1}[g+1] > SA^{-1}[h+1]$, as SA is also ascending-to-max and non-nesting. Now we have another reverse pair $(g+1, h+1)$. Its first item $(g+1)$ is greater than g, which is a contradiction. We can reach a contradiction by similar reasoning for the case when pair (g, h) satisfies Condition (ii) of Lemma 4.4. □ (Thm 4.1)

We have the following corollary.

Corollary 4.1. *For a text string T over alphabet $\{a, b\}$, if its longest run of a's is of length l_1, and its longest run of b's is of length l_2, then its suffix array SA can be divided into $l_1 + l_2 + 1$ segments numbered $1, 2, ..., l_1 + l_2 + 1$, such that:*

(i) suffixes corresponding to the entries in segments $1, 2, ..., l_1$ *are prefixed with* $l_1, l_1 - 1, ..., 1$ *a's followed by b or #, respectively, and the right links in segment i point to elements in segment* $(i + 1)$*, for* $1 \leq i \leq l_1 - 1$*;*

(ii) segment $(l_1 + 1)$ *only has one entry, n;*

(ii) suffixes corresponding to the entries in segments $l_1 + 2, l_1 + 3, ..., l_1 + l_2 + 1$ *are prefixed with* $1, 2, ..., l_2$ *b's followed by a or #, respectively, and the left links in segment j point to elements in segment* $(j - 1)$*, for* $l_1 + 3 \leq j \leq l_1 + l_2 + 1$*.*

Proof. By the definition of suffix arrays, we can divide SA into $l_1 + l_2 + 1$ segments in the above way. Assume that a right link starts from the j^{th} entry, which is in segment i $(1 \leq i \leq l_1 - 1)$. Then $T[SA[j]]$ is prefixed with $l_1 - i + 1$ a's, so $T[SA[j] + 1]$ is prefixed with $l_1 - i$ a's. Hence the $(SA^{-1}[SA[j] + 1])^{\text{th}}$ entry of SA is in segment $i + 1$, and this is the entry that the right link points to. Similarly, we can prove that the left links in segment j point to elements in segment $(j - 1)$, for $l_1 + 3 \leq j \leq l_1 + l_2 + 1$. □

4.4.3 An Efficient Algorithm to Check Whether a Permutation is Valid

The proof of Theorem 4.1 suggests a method to determine whether a permutation is a suffix array. We first construct a text string from the permutation by the method in the proof, and then construct the suffix array of the text. If the suffix array constructed is the same as the permutation, then the permutation is a suffix array. Otherwise, it is not. This algorithm takes $O(n)$ time and $O(n)$ words of memory, because the construction of the text string and the suffix array, and the comparison all cost $O(n)$ time and space. However, the constants hidden in the big-oh notation for suffix array construction algorithms are large [57, 60], and these algorithms are hard to implement.

We have the following theorem on efficiently testing whether a permutation is valid.

Theorem 4.2. *There is an algorithm that can test whether a permutation* M *of* $[n]$ *is a suffix array of a binary string in* $O(n)$ *time using* $n + o(n)$ *words of working space.*

Algorithm `Check`(M)

1. Scan M to compute M^{-1}.
2. Scan M^{-1} to check whether M is ascending-to-max.
3. Check Condition (i) of the non-nesting feature by scanning M from the beginning. At the i^{th} step, compute $M^{-1}[M[i]+1]$. If $M^{-1}[M[i]+1] > i$, then keep the value. If M satisfies the condition, the sequence of values computed and kept at each step is ascending.
4. Similarly, check Condition (ii) of the non-nesting feature.

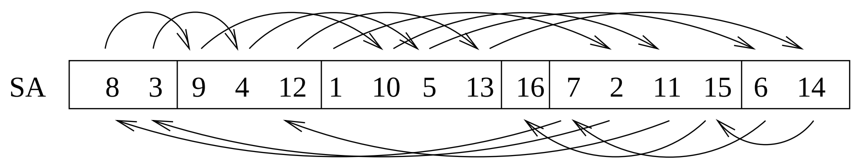
Figure 4.3: An algorithm to check whether a permutation is a suffix array.

Proof. Figure 4.3 shows a simple algorithm that determines whether a permutation is a suffix array of a binary string using the characterization of Theorem 4.1. Each phase takes $O(n)$ time and the algorithm only needs $n+O(1)$ additional words of memory to store M^{-1} and some other temporary results, which is roughly the same as the size of the input. □

A more restricted problem is studied by Burkhardt and Kärkkäinen [13]. They propose a linear time algorithm to test whether a permutation is the suffix array of a given text string.

4.5 Space Efficient Suffix Arrays Supporting Cardinality Queries

We now explore Theorem 4.1 and Corollary 4.1 to design a space efficient full-text index. Figure 4.4 shows the suffix array for the text $abaaabbaaabaabb\#$. We divide the suffix array into 6 segments using Corollary 4.1 and draw arrows as in Section 4.4.2. Each arrow links a suffix to the suffix whose starting position is one character behind, i.e. each arrow is from position $SA^{-1}[i-1]$ in the suffix array to position $SA^{-1}[i]$, for $i = 2, 3, ..., n$. For each position $SA^{-1}[i]$ in the suffix array, we consider the position $SA^{-1}[i-1]$. From the



B_a	0	0	1	1	0	0	1	1	1	0	0	1	1	0	1	1
B_b	1	1	0	0	1	0	0	0	0	1	1	0	0	1	0	0

Figure 4.4: An example of our data structures over the text *abaaabbaaabaabb#*.

arrows and Corollary 4.1, we observe that $SA^{-1}[i-1]$ is either in the last segment before position $SA^{-1}[i]$ whose corresponding suffixes start with a, if $T[i-1] = a$, or in one of the segments whose corresponding suffixes start with b or #. We design a text index based on such information.

Theorem 4.3. *Given a binary text string T of length n, there is an index structure using $n + o(n)$ bits that answers, without storing the raw text, existential and cardinality queries on any pattern string P of length m in $O(m)$ time.*

Proof. We use SA to denote the suffix array of T, and we construct a bit vector B_a of size n as follows. For $i > 1$, if $T[i-1] = a$, we store a 1 in $B_a[SA^{-1}[i]]$, and we store a 0 otherwise. We set $B_a[SA^{-1}[1]] = 0$. We conceptually define the analogous bit vector B_b with the value $B_b[SA^{-1}[i]] = 1$ iff $T[i-1] = b$ for $i > 1$, and $B_b[SA^{-1}[1]] = 0$. Clearly B_b is the complement of B_a except in position $SA^{-1}[1]$, where they are both 0. Each of these bit vectors, in fact, stores the information of the Burrows-Wheeler transform (see Section 4.3.2) of the text T [14]. (See Figure 4.4.)

We build rank structures over B_a using part (a) of Lemma 2.1. From this data on B_a, and by storing $SA^{-1}[1]$, we can also perform rank queries on B_b in constant time. However, to explain our algorithm, we retain the notion of two bit vectors. We also store the number of a's in an integer n_a. The bit vector B_a with corresponding rank structures, n_a, and $SA^{-1}[1]$ are our main indexing data structures, which together use $n + o(n)$ bits.

Figure 4.5 gives an algorithm for answering existential and cardinality queries using the above data structures. This algorithm starts from the end of the pattern P and, at each phase of the loop, computes the interval $[s, e]$ of SA whose corresponding suffixes are

Algorithm $\texttt{Count}(T, P)$

$s \leftarrow 1$, $e \leftarrow n$, $i \leftarrow m$
while $i > 0$ and $s \leq e$ **do**
  if $P[i] = a$ **then**
    $s \leftarrow \texttt{bin\_rank}_{B_a}(1, s-1) + 1$, $e \leftarrow \texttt{bin\_rank}_{B_a}(1, e)$
  else
    $s \leftarrow n_a + 2 + \texttt{bin\_rank}_{B_b}(1, s-1)$, $e \leftarrow n_a + 1 + \texttt{bin\_rank}_{B_b}(1, e)$
  $i \leftarrow i - 1$
return $\max(e - s + 1, 0)$

Figure 4.5: An algorithm for answering existential and cardinality queries.

prefixed with $P[i, m]$. To show the correctness of the algorithm, we need show that we update the values of s and e correctly. Assume that at the beginning of phase $m - i + 1$, the interval $[s, e]$ of SA corresponds to suffixes that are prefixed with $P[i+1, m]$. Assume, without loss of generality, that $P[i] = a$. The entries of SA corresponding to suffixes that start with a occupy the interval $[1, n_a]$. Because all such suffixes start with the same character a, they are sorted according to the suffixes whose starting positions are one character after them. Therefore, the lexicographically smallest suffix prefixed by $P[i, m]$, and the lexicographically smallest suffix prefixed by $P[i+1, m]$ that follows character a, are one character apart in T by their starting positions. On the other hand, because $B_a[SA^{-1}[i]] = 1$ when $T[i-1] = a$, $\texttt{bin\_rank}_{B_a}(1, s-1)$ computes how many suffixes smaller than $P[i+1, m]$ in lexicographic order follow character a in the original text T. Therefore, $\texttt{bin\_rank}_{B_a}(1, s-1) + 1$ points to the lexicographically smallest suffix that starts with $P[i, m]$. A similar analysis applies to e. Therefore, our algorithm is correct. The runtime is clearly $O(m)$.[1] $\square$

[1]Our algorithm is similar to the backward search algorithm of the FM-index [30]. An anonymous reviewer of [52] commented that this result could also have been proved by combining the backward search of FM-index [30] and wavelet trees [44].

4.6 Space Efficient Self-indexing Suffix Arrays Supporting Listing Queries

4.6.1 Locating Multiple Occurrences

Lemma 4.5. *Using an auxiliary data structure of $\gamma n + o(n)$ bits, for any $0 < \gamma < 1$, the index structure in Theorem 4.3 can list all the occurrences in $O(\texttt{occ} \lg n)$ additional time.*

Proof. We now apply the techniques developed by Ferragina and Manzini [30] to support listing queries using our index structure.

To perform listing queries, we first show that given a position i in the original text T, if we know $SA^{-1}[i]$, we can compute $SA^{-1}[i-1]$ in constant time. We claim that if $B_a[SA^{-1}[i]] = 1$, then $SA^{-1}[i-1] = \texttt{bin\_rank}_{B_a}(1, SA^{-1}[i])$, and if $B_b[SA^{-1}[i]] = 1$, then $SA^{-1}[i-1] = n_a + 1 + \texttt{bin\_rank}_{B_b}(1, SA^{-1}[i])$. To prove this claim, we assume, without loss of generality, that $B_a[SA^{-1}[i]] = 1$. By the definition of B_a, we have that $T[i-1] = a$. Hence $T[i-1, n] = aT[i, n]$. We observe that $\texttt{bin\_rank}_{B_a}(1, SA^{-1}[i])$ computes how many suffixes smaller than or equal to $T[i, n]$ in lexicographic order follow character a in the original text T. As the suffixes whose starting positions are one character ahead of the above suffixes are the suffixes of T that are smaller than or equal to $aT[i, n] = T[i-1, n]$ in lexicographic order, we conclude that $SA^{-1}[i-1] = \texttt{bin\_rank}_{B_a}(1, SA^{-1}[i])$.

Now we describe our auxiliary data structure supporting listing queries. As shown above, we can go backward in the text character by character in constant time. We explicitly store every position of the original text that is of the form $i\lceil \lg n/\gamma \rceil + 1$, for $i = 0, 1, ..., n/\lceil \lg n/\gamma \rceil - 1$ (assume that n is a multiple of $\lceil \lg n/\gamma \rceil$ for simplicity), and organize them in an array S sorted by lexicographic order of the suffixes starting at these positions. We use an additional bit vector F of length n to indicate whether a given entry in SA points to a position that is stored in S. With S and F, we can retrieve the occurrences. Recall that in Algorithm `Count`, we compute the interval $[s, e]$ of SA in which the entries point to the actual positions of all the occurrences of P in T. For each $i \in [s, e]$, we need find $SA[i]$. Figure 4.6 gives an algorithm for retrieving $SA[i]$. In this algorithm, we check whether $F[i]$ is 1. If it is, then $S[\texttt{bin\_rank}_F(1, i)]$ is the answer. If it is not, we go backward in the text one step at a time. In each step, we find the index of the suffix

Algorithm $\texttt{Retrieve}(T, i)$

$j \leftarrow 0$
while $F[i] \neq 1$ **do**
  $i \leftarrow \texttt{Backward}(T, i)$
  $j \leftarrow j + 1$
return $S[\texttt{bin\_rank}_F(1, i)] + j$

Algorithm $\texttt{Backward}(T, i)$

if $B_a[i] = 1$ **then**
  $i \leftarrow \texttt{bin\_rank}_{B_a}(1, i)$
else
  $i \leftarrow n_a + 1 + \texttt{bin\_rank}_{B_b}(1, i)$
return i

Figure 4.6: An algorithm for retrieving an occurrence.

array entry that points to the position one character before the current position. We stop when we reach a position that is stored in S according to F, retrieve the position from S, and the answer is the position retrieved plus the number of steps we go backward in the text.

Array S uses at most γn bits because it has $n/\lceil \lg n/\gamma \rceil$ entries and each of them uses $\lg n$ bits. We use part (b) of Lemma 2.1 to store F, which uses $\lg \binom{n}{n/\lceil \lg n/\gamma \rceil} + o(n) = O(n \lg\lg n/\lg n) + o(n) = o(n)$ bits. Because we store every $\lceil \lg n/\gamma \rceil^{\text{th}}$ position of the original text, we need go backward at most $\lceil \lg n/\gamma \rceil$ number of steps to locate each occurrence. As each of the operations of going backward, rank and accessing any entry in F and S costs constant time, we need $O(\lg n)$ time to locate an occurrence. $\square$

When `occ` is large, retrieving all the occurrences is costly. We design additional approaches to speed up the reporting of occurrences in Sections 4.6.3 and 4.6.4.

4.6.2 Self-Indexing and Context Reporting

We now show how to make our data structures self-indexing.

Lemma 4.6. *Using an auxiliary data structure of ηn bits, for any $0 < \eta < 1$, without storing the text, the index structure in Theorem 4.3 can output a substring of length l that starts at a given position in the text in $O(l + \lg n)$ time.*

Proof. We make use of the property that the first n_a suffix array entries correspond to suffixes starting with a, the (n_a+1)'st entry corresponds to suffix #, and the rest correspond to suffixes starting with b. Therefore, we can output the substring $T[i, i+l-1]$ (without retrieving T) by locating the suffix array entry that points to each position in the substring. To do this, we use an array V to store, for every $(\lceil \lg n/\eta \rceil)^{\text{th}}$ position in T, the index of its corresponding entry in SA, sorted by its position in the text. Array V uses $\frac{n}{\lceil \lg n/\eta \rceil} \times \lg n \leq \eta n$ bits. Given the query to retrieve the substring $T[i, i+l-1]$, we locate the first position in T after and including position $j \geq i + l - 1$, such that the index of its corresponding entry in SA is stored in V. To ensure that such a j always exists, we always store position n in V. From V, we can retrieve the index of the suffix array entry that corresponds to position j in T in constant time. We can now output T[j]. We then use the method in the proof of Theorem 4.5 to walk backward in the text. At each step, we compute the index of the suffix array entry that corresponds to a position in substring $T[i, j]$ and output a character according to it. We repeat until we output the string $T[i, j]$ in reverse order, from which we have the string $T[i, i+l-1]$.

Because we store the suffix array index for every $\lceil \lg n/\eta \rceil^{\text{th}}$ position in T, we have $i + l - 1 \leq j \leq i + l + \lceil \lg n/\eta \rceil - 2$. Therefore, the above process outputs a substring of length l using $O(l + \lg n)$ time. □

4.6.3 Speeding up the Reporting of Occurrences of Long Patterns

Based on an idea in [32, 46], we show how to reduce the problem of reporting occurrences of long patterns to orthogonal range queries on a two-dimensional grid and solve it efficiently.

Lemma 4.7. *Using an auxiliary data structure of $n + o(n)$ bits, the index structure in Theorem 4.3 can support pattern searching on any pattern string P of length m in $O(m + \mathsf{occ})$ time, when $m = \Omega(\lg^{1+\mu} n)$, for any μ where $0 < \mu < 1$.*

Proof. We use T' to denote the reverse of T, so $T' = T[n]T[n-1]...T[1]$. We build a suffix array for T' and denote it SA'. For any μ' and c, where $0 < \mu' < 1$ and $c > 0$, let $d = c\lg^{1+\mu'} n$. We mark every position in T that is a multiple of d. For simplicity, we assume that n is a multiple of d. Then the i^{th} marked position is position id, for $i = 1, 2, ..., n/d$. For the i^{th} marked position, let $s = id$, which is its position in T. Let $x_i = SA^{-1}[s]$, which is the index of the entry of SA that corresponds to suffix $T[s, n]$. For the substring $T[1, s-1]$ that appears before position s, its corresponding suffix in the reverse text is $T'[n-s+2, n]$. Let $y_i = SA'^{-1}[n-s+2]$, which is the index of its corresponding entry in SA'. We now have a set of pairs $Q = \{(x_1, y_1), (x_2, y_2), ..., (x_{n/d}, y_{n/d})\}$. It is obvious that all the x_i's and y_i's are different from each other, so the set Q corresponds to n/d points on an $n \times n$ grid. We first observe the following.

Observation. *Given a pattern P whose length is at least d, for any given occurrence of P in T, there exists one and only one j, where $1 \le j \le d$, such that the position of the j^{th} character in this occurrence is marked.*

From this, we observe that for $j = 1, 2, ..., d$, if we can report all the occurrences of P whose j^{th} character is located at a marked position, we can report all the occurrences of P in T. To report such occurrences for a given j, we first use Algorithm `Count` (Figure 4.5) to retrieve the interval $[i_1, i_2]$ in SA in which all the entries correspond to suffixes of T that start with $P[j]P[j+1]...P[m]$, and the interval $[i_3, i_4]$ in SA' in which all the entries correspond to suffixes of T' that start with $P[j-1]P[j-2]...P[1]$. Let $i_3 = 1$ and $i_4 = n$ when $j = 1$. Now the problem has been reduced to an orthogonal range searching over n/d points in an $n \times n$ grid: we need find all the points (x_i, y_i) in Q such that $i_1 \le x_i \le i_2$ and $i_3 \le y_i \le i_4$. For any point (x_i, y_i) returned, its corresponding marked position in the text is id. There exists an occurrence of P whose j^{th} character is located at the above position. Hence we return $id - j + 1$ as the position of the occurrence.

By Lemma 4.1, we can answer range queries over n/d points on an $n/d \times n/d$ grid in $O(\lg\lg n + k)$ time (k is the size of the answer), using $O((n/d)\lg^{1+\delta} n) = O(n/\lg^{\mu'-\delta} n) = o(n)$ bits, for any δ that satisfies $0 < \delta < \mu' < 1$. However, we need perform range queries over n/d points in an $n \times n$ grid. We use the following approach based on the reduction algorithm in Section 2.2 of [1].

Construct a bit vector X of length n, in which $X[i] = 1$ iff there exists an integer

p such that $x_p = i$. Because there are n/d points in Q, and all the x_i's are different from each other, there are exactly d 1s in X. Thus we can store X using part (b) of Lemma 2.1 in $\lg \binom{n}{n/d} + o(n) = O(n \lg\lg n / \lg^{1+\mu'} n) + o(n) = o(n)$ bits. Similarly, we construct a bit vector Y of length n, in which $Y[i] = 1$ iff there exists an integer l such that $y_l = i$, and store it in $o(n)$ bits using the same approach. We then construct a set of points Q' on an $n/d \times n/d$ grid as follows. For any point (x_i, y_i) in Q, we store in Q' a point $(x'_i, y'_i) = (\texttt{bin\_rank}_X(1, x_i), \texttt{bin\_rank}_Y(1, y_i))$. There is a one-to-one correspondence between the points in Q and the points in Q'; given a point (x'_i, y'_i) in Q', we can compute its corresponding point (x_i, y_i) in Q in constant time, because $(x_i, y_i) = (\texttt{bin\_select}_X(1, x'_i), \texttt{bin\_select}_Y(1, y'_i))$. We store Q' in $o(n)$ bits using the approach described in the previous paragraph to support orthogonal range searching in $O(\lg\lg n + k)$ time. To support orthogonal range search over Q, assume that the given query range is $[a, b] \times [c, d]$. We first map this range to a range $[a', b'] \times [c', d']$ on an $n/d \times n/d$ grid where Q' is in. By the definition of Q', we observe that if $X[a] = 1$, then $a' = \texttt{bin\_rank}_X(1, a)$. Otherwise, $a' = \texttt{bin\_rank}_X(1, a) + 1$. Similarly, we have that $b' = \texttt{bin\_rank}_X(1, b)$. We compute c' and d' using bit vector Y in the same way. We then retrieve the points in Q' that are in the range $[a', b'] \times [c', d']$, and locate their corresponding points in the set Q. Therefore, we can answer range queries over Q in $O(\lg\lg n + k)$ time using data structures occupying $o(n)$ bits.

To analyze the space cost of the data structures we construct in this proof, the set Q is preprocessed in the above data structures using $o(n)$ bits. The index structures constructed over T' using Theorem 4.3 occupies additional $n + o(n)$ bits. Therefore, our auxiliary data structures occupy $n + o(n)$ bits. To efficiently retrieve the occurrences, instead of using Algorithm `Count` for each j, where $1 \le j \le d$, we use it once on P over T, because during the execution of the algorithm, for each suffix $P[i, m]$ of P, we need compute the interval of suffix array whose entries correspond to all the suffixes that start with $P[i, m]$. It is the same with the reverse of P. This requires an additional working space of $O(m)$ words. Therefore, in $O(m)$ time, we can retrieve all the intervals required. We need perform d range queries, which cost $O(\lg^{1+\mu'} n \lg\lg n + \texttt{occ}) = O(\lg^{1+\mu} n + \texttt{occ})$ time, for any μ such that $0 < \mu' < \mu < 1$. Thus we can perform pattern searching in $O(m + \lg^{1+\mu} n + \texttt{occ}) =$ O(m+ `occ`) time when $m = \Omega(\lg^{1+\mu} n)$. $\square$

Combined with Theorem 4.3, Lemma 4.5, and Lemma 4.6, we have:

Theorem 4.4. *Given a binary text string T of length n, there is an index structure using $n+o(n)$ bits that supports, without storing the raw text, for any pattern string P of length m,*

(i) pattern searching in $O(m + \mathsf{occ} \lg n)$ time using an additional $\gamma n + o(n)$ bits, for any $0 < \gamma < 1$.

(ii) when $m = \Omega(\lg^{1+\mu} n)$, for any μ where $0 < \mu < 1$, pattern searching in $O(m + \mathsf{occ})$ time using an additional $n + o(n)$ bits;

This data structure also supports the output of a substring of length l in $O(l + \lg n)$ time using an additional ηn bits, for any $0 < \eta < 1$.

4.6.4 Listing Occurrences in $O(\mathsf{occ} \lg^{\lambda} n)$ Additional Time Using $O(n)$ Bits

In this section, we give another implementation of our index structure that uses $O(n)$ bits and supports listing queries in $O(m + \mathsf{occ} \lg^{\lambda} n)$ time for any λ such that $0 < \lambda < 1$ by designing auxiliary structures to speed up the reporting of occurrences.

Lemma 4.8. *Given a binary text string T of length n, for any λ such that $0 < \lambda < 1$, there is an index structure using O(n) bits that answers existential and cardinality queries on any pattern P of length m in O(m) time, and listing queries in additional $O(\mathsf{occ} \lg^{\lambda} n)$ time. This data structure also supports the output of a substring of length l in $O(l/\lg n)$ time.*

Proof. To illustrate the approach, we take $\lambda = 1/2$. Let $g = \lceil \sqrt{\log_2 n} \rceil$. In this case, we mark every position of the text T that is of the form $1 + ig$, for $i = 0, 1, ..., n/g - 1$ (assume n is a multiple of g for simplicity). We use a bit vector G in which the j^{th} bit is 1 iff the j^{th} entry in SA points to a marked position, and we store G using part (b) of Lemma 2.1.

We construct a text string T^* of length n/g drawn from the alphabet $\Sigma' = \{0, 1, ..., 2^g - 1\}$, in which symbol α corresponds to the α^{th} smallest binary string of length g in lexicographic order. We generate T^* by replacing every substring of length g in T that starts at a marked position by the corresponding symbol in Σ'. We also retain an array C that stores the prefix sum of the vector of frequencies of the characters (binary strings of length g) in T^*. That is, for each character α, we count the number of occurrences of the characters $0, 1, ..., \alpha - 1$ in T^*, and store this value in $C[\alpha]$. For each alphabet symbol α in Σ', we construct a bit vector B_α in which $B_\alpha[SA^{*-1}[i]] = 1$ iff $T^*[i-1] = \alpha$ for $i > 1$, and $B_\alpha[SA^{*-1}[1]] = 0$. We store B_α using Lemma 2.2. We use an array Z to store, for each position in SA^* except position $SA^{*-1}[1]$, the symbol that precedes the suffix it points to in T^*.

We claim that, for a given position $SA^{*-1}[i]$ in SA^*, if $Z[SA^{*-1}[i]] = \alpha$, then $SA^{*-1}[i-1] = C[\alpha] + \texttt{bin\_rank}'_{B_\alpha}(1, SA^{-1}[i])$. To prove this claim, we assume that $Z[SA^{*-1}[i]] = \alpha$. Then $T^*[i-1] = \alpha$. Hence $T^*[i-1, n] = \alpha T^*[i, n]$. We observe that $\texttt{bin\_rank}_{B_\alpha}(1, SA^{*-1}[i])$ computes how many suffixes smaller than or equal to $T^*[i, n]$ in lexicographic order follow character α in text T^*. The suffixes smaller than or equal to $T^*[i-1, n]$ in lexicographic order can be categorized into two types. The first type includes suffixes whose first character is smaller than α in lexicographic order, and the number of such suffixes is stored in $C[\alpha]$. The second type includes suffixes that are prefixed with α, followed by suffixes smaller than or equal to $T^*[i, n]$ in lexicographic order. The number of the suffixes of the second type is $\texttt{bin\_rank}_{B_\alpha}(1, SA^{*-1}[i])$ as computed above. Hence we conclude that $SA^{*-1}[i-1] = C[\alpha] + \texttt{bin\_rank}'_{B_\alpha}(1, SA^{*-1}[i])$. With this claim, we can go backwards in T^* by one position in constant time.

We build another set of data structures. We explicitly store every position of the original text T that is of the form $ig^2 + 1$, for $i = 0, 1, ..., \lceil n/g^2 \rceil - 1$, and organize them in an array S' sorted by lexicographic order of the suffixes starting at these positions. We use an additional bit vector F' of length n to indicate whether a given entry in SA points to a position that is stored in S'. Finally, we observe that, every g^{th} position in T^* corresponds to a position stored in S', as these positions are of the form $ig^2 + 1$. We store another bit vector W using part (b) of Lemma 2.1, in which $W[i] = 0$ iff $SA^*[i]$ points to a position in T^* that corresponds to a position stored in S'.

Figure 4.7 shows our algorithm, in which $\texttt{Backward}(T, i)$ is the algorithm in Section 4.6.1 that enable us to find $SA^{-1}[SA[i] - 1]$ for a given T and an index i of a suffix array entry (see Figure 4.6). Given the index of an entry in SA that points to an occurrence of P, we

Algorithm $\texttt{Retrieve2}(T, i)$

$s_1 \leftarrow 0$
while $G[i] \neq 1$ **do**
  $i \leftarrow \texttt{Backward}(T, i)$
  $s_1 \leftarrow s_1 + 1$
$j \leftarrow \texttt{bin\_rank}_G(1, i)$
$s_2 \leftarrow 0$
while $W[j] \neq 1$ **do**
  $j \leftarrow \texttt{Backward}^*(T^*, j)$
  $s_2 \leftarrow s_2 + 1$
$k \leftarrow \texttt{bin\_select}_G(1, j)$
return $S'[\texttt{bin\_rank}_{F'}(1, k)] + s_1 + s_2 g$

Figure 4.7: An algorithm for retrieving an occurrence in $O(\sqrt{\lg n})$ time.

first check whether it points to a marked position using G. If it does not, we can find the closest marked position that precedes it by going backwards in T at most g times using `Backward`. When we reach a marked position pointed to by the i^{th} entry of SA, the index of its corresponding entry in SA^* is $\texttt{bin\_rank}_G(1, i)$. We check whether it corresponds to a position stored in S' using W. If not, we use the method described above (we call it $\texttt{Backward}^*$) to go backwards in T^*, at most g times, until we reach a position of T^* that corresponds to a position stored in S'. Assume that the j^{th} entry of SA^* points to the above position. It corresponds to the $k = \texttt{bin\_select}_G(1, j)^{\text{th}}$ entry of SA. We then retrieve $S[\texttt{bin\_rank}_{F'}(1, k)]$. Let s be the retrieved position. Assume that we go backwards s_1 steps to reach a marked position, and then another s_2 steps to reach a position stored in S, then the occurrence is $s + s_1 + s_2 g$.

As the above procedure calls `Backward` or $\texttt{Backward}^*$ at most g times, it takes $O(g) = O(\sqrt{\lg n})$ time. G uses $\lg \binom{n}{n/g} + o(n) = O(n \lg \lg g/g) + o(n) = o(n)$ bits. C uses $2^g \lg n = o(n)$ bits. W uses $\lg \binom{n/g}{\lceil n/g^2 \rceil} + o(n/g) = o(n)$ bits. Array Z uses $\frac{n}{g} \times g = n$ bits. S' uses $\lceil n/g^2 \rceil \times \lg n \leq n \lg n / \log_2 n = n + o(n)$ bits. F' uses $\lg \binom{n}{\lceil n/g^2 \rceil} + o(n) = o(n)$ bits. Hence G, W, Z, S' and F' use $2n + o(n)$ bits in total. We do not explicitly store T^* or SA^*. To analyze the space cost of all the B_α's, we make use of Stirling's formula: $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + O(\frac{1}{n}))$.

With this, we have $\log_2 n! = n\log_2 n - n\log_2 e + \frac{1}{2}\log_2 n + O(1)$. Assume that symbol α occurs n_α times in T^*. Let $l = n/g$. Then B_α uses $\lg\binom{l}{n_\alpha} + o(n_\alpha) + O(\lg\lg n)$ bits. We rewrite the first term into:

$$\begin{aligned}
\lg\binom{l}{n_\alpha} &< \log_2\binom{l}{n_\alpha} + 1 \\
&= \log_2 l! - \log_2 n_\alpha! - \log_2(l-n_\alpha)! + 1 \\
&= l\log_2 l - l\log_2 e - n_\alpha\log_2 n_\alpha + n_\alpha\log_2 e - (l-n_\alpha)\log_2(l-n_\alpha) \\
&+ (l-n_\alpha)\log_2 e + \frac{1}{2}(\log_2 l - \log_2 n_\alpha - \log_2(l-n_\alpha)) + O(1) \\
&= l\log_2 l - n_\alpha\log_2 n_\alpha - (l-n_\alpha)\log_2(l-n_\alpha) + O(1) \\
&= (l-n_\alpha)\log_2 l + n_\alpha\log_2 l - n_\alpha\log_2 n_\alpha - (l-n_\alpha)\log_2(l-n_\alpha) + O(1) \\
&= n_\alpha\log_2\frac{l}{n_\alpha} + (l-n_\alpha)\log_2\frac{l}{l-n_\alpha} + O(1) \qquad (4.1)
\end{aligned}$$

To analyze the second term of equation 4.1, we rewrite it into $n_\alpha\log_2(1+\frac{n_\alpha}{l-n_\alpha})^{\frac{l-n_\alpha}{n_\alpha}}$. By the definition of e, this term is less than or equal to $n_\alpha\log_2 e$. With equation 4.1, we have:

$$\lg\binom{l}{n_\alpha} < n_\alpha\log_2\frac{el}{n_\alpha} + O(1) \qquad (4.2)$$

Note that equation 4.1 is true even for the special case when $n_\alpha = 0$, if we follow the interpretation that $0\log_2 0 = 1$ used in Definition 2.1. Therefore, we have $B_\alpha < n_\alpha\log_2\frac{el}{n_\alpha} + o(n_j) + O(\lg\lg n)$. When we compute the total space cost of all the B_α's, the last two items of right hand side of this inequality clearly sum up to $o(n)$. The first item sums up to $nH_0(T^*)/g + (\lg e)n/g = nH_0(T^*)/g + o(n) \le n + o(n)$. Therefore, the B_α's use at most $n + o(n)$ bits together. Hence all the auxiliary data structures use at most $3n + o(n)$ bits.

For an arbitrary λ, we design additional data structures of $\lceil\lambda^{-1}\rceil - 1$ levels. Let $p = \lceil\log_2^\lambda n\rceil$. At each level, we group p^i bits to construct a string drawn from an alphabet of size 2^{p^i}, for $i = 1, 2, ..., \lceil\lambda^{-1}\rceil - 1$. We design similar data structures and search algorithms as described above. Data structures at each level occupy $2n + o(n)$ bits, and we store every $(p^{\lceil\lambda^{-1}\rceil})^{\text{th}}$ position of T using data structures occupying at most $n + o(n)$ that are similar to S' and F'. With these data structures, we can answer listing queries using $O(m + \mathtt{occ}\lg^\lambda n)$ time. The overall data structures occupy $(2\lceil\lambda^{-1}\rceil - 1)n + o(n)$ bits[2].

[2]This multi-level tradeoff is similar to the multi-level compressed suffix array by Grossi and Vitter [45].

To output a substring of size l, we simply store T explicitly in n bits and output the substring word by word (i.e. $\Theta(\lg n)$ bits each time). $\square$

4.6.5 Speeding up the Existential and Cardinality Queries of Short Patterns

Another technique can be used to support existential and cardinality queries for patterns of length at most $\lg n$ in $O(1)$ time using $2n + o(n)$ bits of space, either with or without any of our index structures.

Lemma 4.9. *Given a binary text string T of length n, there is a data structure using $2n + o(n)$ bits that can answer existential and cardinality queries on any pattern P of length m in constant time, when $m \leq \lg n$.*

When combined with the index structure in Theorem 4.4, it can answer existential and cardinality queries on any pattern P of length m in $O(m - \lg n)$ time, when $m > \lg n$.

Proof. Construct a bit vector of length $2n$ which has 1s corresponding to all the suffix array entries and 0s corresponding to all possible patterns of length $\lg n$ in the positions where they "fit" in the suffix array (i.e. for a pattern A of length $\lg n$, the suffix array entries to the left of A correspond to suffixes whose prefixes of length $\lg n$ are lexicographically smaller than A). We store a rank / select structure for this bit vector using part (a) of Lemma 2.1 in $2n + o(n)$ bits. A cardinality query for a pattern is done by finding the difference between the positions of p^{th} and $(p+1)^{\text{st}}$ 0s in the bit vector, where p is the value obtained by treating the pattern as a number in binary (if $m < \lg n$, we shift the binary representations of p and $p+1$ to the left by $(\lg n - m)$ before the select operations). The number of 1s between these two positions is the number of occurrences of the given pattern. From the two positions, by performing rank operations, we can get the interval of SA in which all the entries point to suffixes that are prefixed with P, and use our index structures designed in Lemma 4.5 to list the occurrences.

We can further answer existential and cardinality queries in $O(m - \lg n)$ time for an arbitrary pattern P whose length is longer than $\lg n$. We first shift P to get its last $\lg n$ bits and treat them as a new pattern P'. We then use the method above to retrieve the range of SA whose entries correspond to suffixes prefixed with P'. Then we apply the backward search in algorithm `Count` for the remaining $m - \lg n$ bits of P, which takes $O(m - \lg n)$ time. $\square$

The above result is particularly useful when $m = \lg n + o(\lg n)$. Combined with Theorem 4.4 and Lemma 4.8, we have:

Theorem 4.5. *Given a binary text string T of length n, for any λ and μ such that $0 < \lambda, \mu < 1$, there is a data structure using $O(n)$ bits that can answer existential and cardinality queries on any pattern P of length m in $O(m - \lg n)$ time (when $m > \lg n$), or in constant time (when $m \leq \lg n$). This data structure can answer listing queries in additional $O(\mathtt{occ} \lg^{\lambda} n)$ time. When $m = \Omega(\lg^{1+\mu} n)$, this data structure can support pattern searching in $O(m + \mathtt{occ})$ time. It can also output a substring of T in $O(l / \lg n)$ time, where l is the length of the substring.*

4.7 Extensions to Larger Alphabets

In this section, we generalize our previous results on binary text in Sections 4.4, 4.5 and 4.6 to the case of larger alphabets. We first compare a suffix array with an arbitrary permutation of $[n]$, and describe a categorization theorem by which we can determine whether a given permutation is a suffix array. We then generalize our index structures to the case of larger alphabets.

4.7.1 The Categorization Theorem

We adopt the convention that the text T of length n is a string of $n - 1$ symbols drawn from alphabet $[\sigma]$, followed by a special end-of-file symbol #, where $\# = 0$. We follow the convention that $\sigma \leq n$. Based on our convention, there are σ^{n-1} different text strings of length n, so there are at most σ^{n-1} different suffix arrays associated with them. However, there are $n!$ different permutations of $[n]$. Therefore, for a large enough n, not all of the $n!$ permutations are suffix arrays of texts drawn from an alphabet of size σ.

We first give two definitions on permutations.

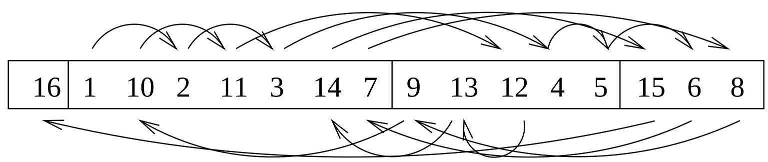


Figure 4.8: A permutation with 3 maximal ascending runs.

Definition 4.4. *Given a segment* $[i, j]$ *(*$1 < i \leq j \leq n$*) of a permutation* $M[1..n]$*, we call it an* **ascending run** *iff for any* k, l *where* $1 \leq k, l < n$*, if* $i \leq M^{-1}[k] < M^{-1}[l] \leq j$*, then* $M^{-1}[k+1] < M^{-1}[l+1]$*.*

Definition 4.5. *Given an ascending run* $[i, j]$ *of a permutation* $M[1..n]$*, it is a* **maximal ascending run** *iff for any segment* $[s, t]$ *of* M*, if* $[s, t] \supset [i, j]$*, then* $[s, t]$ *is not an ascending run.*

Figure 4.8 shows a permutation with 3 maximal ascending runs. We draw arrows as in Section 4.4.2. Each block (except the first block which corresponds to the suffix #) contains one maximal ascending run of the permutation. If the start positions of two arrows are in the same block, their end positions are in the same order as that of the start positions. The permutation has 3 maximal ascending runs, and we can find a 3-symbol string that corresponds to it, which is *aaabbcacbaabbac#*. However, there does not exist a binary string whose suffix array is this permutation.

Now we can describe our theorem.

Theorem 4.6. *A permutation* M *is a suffix array of a text string drawn from an alphabet of size* σ *iff it has at most* σ *maximal ascending runs.*

Proof. We first prove that a suffix array has at most σ maximal ascending runs.

For each alphabet symbol α, we assume that there are N_α characters in the text T that lexicographically precede it. We claim that any of the σ segments $[N_1+1, N_2], [N_2+1, N_3], ..., [N_\sigma+1, N_{\sigma+1}]$ (let $N_{\sigma+1} = n$) of SA is an ascending run if it is nonempty. To prove this, we need prove that any given nonempty segment $[N_\alpha+1, N_{\alpha+1}]$, where $1 \leq \alpha \leq n$, is an ascending run. We only need consider the nontrivial case when there are at least 2 entries in the segment. We observe that each suffix array entry in this segment corresponds to a text suffix whose first character is α according to the definition of suffix arrays. Therefore, for any k, l where $1 \le k, l < n$ and $N_{\alpha-1} + 1 \le SA^{-1}[k] < SA^{-1}[l] \le N_\alpha$, we have $T[k, n] = \alpha T[k+1, n] \prec T[l, n] = \alpha T[l+1, n]$, from which we can conclude that $T[k+1, n] \prec T[l+1, n]$. By the definition of suffix arrays, the inequality $SA^{-1}[k+1] < SA^{-1}[l+1]$ holds, which shows that $[N_{\alpha-1}+1, N_\alpha]$ is an ascending run.

From above we observe that $SA[2, n]$ can be divided into at most σ segments, each of which is an ascending run. According to the definition of ascending run, $SA[1]$ is not in any ascending run, and because each maximal ascending run is the union of one or more ascending runs described above (otherwise, if a maximal ascending run contains only part of one of the above ascending run, we can extend it by appending the rest of the ascending run to it), we can conclude that SA has at most σ maximal ascending runs.

Second, we prove that if a permutation M has σ maximal ascending runs, it is a suffix array of a text drawn from an alphabet of size σ. To prove this, we first present an algorithm to generate a text T drawn from an alphabet of size σ according to M, and then we prove that the suffix array SA of T is the same as M.

We construct T as follows. Because no two maximal ascending runs intersect (otherwise the union of the two ascending runs is another ascending run), and $M[1]$ is not in any ascending run, we can divide $M[2, n]$ into σ segments, each of which is a maximal ascending run. For each entry $M[i]$, if it is in the α^{th} segment, we set $T[M[i]] = \alpha$. We also set $T[n] = \#$.

We divide SA into σ segments $[N_1+1, N_2], [N_2+1, N_3], ..., [N_\sigma+1, N_{\sigma+1}]$ as above. We have the following lemma on reverse pairs (see Definition 4.3) on M and SA:

Lemma 4.10. *For any reverse pair (i, j) on M and SA, there exists α, where $1 \le \alpha \le n$, such that $N_\alpha + 1 \le M^{-1}[i] < M^{-1}[j] \le N_{\alpha+1}$ and $N_\alpha + 1 \le SA^{-1}[j] < SA^{-1}[i] \le N_{\alpha+1}$.*

We prove this lemma by contradiction. Assume that the lemma is not true. Then there are two cases. In the first case, $M^{-1}[i]$ and $M^{-1}[j]$ are not in the same segment. Then, by the construction algorithm of T, we have $T[i] < T[j]$. By the definition of suffix arrays, we have $SA^{-1}[i] < SA^{-1}[j]$, which is a contradiction. In the second case, $SA^{-1}[i]$ and $SA^{-1}[j]$ are not in the same segment. The proof in this case is the converse of the first case. $\square$

We now continue to prove Theorem 4.6 by proving that reverse pairs do not exist. Assume, contrary to what we are going to prove, there is one reverse pair (g, h) such that

g is the greatest among the first items of all the reverse pairs. We observe that both g and h are less than n because neither $M^{-1}[g]$ or $M^{-1}[h]$ is 1. Therefore, the inequality $1 < g+1, h+1 \leq n$ holds. Assume, without the loss of generality, that $M^{-1}[g] < M^{-1}[h]$. According to Lemma 4.10, there exists α, where $1 \leq \alpha \leq n$, such that $N_\alpha + 1 \leq M^{-1}[g] < M^{-1}[h] \leq N_{\alpha+1}$ and $N_\alpha + 1 \leq SA^{-1}[h] < SA^{-1}[g] \leq N_{\alpha+1}$. From the construction method of T, we observe that $[N_\alpha + 1, N_{\alpha+1}]$ is a maximal ascending run of M. Therefore, the inequality $M^{-1}[g+1] < M^{-1}[h+1]$ holds. By the definition of N_α, we have $T[g] = T[h] = \alpha$. Because $SA^{-1}[h] < SA^{-1}[g]$, the inequality $T[h, n] = \alpha T[h+1, n] \prec T[g, n] = \alpha T[g+1, n]$ holds. Therefore, $T[h+1, n] \prec T[g+1, n]$. By the definition of suffix arrays, we have the inequality $SA^{-1}[h+1] < SA^{-1}[g+1]$. Now we have another reverse pair $(g+1, h+1)$. Its first item $(g+1)$ is greater than g, which is a contradiction. □ (Thm 4.6)

Theorem 4.6 also applies to suffix arrays of binary texts, although in this case, its presentation is different from that of Theorem 4.1. This is because in Theorem 4.1, we assume that the end-of-file symbol, #, is lexicographically larger than the alphabet symbol 0 and smaller than 1 (this assumption, initially adopted by Grossi and Vitter [45, 46], has the property that each binary text of length n has a distinct suffix array), while in Theorem 4.6, we assume that # is smaller than all the alphabet symbols. Thus it is not possible to generalize the definition of ascending-to-max to the case of general alphabets, so we define maximal-ascending run and use it to present Theorem 4.6.

We discovered after the publication of [52] that our theorem is essentially equivalent to Theorem 6 in [2], which shows given a permutation M, how to infer a string with a minimal alphabet size whose suffix array is M. We choose to include our result as it was discovered independently.

4.7.2 Space Efficient Suffix Arrays

We generalize the three types of indexes for binary strings designed in Sections 4.5 and 4.6 to general alphabets. We present our results in three theorems.

Theorem 4.7. *Given a text string T of length n drawn from alphabet $[\sigma]$ ($\sigma = o(\frac{n}{\lg n})$), there is an index structure using $n \lg \sigma + o(n) \cdot \lg \sigma$ bits (without storing the raw text)* *that can answer existential and cardinality queries on any pattern string P of length m in $O(m \lg \sigma)$ time.*

Proof. To generalize our index structures to text strings drawn from larger alphabets, we conceptually think of having a bit vector B_α for each alphabet symbol α as in the proof of Theorem 4.3, i.e. $B_\alpha[SA^{-1}[i]] = 1$ iff $T[i-1] = \alpha$ for $i > 1$, and $B_\alpha[SA^{-1}[1]] = 0$. Storing the σ bit vectors explicitly using part (a) of Lemma 2.1 costs $n\sigma + o(n)$ bits, which is impractical for large alphabets. However, by exploring the dependency of these σ bit vectors, we can reduce the space cost. For any $i > 1$, there is one and only one α such that $B_\alpha[SA^{-1}[i]] = 1$, and $B_\alpha[SA^{-1}[1]] = 0$ for any α. Therefore, we can remove the space redundancy by combining these conceptual bit vectors to get a string in which the i^{th} character is α iff $B_\alpha[SA^{-1}[i]] = 1$, and the $SA^{-1}[1]^{\text{th}}$ character is $\#$. This string is in fact $T^{\texttt{BWT}}$, the Burrows-Wheeler transformed string of T (see Section 4.3.2). We use a wavelet tree [44] (see Section 3.2.1 for a description) to encode $T^{\texttt{BWT}}$. For any given k where $1 \leq k \leq n$ and $k \neq SA^{-1}[1]$, we can determine the character α such that $B_\alpha[k] = 1$ in $O(\lg \sigma)$ time with $T^{\texttt{BWT}}$. The rank and select operations on each conceptual bit vector can also be supported in $O(\lg \sigma)$ time by performing `string_rank` and `string_select` operations on $T^{\texttt{BWT}}$. In addition, we construct a conceptual array N of size σ such that for each alphabet symbol α, $N[\alpha]$ stores the number of characters in the text that lexicographically precede it.

With the above data structures, we can now modify the searching algorithm. Figure 4.9 presents the algorithm that answers existential and cardinality queries in the case of larger alphabets. The correctness proof is essentially the same as that for Algorithm `Count`. As each rank operation on a bit victor cost $O(\lg \sigma)$ time using A, the algorithm costs $O(m \lg \sigma)$ time. $T^{\texttt{BWT}}$ can be stored in $nH_0(T^{\texttt{BWT}}) + o(n) \cdot \lg \sigma = nH_0(T) + o(n) \cdot \lg \sigma \leq n \lg \sigma + o(n) \cdot \lg \sigma$ bits (see Section 3.2.1). Array N uses $\sigma \lg n = o(n)$ bits, when $\sigma = o(\frac{n}{\lg n})$. □

Theorem 4.8. *Given a text string T of length n drawn from alphabet $[\sigma]$ ($\sigma = o(\frac{n}{\lg n})$), there is an index structure using $n \lg \sigma + o(n) \cdot \lg \sigma$ bits (without storing the raw text) that supports, for any pattern P of length m,*

(i) pattern searching in $O((m + \texttt{occ} \lg n) \lg \sigma)$ time using an additional $\gamma n + o(n)$ bits, for any $0 < \gamma < 1$;

Algorithm $\texttt{Count}^*(T, P)$

$s \leftarrow 1$, $e \leftarrow n$, $i \leftarrow m$
while $i > 0$ and $s \leq e$ **do**
  $s \leftarrow N[P[i]] + \texttt{bin\_rank}_{B_{P[i]}}(1, s-1) + 1$, $e \leftarrow N[P[i]] + \texttt{bin\_rank}_{B_{P[i]}}(1, e)$
  $i \leftarrow i - 1$
return $\max(e - s + 1, 0)$

Figure 4.9: Answering existential and cardinality queries in the case of larger alphabets.

(ii) when $m = \Omega(\lg^{1+\mu} n)$, for any μ where $0 < \mu < 1$, pattern searching in $O(m \lg \sigma + \texttt{occ})$ time using an additional $n \lg \sigma + o(n)$ bits.

It also supports the output of a substring of length l in $O((l + \lg n) \lg \sigma)$ time using an additional ηn bits, for any $0 < \eta < 1$.

Proof. We construct the index structure of Theorem 4.7. To perform listing queries, we first show that given a position i in the original text T, if we know $SA^{-1}[i]$, we can compute $SA^{-1}[i-1]$ in constant time. We claim that if $B_\alpha[SA^{-1}[i]] = 1$, then $SA^{-1}[i-1] = N[\alpha] + \texttt{bin\_rank}_{B_\alpha}(1, SA^{-1}[i])$. The proof of the claim is essentially the same as the correctness proof of algorithm `Backward` and `Backward`* in Sections 4.6.1 and 4.6.4. Since finding the α such that $B_\alpha[SA^{-1}[i]] = 1$, and performing rank operation on B_α cost $O(\lg \sigma)$ time each over $T^{\texttt{BWT}}$, we can go backward in the text character by character in $O(\lg \sigma)$ time. Therefore, using the auxiliary data structures (S and F) of $\gamma n + o(n)$ bits and the searching algorithm in the proof of Lemma 4.5, we can list all the occurrences in $O(\texttt{occ} \lg \sigma \lg n)$ time.

The techniques in Section 4.6.3 can also be modified to speed up the listing queries of long patterns. The only modification is that we use Algorithm `Count`*(Figure 4.9) instead of Algorithm `Count` (Figure 4.5) in the case of larger alphabets. Thus we can support pattern searching in $O(m \lg \sigma + \texttt{occ})$ time, when $m = \Omega(\lg^{1+\mu} n)$.

Finally, the techniques in Section 4.6.2 can be used to make our data structures self-indexing. Given the query to retrieve the substring $T[i, i + l - 1]$, we first locate the first position j whose value is stored in V (see the proof of Lemma 4.6), where $j > i + l - 1$. From V, we can retrieve the index of the suffix array entry that corresponds to position

j in T in constant time. From $T^{\texttt{BWT}}$, we determine the character k such that $B_k[j] = 1$ in $O(\lg \sigma)$ time, which means the character before the j^{th} character in T is k. We can now output $T[j-1]$. We then use the method described above to walk backward in the text in $O(\lg \sigma)$ time and repeat the above process. Thus we can output the substring $T[i, i+l-1]$ in $O((l + \lg n) \lg \sigma)$ time using V, which occupies ηn bits. $\square$

Theorem 4.9. *Given a text string T of length n drawn from alphabet $[\sigma]$, for any constant λ and μ such that $0 < \lambda, \mu < 1$, there is a data structure using $O(n \lg \sigma)$ bits that can answer existential and cardinality queries on any pattern P of length m in $O((m - \lg_\sigma n) \lg \sigma)$ time (when $m > \lg_\sigma n$), or in constant time (when $m \leq \lg_\sigma n$). This data structure can answer listing queries in additional $O(\texttt{occ} \lg \sigma \lg^\lambda n)$ time. When $m = \Omega(\lg^{1+\mu} n)$, this data structure can support pattern searching in $O(m \lg \sigma + \texttt{occ})$ time using an additional $n \lg \sigma + o(n)$ bits. It can also output a substring of T in $O(l \lg \sigma / \lg n)$ time, where l is the length of the substring.*

Proof. The techniques in Section 4.6.4 can be used directly in the case of larger alphabets, except that now it takes $O(\lg \sigma \lg^\lambda n)$ time to locate each occurrence since the rank operation on any bit vector over A costs $O(\lg \sigma)$ time, and the data structures in each of the $\lceil \lambda^{-1} \rceil - 1$ levels use $2n \lg \sigma + o(n)$ bits. To apply the techniques in Section 4.6.5, we consider patterns of length $\lg_\sigma n$ instead of $\lg n$. $\square$

We observe a similarity with the alphabet-friendly FM-index discovered concurrently with our work by Ferragina *et al.* [34]. The bit vectors and basic algorithms are essentially the same. Our work differs from theirs in the auxiliary data structures and techniques used to speed up the queries on long patterns and short patterns. We also designed a multi-level trade-off between time and space.

Finally, we claim that the restriction $\sigma = o(\frac{n}{\lg n})$ in Theorem 4.7 and Theorem 4.8 can be removed by using a more recent implementation of wavelet trees by Mäkinen and Navarro *et al.* [64]. In their implementation, they concatenate the bit vectors at each level of the wavelet tree and use part (a) of Lemma 2.1 to store the concatenated bit vector at each level. The resulting tree occupies $n \lg \sigma + o(n) \cdot \lg \sigma$ bits. In this implementation, to count the number of characters in the text that are lexicographically smaller than a given character, we can use their algorithm to locate the conceptual bit vector that corresponds to the given character in the concatenated bit vector at the leaf level. The starting position of this bit vector minus 1 is the answer. This process takes $O(\lg \sigma)$ time. Thus in the proofs of Theorem 4.7 and Theorem 4.8, if we use this implementation, we can compute any element of N in $O(\lg n)$ time without explicitly storing it. This removes the above restriction, while still providing the same support for queries.

4.7.3 High-Order Entropy-Compressed Text Indexes for Large Alphabets

We now show how to apply the succinct indexes for strings to design high-order entropy-compressed text indexes. We first present the following lemma to encode strings in zeroth order entropy while supporting rank and select (we following the convention that the size of the alphabet is at most the size of the length of the string as in Chapter 3):

Lemma 4.11. *A string S of length n over alphabet $[\sigma]$ can be represented using $n(H_0(S) + O(\lg \sigma / \lg \lg \sigma)) = n(H_0(S) + o(\lg \sigma))$ bits to support* `string_access` *and* `string_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg \lg \sigma)$ time, and* `string_select` *for any character $\alpha \in [\sigma]$ in $O(1)$ time.*

Proof. As in the proof of Theorem 3.1, we consider the conceptual table E for string S. Each row of E is a bit vector, and we denote the α^{th} row by $E[\alpha]$ for $\alpha \in [\sigma]$. For each $\alpha \in [\sigma]$, we store $E[\alpha]$ using Lemma 2.2 in $\lg \binom{n}{n_\alpha} + o(n_\alpha) + O(\lg \lg n)$ bits, where n_α is the number of occurrences of α in S. By equation 4.1 (see Section 4.6.4), $E[\alpha]$ occupies at most $n_\alpha \lg \frac{en}{n_\alpha} + o(n_\alpha) + O(\lg \lg n)$ bits. Using this equation to sum the space cost of all the $E[\alpha]$'s for $\alpha \in [\sigma]$, the last two terms sum to $o(n) + O(\sigma \lg \lg n) = O(n \lg \lg \sigma)$ (as $\sigma \leq n$), while the first term on the right-hand side sums to $nH_0(S) + n \lg e$. Therefore, the total space cost is at most $n(H_0(S) + o(\lg \sigma))$ bits.

With the table E stored as above, `string_select` can be supported in $O(1)$ time, as $\texttt{string\_select}(\alpha, i) = \texttt{bin\_select}_{E[\alpha]}(1, i)$, for $\alpha \in [\sigma]$. With the constant-time support for `string_select` on S, we can construct a succinct index using Theorem 3.2 to support `string_rank` and `string_access` in $O(\lg \lg n)$ time. This index uses $n \cdot \lg \sigma / \lg \lg \sigma$ bits according to the proof of Theorem 3.2, so the overall space cost is $n(H_0(S) + O(\lg \sigma / \lg \lg \sigma))$ bits. □

Compared with Theorem 3.4, Lemma 4.11 compresses the string to zeroth order entropy instead of high-order entropy, but it supports navigational operations more efficiently. With this lemma, we can now prove our theorem.

Theorem 4.10. *A text string T of length n over alphabet $[\sigma]$ can be stored using $n(H_k(T) + o(\lg \sigma))$ for any $k \leq \beta \log_\sigma n - 1$ and $0 < \beta < 1$. Given a pattern P of length m, this encoding can answer existential and cardinality queries in $O(m \lg\lg \sigma)$ time, list each occurrence in $O(\lg^{1+\epsilon} n \lg\lg \sigma)$ time for any ϵ where $0 < \epsilon < 1$, and output a substring of length l in $O((l + \lg^{1+\epsilon} n) \lg\lg \sigma)$ time.*

Proof. As stated in Section 4.7.2, there is a similarity between the indexing techniques presented in that section and the alphabet-friendly FM-index discovered concurrently with our work and presented by Ferragina *et al.* [34]. Here we borrow some techniques they developed, and combine them with our results to prove this theorem.

In the proof of Theorem 4.7, we construct a bit vector B_α for each alphabet symbol α. As observed in the proof, these bit vectors can be combined to get $T^{\texttt{BWT}}$. We use Lemma 4.2 to partition $T^{\texttt{BWT}}$ into a set of strings $S_1, S_2, \cdots, S_z$. We use Lemma 4.11 to encode each string. We construct a bit vector B of length n, in which $B[i] = 1$ iff there exists a string S_j, whose starting position is position i of $T^{\texttt{BWT}}$. We encode B using part (b) of Lemma 2.1. We construct a two-dimensional array $M[1..z][1..\sigma]$, where $M[i][\alpha]$ stores the total number of occurrences of character α in strings $S_1, S_2, \cdots, S_{i-1}$. We also construct N as in the proof of Theorem 4.7.

With the above data structures, we can compute $\texttt{bin\_rank}_{B_\alpha}(1, i)$. We observe that this operation returns the number of occurrences of α in $T^{\texttt{BWT}}[1..i]$. To support it, we first locate the string S_j that position i is in. As $j = \texttt{bin\_rank}_B(1, j)$, we can locate S_j in constant time. We also have position i of $T^{\texttt{BWT}}$ is the l^{th} position of S_j, where $l = i - \texttt{bin\_select}_B(1, j) + 1$. By the definition of $M[\alpha][j]$, we have $\texttt{bin\_rank}_{B_\alpha}(1, i) = M[\alpha][j] + \texttt{string\_rank}_{S_j}(\alpha, l)$. As $\texttt{string\_rank}_{S_j}$ is supported in $O(\lg\lg \sigma)$ time, we can compute $\texttt{bin\_rank}_{B_\alpha}(1, i)$ in $O(\lg\lg \sigma)$ time. We can also compute $B_\alpha[i]$ in $O(\lg\lg \sigma)$ time, as $B_\alpha[i]$ is 1 iff $\texttt{string\_access}_{B_j}(l) = \alpha$.

With the above operations supported, we now show how to answer queries. To answer existential and cardinality queries, we directly apply the searching algorithm in the proof of Theorem 4.7. The runtime is now $O(m \lg\lg \sigma)$, as we can support the computation of

$\texttt{bin\_rank}_{B_\alpha}(1, i)$ in $O(\lg\lg\sigma)$ time. To answer listing queries, we construct auxiliary data structures S and F as in the proof of Theorem 4.8. We slightly modify the way they are constructed by marking every $\lg^{1+\epsilon} n^{\text{th}}$ position of the original text instead of every $\lceil \lg n/\gamma \rceil^{\text{th}}$ position. This is to reduce the space cost of S and F to $o(n)$ bits. With such a modification and the support of the operations stated in the previous paragraph, we list each occurrence of P in $O(\lg^{1+\epsilon} n \lg\lg\sigma)$ time. Finally, the support for self-indexing can be achieved by using the technique in the proof of Theorem 4.8. We also reduce the space cost of V to $o(n)$ by storing, for every $(\lg^{1+\epsilon} n)^{\text{th}}$ position in T, the index of its corresponding entry in SA. The time required to output a substring of length l is thus $O((l + \lg^{1+\epsilon} n) \lg\lg\sigma)$.

To analyze the space cost of all the data structures, the encoding of B_α's occupy $\sum_{i=1}^{z}(|S_i|H_0(S_i) + O(\lg|S_i|\lg\sigma/\lg\lg\sigma))$ bits in total. B occupies $\lg\binom{n}{z} + o(n)$ bits. M occupies $z\sigma\lg n$ bits. N occupies $\sigma\lg n$ bits. All the other data structures occupy $o(n)$ bits. Therefore, the total space cost in bits is:

$$\begin{aligned}
&\sum_{i=1}^{z}(|S_i|H_0(S_i) + O(\lg|S_i|\lg\sigma/\lg\lg\sigma) + z\sigma\lg n + \lg\binom{n}{z} + \sigma\lg n + o(n) \\
&< \sum_{i=1}^{z}(|S_i|H_0(S_i) + O(\lg|S_i|\lg\sigma/\lg\lg\sigma)) + z(\sigma+1)\lg n + \sigma\lg n + o(n) \\
&= \sum_{i=1}^{z}(|S_i|H_0(S_i) + O(\lg|S_i|\lg\sigma/\lg\lg\sigma) + (\sigma+1)\lg n) + \sigma\lg n + o(n).
\end{aligned}$$

By the definition of order notations, there exists a constant c, such that the above value is bounded by

$$\sum_{i=1}^{z}(|S_i|H_0(S_i) + c\lg|S_i|\lg\sigma/\lg\lg\sigma + (\sigma+1)\lg n) + \sigma\lg n + o(n).$$

We then apply Lemma 4.2 to bound the above value by:

$$nH_k(T) + cn\lg\sigma/\lg\lg\sigma + O(\sigma^{k+1}\lg n) + o(n) = nH_k(T) + n\cdot o(\lg\sigma) + O(\sigma^{k+1}\lg n) + o(n). \tag{4.3}$$

When $k \le \beta\log_\sigma n - 1$ for $0 < \beta < 1$, we have $\sigma^{k+1} \le n^\beta$. In this case, the third item in equation 4.3 is bounded by $o(n)$, so equation 4.3 is bounded by $nH_k(T) + n\cdot o(\lg\sigma) + o(n) = n(H_k(T) + o(\lg\sigma))$. □

Grossi *et al.* [44] designed a text index that uses $nH_k + o(n) \cdot \lg \sigma$ bits, and supports existential and cardinality queries in $O(m \lg \sigma + \text{polylog}(n))$ time (See Section 4.2). Golynski *et al.* [41] reduced the $\lg \sigma$ factor in the query time to a $\lg \lg \sigma$, but their index is not easily compressible. Our text index has the advantages of both these indexes.

4.8 Discussion

In this chapter, we gave a theorem that characterizes a permutation as the suffix array of a binary string. Based on the theorem, we designed a succinct representation of suffix arrays of binary strings that uses $n + o(n)$ bits (the theoretical minimum plus a lower order term), and answers existential and cardinality queries in $O(m)$ time without storing the raw text. With additional data structures in $\gamma n + o(n)$ bits, for any $0 < \gamma < 1$, we can answer listing queries in $O(m + \mathsf{occ} \lg n)$ time in the general case. For long patterns (i.e. when $m = \Omega(\lg^{1+\mu} n)$), for $0 < \mu < 1$, we answer listing queries in $O(m + \mathsf{occ})$ time using an additional $n + o(n)$ bits. Using only $\eta n + o(n)$ additional bits, for any $0 < \eta < 1$, we can make our index a self-indexing structure, which can output a substring of length l in $O(l + \lg n)$ time without storing the raw text, and this technique saves a lot of space especially for text strings drawn from larger alphabets. Another implementation of our index uses $O(n)$ bits, answers listing queries in $O(m + \mathsf{occ} \lg^{\lambda} n)$ time, for $0 < \lambda < 1$, and outputs a substring of length l in $O(l / \lg n)$ time. This implementation also provides the same support for long patterns. An independent approach that answers existential and cardinality queries for patterns of length at most $\lg n$ in $O(1)$ time using $2n + o(n)$ bits of space is also presented. In addition to designing text indexes, an efficient algorithm that checks whether a given permutation is a suffix array of a binary string is also developed.

Each of the three different implementations of our index structures has its own merits. The first one (Theorem 4.3), although only supporting existential and cardinality queries, has space cost of only $n + o(n)$, which is optimal. The constant factor of the second one (Theorem 4.4) is also small. The third approach (Theorem 4.5) supports more efficient searching using $O(n)$ space. When combined with the compressed suffix tree designed by Grossi and Vitter [45], it supports listing queries in $O(\frac{m}{\lg n} + \mathsf{occ} \lg^{\mu} n)$, which is the same as their result [45].

We also generalized our results to the case of larger alphabets. When we apply our succinct indexes for strings to succinct text indexes, we design a text index using $n(H_k(T)+o(\lg\sigma))$ bits that supports pattern searching in $O(m\lg\lg\sigma+\mathsf{occ}(\lg^{1+\epsilon} n\lg\lg\sigma))$ time. This is the first high-order entropy text index that supports cardinality queries in $O(m\lg\lg\sigma)$ time.

An open problem in this field is to design a text index using $O(n\lg\sigma)$ bits to support pattern searching in $O(m+\mathsf{occ})$ time.

Chapter 5

Trees

This chapter deals with the problem of designing succinct representations of (unlabeled) ordinal trees and multi-labeled trees. The chapter starts with an introduction in Section 5.1, followed by a brief review of previous work in Section 5.2, and a summary of the existing results we use in Section 5.3. In Section 5.4, we design a succinct representation of ordinal trees that supports all the navigational operations supported by various succinct tree representations. In Section 5.5, we show that our method supports two other encoding schemes of ordinal trees as abstract data types. We design succinct indexes for multi-labeled trees in Section 5.6. Section 5.7 gives some conclusion remarks and suggestions for future work.

5.1 Introduction

Trees are fundamental data structures in computer science. Two forms are of particular importance. An *ordinal tree* is a rooted tree in which the children of a node are ordered and specified by their rank, while in a *cardinal tree* of degree k, each child of a node is identified by a unique number from the set $[k]$. In this chapter, we mainly consider ordinal trees.

The straightforward representation of trees explicitly associates with each node the pointers to its children. Thus, an ordinal tree of n nodes is represented by $\Theta(n)$ words or $\Theta(n \lg n)$ bits. This representation allows straightforward, efficient parent-to-child navigation in trees. However, as current applications often consider very large trees, such a representation often occupies too much space.

To solve this problem, various methods have been proposed to encode an ordinal tree of n nodes in $2n + o(n)$ bits, which is close to the information-theoretic minimum of $2n - O(\lg n)$ bits (as there are $\binom{2n}{n}/(n+1)$ different ordinal trees), while supporting various navigational operations efficiently. These representations are based on various traversal orders of the nodes in the tree: *preorder* (in which each node is visited before its descendants) and *postorder* (in which each node is visited after its descendants) are well-known. The `DFUDS` *(depth first unary degree sequence) order* in which all the children of a node are visited before its other descendants [10, 9] is another useful ordering. However, different representations of trees usually support different sets of navigational operations. It is desirable to design a succinct representation that supports all the navigational operations of various succinct tree structures. We consider the following operations:

- `child`(x, i): the i^{th} child of node x for $i \geq 1$;
- `child_rank`(x): the number of left siblings of node x plus 1;
- `depth`(x): the depth of x, i.e. the number of edges in the rooted path to node x;
- `level_anc`(x, i): the i^{th} ancestor of node x for $i \geq 0$ (given a node x at depth d, its i^{th} ancestor is the ancestor of x at depth $d - i$);
- `nbdesc`(x): the number of descendants of node x;
- `degree`(x): the degree of node x, i.e. the number of its children;
- `height`(x): the height of the subtree rooted at node x;
- `LCA`(x, y): the lowest common ancestor of nodes x and y;
- `distance`(x, y): the number of edges of the shortest path between nodes x and y;
- `leftmost_leaf`(x) (`rightmost_leaf`(x)): the leftmost (or rightmost) leaf of the subtree rooted at node x;
- `leaf_rank`(x): the number of leaves before node x in preorder plus 1;

- `leaf_select`(i): the i^{th} leaf among all the leaves from left to right;
- `leaf_size`(x): the number of leaves of the subtree rooted at node x;
- $\texttt{node\_rank}_{\texttt{PRE/POST/DFUDS}}(x)$: the position of node x in the preorder, postorder or `DFUDS` order traversal of the tree;
- $\texttt{node\_select}_{\texttt{PRE/POST/DFUDS}}(r)$: the r^{th} node in the preorder, postorder or `DFUDS` order traversal of the tree;
- `level_leftmost`(i) (`level_rightmost`(i)): the first (or last) node visited in a preorder traversal among all the nodes whose depths are i;
- `level_succ`(x) (`level_pred`(x)): the *level successor* (or *predecessor)* of node x, i.e. the node visited immediately after (or before) node x in a preorder traversal among all the nodes whose depths are equal to `depth`(x).

Motivated by the research on XML databases, the problem of representing trees with labels has attracted much attention. A *labeled tree* is a tree in which each node is associated with a label from a given alphabet $[\sigma]$, while in a *multi-labeled tree*, each node can be associated with more than one label. We use n to denote the number of nodes in a labeled/multi-labeled tree, and t to denote the total number of node-label pairs in a multi-labeled tree. A node y is a *α-child/descendant/ancestor* of a node x if it is a child/descendant/ancestor of x associated with label α. The operations on labeled/multi-labeled trees not only include the pure navigational operations discussed above, but also include powerful label-based queries. For example, one may need to retrieve all the α-children of a given node.

The first result of this chapter is to extend the succinct ordinal trees based on tree covering (`TC`) by Geary *et al.* [36, 37] to support all the operations on trees proposed in other work. We compare our results with existing results in Table 5.1, in which the columns `BP` and `DFUDS` list the results of tree representations based on balanced parentheses and `DFUDS` (see Section 5.2 for an introduction of these two approaches), respectively, and the columns old `TC` and new `TC` list the results by Geary *et al.* [36, 37] and our results, respectively.

operations	BP [70, 71, 72, 17, 74, 62]	DFUDS [10, 9, 56, 6]	*old* TC [36, 37]	*new* TC
`child`, `child_rank`	✓	✓	✓	✓
`depth`, `level_anc`	✓	✓	✓	✓
`nbdesc`, `degree`	✓	✓	✓	✓
$\texttt{node\_rank}_{\texttt{PRE}}$, $\texttt{node\_select}_{\texttt{PRE}}$	✓	✓	✓	✓
$\texttt{node\_rank}_{\texttt{POST}}$, $\texttt{node\_select}_{\texttt{POST}}$	✓		✓	✓
`height`	✓			✓
`LCA`, `distance`	✓	✓		✓
`leftmost_leaf`, `rightmost_leaf`	✓	✓		✓
`leaf_rank`, `leaf_select`	✓	✓		✓
`leaf_size`	✓	✓		✓
$\texttt{node\_rank}_{\texttt{DFUDS}}$, $\texttt{node\_select}_{\texttt{DFUDS}}$		✓		✓
`level_leftmost`, `level_rightmost`				✓
`level_succ`, `level_pred`	✓			✓

Table 5.1: Navigational operations supported in $O(1)$ time on succinct ordinal trees using $2n + o(n)$ bits.

Our second result deals with BP and DFUDS representations as abstract data types, showing that any operation to be supported by BP or DFUDS in the future can also be supported by TC efficiently. Our third result is a succinct index for multi-labeled trees that supports efficient retrieval of α-children/descendants/ancestors of a given node.

5.2 Previous Work

5.2.1 Ordinal Trees

Jacobson's succinct tree representation [55] was based on the *level order unary degree sequence* (LOUDS) of a tree, which lists the nodes in a level-order traversal[1] of the tree and encodes their degrees in unary. With this, Jacobson [55] encoded an ordinal tree in $2n + o(n)$ bits to support the selection of the first child, the next sibling, and the parent of a given node in $O(\lg n)$ time under the bit probe model. Clark and Munro [21] further showed how to support the above operations in $O(1)$ time under the word RAM model with $\Theta(\lg n)$ word size.

As the original work on the LOUDS ordering supports only a very limited set of operations, various researchers have proposed different ways to represent ordinal trees using $2n + o(n)$ bits. The following are the three main approaches.

Based on the isomorphism between *balanced parenthesis sequences* (BP) and ordinal trees, Munro and Raman [70, 71] proposed another type of succinct representation of trees. The BP sequence of a given tree can be obtained by performing a depth-first traversal, and outputting an opening parenthesis each time a node is visited, and a closing parenthesis immediately after all its descendants are visited. They presented a succinct representation of an ordinal tree of n nodes in $2n + o(n)$ bits based on BP, which supports parent[2], nbdesc, depth, $\texttt{node\_rank}_{\texttt{PRE/POST}}$ and $\texttt{node\_select}_{\texttt{PRE/POST}}$ in constant time, and $\texttt{child}(x, i)$ in $O(i)$ time. Munro *et al.* [72] provided constant-time support for leaf_rank, leaf_select, leftmost_leaf, rightmost_leaf and leaf_size on the BP representation using $o(n)$ additional bits, which has applications to design space-efficient suffix trees. The constant-time support for degree was provided by Chiang *et al.* [17], and the support for level_anc, level_succ and level_pred in constant time was further provided by Munro and Rao [74]. Recently, Lu and Yeh [62] showed how to support child, child_rank, height, LCA and distance in constant time.

[1]The ordering puts the root first, then all of its children, from left to right, followed by all the nodes at each successive level (depth).

[2]We use $\texttt{parent}(x)$ to denote the parent of node x, which is a restricted version of level_anc, as $\texttt{parent}(x) = \texttt{level\_anc}(x, 1)$.

Another type of succinct tree representation is based on the *depth first unary degree sequence* (`DFUDS`). The `DFUDS` sequence represents a node of degree d by d opening parentheses followed by a closing parenthesis. All the nodes are listed in preorder (an extra opening parenthesis is added to the beginning of the sequence), and each node is numbered by its opening parenthesis in its parent's description (`DFUDS` number). Benoit *et al.* [10, 9] presented a succinct tree representation based on `DFUDS` that occupies $2n + o(n)$ bits and supports `child`, `parent`, `degree` and `nbdesc` in constant time. In their representation, each node is referred to by the position of the first of the $d + 1$ parentheses representing it. Jansson *et al.* [56] extended this representation using $o(n)$ additional bits to provide constant-time support for `child_rank`, `depth`, `level_anc`, `LCA`, `distance`[3], `leftmost_leaf`, `rightmost_leaf`, `leaf_rank`, `leaf_select`, `leaf_size`, $\texttt{node\_rank}_{\texttt{PRE}}$ and $\texttt{node\_select}_{\texttt{PRE}}$. Barbay *et al.* [6] further showed how to support $\texttt{node\_rank}_{\texttt{DFUDS}}$ and $\texttt{node\_select}_{\texttt{DFUDS}}$.

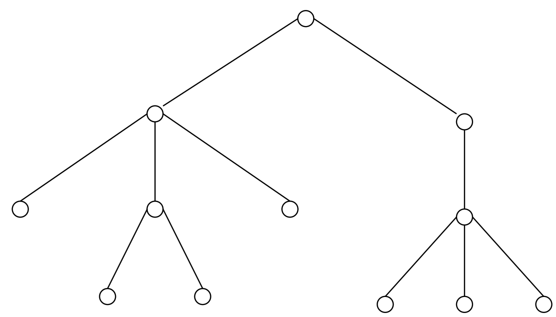
Finally, a more recent approach to represent static ordinal trees is based on a *tree covering* algorithm (`TC`). Geary *et al.* [36, 37] proposed an algorithm to cover an ordinal tree with a set of mini-trees, each of which is further covered by a set of micro-trees. Their representation occupies $2n+o(n)$ bits, and supports `child`, `child_rank`, `depth`, `level_anc`, `nbdesc`, `degree`, $\texttt{node\_rank}_{\texttt{PRE/POST}}$ and $\texttt{node\_select}_{\texttt{PRE/POST}}$ in constant time.

See Table 5.1 in Section 5.2 for a complete list of operations that each of the three representations supports. For an example of the `LOUDS`, `BP` and `DFUDS` sequences of a given ordinal tree, see Figure 5.1.

5.2.2 Labeled and Multi-Labeled Trees

Geary *et al.* [36, 37] defined labeled extensions of the first six operators defined in Section 5.1. Their data structures support those operators in constant time using simple auxiliary data structures to store label information in addition to their succinct ordinal tree representation [36, 37]. However, the overall space required is $2n + n\big(\lg\sigma + O(\sigma\lg\lg\lg n/\lg\lg n)\big)$ bits, which is much more than the lower bound of $n\log_2\sigma + 2n - O(\lg n)$ bits suggested by information theory.

[3]Jansson *et al.* [56] did not explicitly show how to support `distance`, but the support for it directly follows the support for `depth` and `level_anc`.



LOUDS: 110111101001100111000000
BP: ((()(()())())((()()())))
DFUDS: ((()((())(())))()((())))

Figure 5.1: An example of the `LOUDS`, `BP` and `DFUDS` sequences of a given ordinal tree.

Ferragina *et al.* [29] proposed another structure based on the *xbw* transform of a labeled tree, which conceptually builds a compressed suffix array for all the labeled rooted paths in the tree. It supports locating the first child of a given node x labeled α in constant time, and finding all the children of x labeled α in constant time per child. But it does not efficiently support the retrieval of the ancestors or descendants by labels. Also it uses $2n \lg \sigma + O(n)$ bits, which is almost twice the minimum space required to encode the tree. Ferragina *et al.* [29] also showed how to use a wavelet tree to reduce the size to $n \lg \sigma + O(n)$ bits, but each of the above operations then takes $O(\lg \sigma)$ time. This structure can be further compressed to $nH_k + O(n)$ bits, where H_k is the k^{th} order entropy of labeled trees they defined [29], based on the context of upward paths of the nodes. Another interesting operation supported by the above representation is the subpath query, which returns the number of nodes whose upward paths are prefixed with a given pattern. Given a pattern of length p, the above representation can answer the subpath query in $O(p \lg \sigma)$ time.

Based on the succinct integrated encoding for binary relations, Barbay *et al.* [5] gave an encoding for labeled trees using $n\,(\lg \sigma + o(\lg \sigma))$ bits to support the retrieval of the ancestors or descendants by labels in $O(\lg \lg \sigma)$ time per node. It also supports the computation of the number of descendants (of a given node) associated with a given label in $O(\lg \lg \sigma)$ time. The same technique is generalized to represent multi-labeled trees in $t\,(\lg \sigma + o(\lg \sigma))$

Algorithm `cover`(x, M) [36, 37]

1. If x is a leaf, make $\{x\}$ a PCM and return.
2. Otherwise, x has one or more children, $x_1, x_2, \cdots, x_d$. Call `cover`(x_1), `cover`(x_2), $\cdots$, `cover`(x_d).
3. If $x_1, x_2, \cdots, x_d$ are all roots of newly created mini-trees, make $\{x\}$ a PCM and return.
4. Otherwise, one or more children of x are roots of PCM's. $Y = \{S_1, S_2, \cdots, S_p\}$ denotes the set of PCM's whose roots are children of x.
5. If $|\bigcup Y| < M - 1$, make $\{x\} \cup (\bigcup Y)$ a PCM and return.
6. Otherwise, repeat the following steps:
   (a) Create a mini-tree $Z = \bigcup\{\{x\}, S_q, S_{q+1}, \cdots, S_r\}$, where S_q is the leftmost PCM in Y and r is the index such that $|Z| \geq M$ but $|Z - S_r| < M$.
   (b) $Y \leftarrow Y - \{S_q, S_{q+1}, \cdots, S_r\}$.
   (c) If $\bigcup Y < M - 1$, output $Z \cup (\bigcup Y)$ as a mini-tree and return.
   (d) Otherwise, output Z as a mini-tree and go to step (a).

Figure 5.2: An algorithm to cover an ordinal tree [36, 37].

bits and support the same operations[4].

5.3 Preliminaries

5.3.1 Succinct Ordinal Tree Representation Based on Tree Covering

In this section, we briefly summarize the succinct ordinal trees based on tree covering proposed by Geary *et al.* [36, 37], which we extend in Section 5.4 to support new operations. In particular, we introduce the notation and list the data structures we reuse here.

[4]In this proposal, we assume that each node of the tree is associated with at least one label (thus $t \geq n$), and that $n \geq \sigma$. The results can be extended to other cases by simple reductions shown in Section 3.5 for binary relations.

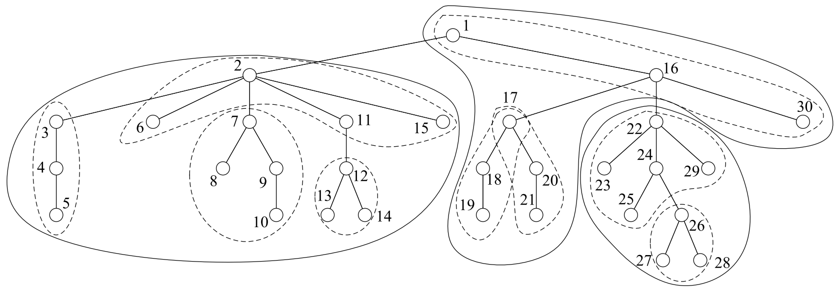


Figure 5.3: An example of covering an ordinal tree with parameters $M = 8$ and $M' = 3$, in which the solid curves enclose mini-trees and dashed curves enclose micro-trees.

The Tree Covering Algorithm Geary *et al.* [36, 37] proposed an algorithm to cover an ordinal tree by *mini-trees* of size $\Theta(M)$ for a given parameter M. Any two mini-trees computed by this algorithm either do not *intersect* (i.e. have one or more common nodes), or only intersect at their common root. Figure 5.2 presents the algorithm `cover`, whose parameters include a node x and M. It either creates a set of new mini-trees rooted at x, or designates a set of nodes as a *partially completed mini-tree*, or PCM (we use $\bigcup S$ to denote $\cup_{T \in S} T$). We can pass the root node as the parameter to cover an ordinal tree T by mini-trees of size $\Theta(M)$. Note that at the end of the algorithm, there might be a PCM at the root of T, and we make it a mini-tree. This is the only mini-tree whose size may be smaller than $\Theta(M)$.

Geary *et al.* [36, 37] showed that the size of any mini-tree is at most $3M - 4$, and the size of any mini-tree that does not contain the root of the entire tree is at least M. In their paper, they choose $M = \lceil \lg^4 n \rceil$ to cover a given tree T by mini-trees. They further use the same algorithm with the parameter $M' = \lceil \lg n/24 \rceil$ to cover each mini-tree by a set of *micro-trees*. We use the same parameters and procedure to cover ordinal trees in this chapter. Figure 5.3 gives an example of covering an ordinal tree using this algorithm.

Identifying Mini-trees, Micro-trees, and Nodes Geary *et al.* [36, 37] list the mini-trees $t_1, t_2, \cdots$ in an order such that in a preorder traversal of T, either the root of t_i is visited before that of t_{i+1}, or if these two mini-trees share the same root, the children of the root of t_i are visited before the children of the root of t_{i+1}. The i^{th} mini-tree in the above sequence is denoted by μ^i. All the micro-trees in a mini-tree are also listed in the same order, and the j^{th} micro-tree in mini-tree μ^i is denoted by μ^i_j. When the context is clear, we also refer to this micro-tree using μ_j. A node is denoted by its preorder number from an external viewpoint. To further relate a node to its position in its mini-tree and micro-tree, they define the *τ-name* of a node x to be a triplet $\tau(x) =< \tau_1(x), \tau_2(x), \tau_3(x) >$, which means that node x is the $\tau_3(x)^{\text{th}}$ node visited in a preorder traversal of micro tree $\mu^{\tau_1(x)}_{\tau_2(x)}$. For a node that exists in more than one mini-tree and/or micro-tree, its lexicographically smallest τ-name is its *canonical name*, and the copy of the node with the canonical name is called a *canonical copy*. For example, in Figure 5.3, node 11 is in mini-tree μ^2 and micro-tree μ^2_1, and its τ-name is $< 2, 1, 3 >$. The canonical name of node 17 is $< 1, 2, 1 >$.

Geary *et al.* [36, 37] also defined the notion of *preorder boundary nodes*. A node is a tier-1 (or tier-2) preorder boundary node iff during a preorder traversal of the tree, it is either the first or the last node of a mini-tree (or micro-tree). For example, in Figure 5.3, nodes 1, 2, 15, 22, 28 and 30 are tier-1 preorder boundary nodes. Nodes 2, 3, 5, 7, 10, 12, 14, 15 and others are tier-2 preorder boundary nodes.

Extended Micro-trees To enable the efficient retrieval of the children of a given node, Geary *et al.* [36, 37] proposed the concept of *extended micro-trees*. To compute the set of extended micro-trees, each micro-tree is initially made into an extended micro-tree, and each of its nodes is called an *original node* of the extended micro-tree. Every node x that is the root of a micro-tree is promoted into the extended micro-tree to which its parent, y, belongs. If y belongs to more than one micro-tree, then we first retrieve the rightmost sibling to the left of x that is not the root of a micro-tree. If such a node exists, x is promoted into the extended micro-tree to which it belongs. Otherwise, x is promoted into the micro-tree that has the canonical copy of y. A node promoted into an extended micro-tree is called a *promoted node*. For example, the extended version of the micro-tree μ^1_1 has original nodes 1, 16 and 30, and promoted nodes 17 and 22. Geary *et al.* [36, 37] proved the following lemma.

Lemma 5.1 ([36, 37]). *Consider a node x that is not the root of any micro-tree. Then (at least a promoted copy of) each of x's children is in the extended micro-tree that contains x as an original node.*

To bound the size of micro-trees, Geary *et al.* [36] defined a *type 1 extended micro-tree* to be a micro-tree whose size is at most $\frac{1}{4}\lg n$, and an extended micro-tree whose size is larger than $\frac{1}{4}\lg n$ is called a *type 2 extended micro-tree*. They further proved that the number of type 2 extended micro trees is $O(n/(\lg n)^2)$, and they have $O(n/\lg n)$ original nodes in total.

The Main Data Structures

Here we list the main data structures designed by Geary *et al.* [36, 37] upon which we construct additional auxiliary data structures. The main data structures are designed to represent each individual extended micro-tree. They take $2n + o(n)$ bits in total for an ordinal tree of n nodes and can be used as building blocks to construct the representation of the tree at a higher level. The following data structures are constructed for a given extended micro tree μ_i of o_i original nodes and p_i promoted nodes, if it is a type 1 extended micro-tree:

- A header information to indicate the type of the extended micro-tree, and to encode p_i and o_i.
- A bit array $\mathtt{tree}_i$ of $2(p_i + o_i)$-bit to encode μ_i. Recall that there are at most 2^{2n} different ordinal trees of n nodes, so there exists a canonical way to encode an ordinal tree in $2n$ bits. Geary *et al.* [36, 37] chose to encode $\mathtt{tree}_i$ using the balanced parenthesis sequence.
- A bit vector $\mathtt{nodetypes}_i$ of $\lg \binom{p_i+o_i}{o_i}$ bits (constructed using Part (b) of Lemma 2.1) to tell whether the j^{th} node of μ_i in preorder is an original node.
- An encoding of all edges that leave μ_i ($\mathtt{edges}_i$):
  - A bit vector of p_i bits to indicate whether a promoted node is in the same mini-tree as μ_i, listed by the preorder of the promoted nodes. The number of 1s in this bit vector is denoted by p'_i.
  - To store the τ-names of the original copies of the promoted nodes, only $p_i - p'_i$ τ_1-names and p_i τ_2-names are stored.

If μ_i is a type 2 extended micro-tree, we store the same information except that $\texttt{tree}_i$ now consists of an $O(p_i + o_i)$-bit DFUDS representation by Benoit *et al.* [10, 9] to represent μ_i, and a $2o_i$-bit encoding to store the tree structure of the corresponding micro-tree excluding the promoted nodes.

Geary *et al.* [36, 37] also defined the *implicit* representations of micro-trees and showed that given an extended micro-tree, we can compute the implicit representation of the micro-tree corresponding to it (excluding its promoted nodes) in constant time. To encode a micro-tree of m nodes, they first encode its size using $\lg(3M') = O(\lg \lg n)$ bits. Then, as there are at most 2^{2m} different micro-trees of m nodes, there exists a canonical way to encode it in $2m$ bits (Geary *et al.* [36, 37] chose to use the balanced parenthesis sequence). The implicit representation of a micro-tree consists of the above two parts. The data structures constructed for each type 2 extended micro-tree already explicitly stores the implicit encoding of its corresponding micro-tree excluding the promoted nodes. For a type 1 micro-tree μ_i, the data structures $\texttt{tree}_i$ and $\texttt{nodetypes}_i$ uniquely determines the structure of its corresponding micro-tree excluding the promoted nodes. Geary *et al.* [36, 37] claimed that the concatenation of $\texttt{tree}_i$ and $\texttt{nodetypes}_i$ has at most $\frac{3}{4} \lg n$ bits for a type 1 extended micro-tree. Based on the fact, they designed an $o(n)$-bit data structure to retrieve the implicit representation of the micro-tree corresponding to a type 1 extended micro tree in constant time.

5.3.2 Range Maximum/Minimum Query

Given an array D of n integers and an arbitrary range $[i, j]$, where $1 \leq i \leq j \leq n$, the *range maximum query* (or *range minimum query*) retrieves the leftmost maximum (or minimum) value among the elements in $D[i..j]$. Bender and Farach-Colton presented a simple algorithm to support range maximum/minimum queries on D in constant time using $O(n \lg n)$ bits [8]. Based on this algorithm, Sadakane [79] showed how to support the range minimum/maximum query in $O(1)$ time on an array of n integers encoded in the form of balanced parentheses using $o(n)$ additional bits. This approach does not work when the integers are stored explicitly in an array, but a useful observation of Sadakane's algorithm is that, when the starting and ending positions of the range are multiples of $\lg n$, it only uses the auxiliary data structures constructed to support range maximum/minimum queries, without accessing D.

Lemma 5.2 ([79])**.** *Given an array $D[1..n]$ of integers, there is an auxiliary data structure of $o(n)$ bits that, when the starting and ending positions of the range are multiples of $\lg n$ (i.e. the given range is of the form $[k \lg n..l \lg n]$), this auxiliary data structure can, without accessing the array D, support ranged maximum/minimum queries in $O(1)$ time.*

5.3.3 Lowest Common Ancestor

The problem of supporting operator `LCA` on ordinal trees was initially studied for the explicit, pointer-based representation of trees. Bender *et al.* [8] showed how to support `LCA` using an additional $O(n \lg n)$ bits[5] for any tree representation through a reduction to a particular case of range minimum query. If we store the tree using the tree representation by Geary *et al.* [36, 37], we have:

Lemma 5.3. *An ordinal tree can be represented in $O(n \lg n)$ bits to support* `LCA` *and the operations listed in the column old* `TC` *of Table 5.1 in $O(1)$ time.*

5.3.4 Visibility Representation of Graphs

In a *visibility representation* of graphs, vertices are mapped to horizontal segments and edges to vertical segments that intersect only adjacent vertex segments [82]. Various visibility representations have been proposed in the literature. In this chapter, we adopt the notion of *weak visibility representation* [82].

Definition 5.1 ([82])**.** *A* **weak visibility representation** *of a graph G is a mapping of its vertices into non-overlapping horizontal segments called* **vertex segments** *and of its edges into vertical segments called* **edge segments**. *Under this mapping, the edge between any two given vertices x and y is mapped to an edge segment whose end points are on the vertex segments of x and y, and this edge segment does not cross any other vertex segment.*

[5] It is not shown in [8] how much space their data structures occupy. However, it is easy to verity that the space cost is $O(n \lg n)$ bits.

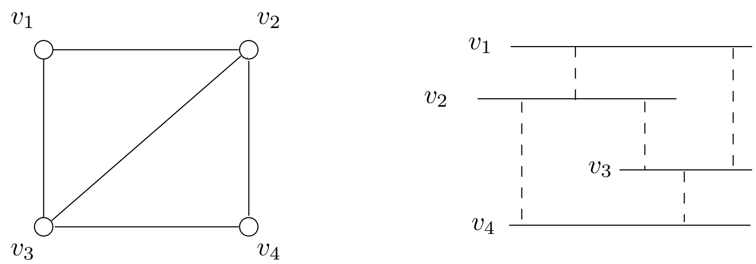


Figure 5.4: An example of a weak visibility representation of a planar graph.

For an example of a weak visibility representation of a planar graph, see Figure 5.4. In this figure, vertex segments are represented by solid horizontal lines, while edge segments are represented by dashed vertical lines. It is clear that a graph that admits a weak visibility representation is a planar graph [82]. Duchet *et al.* [25] further proved that every planar graph has a weak visibility representation[6].

5.3.5 Balanced Parentheses

Munro and Raman [71] showed how to succinctly represent a balanced parenthesis sequence S of length $2n$, where there are n opening parentheses and n closing parentheses, to support the following operations:

- $\texttt{rank\_open}_S(i)$ ($\texttt{rank\_close}_S(i)$), the number of opening (closing) parenthesis in the sequence up to (and including) position i;
- $\texttt{select\_open}_S(i)$ ($\texttt{select\_close}_S(i)$), the position of the i^{th} opening (closing) parenthesis in the sequence;
- $\texttt{find\_close}_S(i)$ ($\texttt{find\_open}_S(i)$), the matching closing (opening) parenthesis for the opening (closing) parenthesis at position i;
- $\texttt{excess}_S(i)$, the number of opening parentheses minus the number of closing parentheses in the sequence up to (and including) position i;

[6]Duchet *et al.* [25] adopted the notion of S-representation, which is identical to weak visibility representation.

- `enclose`$_S(i)$, the closest enclosing (matching parenthesis) pair of a given matching parenthesis pair whose opening parenthesis is at position i.

The subscript S is omitted when it is clear from the context. Their result is:

Lemma 5.4 ([71]). *A sequence of balanced parentheses S of length $2n$ can be represented using $2n+o(n)$ bits to support the operations* `rank_open`*,* `rank_close`*,* `select_open`*,* `select_close`*,* `find_close`*,* `find_open`*,* `excess` *and* `enclose` *in constant time.*

5.4 New Operations Based on Tree Covering (TC)

We now extend the succinct tree representation proposed by Geary *et al.* [36, 37] to support more operations on ordinal trees. We achieve this by constructing $o(n)$-bit auxiliary data structures in addition to their main data structures listed in Section 5.3.1 that use $2n+o(n)$ bits. Recall that we denote a node by its preorder number. As the conversion between the preorder number and the τ-name of a given node can be done in constant time [36, 37], we omit the steps of performing such conversions in our algorithms (e.g. we may return the τ-name of a node directly when we need return its preorder number). We use T to denote the (entire) ordinal tree.

5.4.1 `height` in $O(1)$ Time with $o(n)$ Extra Bits

We first give the following definition.

Definition 5.2. *Node x is a* **tier-1** *(or* **tier-2***)* **preorder changer** *if $x = 1$, or if nodes x and $(x-1)$ are in different mini-trees (or micro-trees).*

For example, in Figure 5.3, nodes 1, 2, 16, 22, and 30 are tier-1 preorder changers. Nodes 16, 17, 20, 22, 26 and others are tier-2 preorder changers. It is obvious that all the tier-1 preorder changers are also tier-2 preorder changers. In order to bound the number of tier-1 (or tier-2) preorder changers, we first prove the following lemma.

Lemma 5.5. *If node x is a tier-1 (or tier-2) preorder changer that is not the root of any mini-tree (or micro-tree), then node $(x-1)$ is the last node of its mini-tree (or micro-tree) in preorder.*

Proof. We only prove the theorem in the case when node x is a tier-1 preorder changer, and the case when node x is a tier-2 preorder changer can be handled similarly.

Let $i = \tau_1(x-1)$ and $j = \tau_1(x)$. Then mini-trees μ^i and μ^j contain nodes $(x-1)$ and x, respectively. Let r be the root of μ^j. Then $r < x$. As nodes $(x-1)$ and x are in two different mini-trees, we have $r < x-1$. There are two cases.

In the first case, node r is also the root of μ^i. Then, by the tree covering algorithm, all the nodes of mini-tree μ^i are before node x in preorder. Therefore, node $(x-1)$ is the last node in μ^i in preorder.

In the second case, node r is not the root of μ^i. Then μ^i and μ^j do not intersect. As μ^i contains node $(x-1)$ and μ^j contains node x, by the tree covering algorithm, either all the nodes of μ^i appears before those of μ^j in preorder, or the root of μ^i, s, is a descendant of a leaf node of μ^j that appears before x in preorder. If the former is true, then x is the root of μ^j, which is a contradiction. Thus s is a descendant of a leaf node of μ^j whose preorder number is smaller than x. By the tree covering algorithm, all the nodes of μ^i are before node x in preorder. Therefore, node $(x-1)$ is the last node in μ^i. □

We have the following lemma to bound the number of tier-1 (or tier-2) preorder changers.

Lemma 5.6. *The total number of tier-1 (or tier-2) preorder changers in any tree is at most twice the number of mini-trees (or micro-trees).*

Proof. We only show how to bound the number of tier-1 preorder changers, and the number of tier-2 preorder changers can be bounded similarly.

We first present a method to map each tier-1 preorder changer to a tier-1 preorder boundary node (see Section 5.3.1 for the definition of preorder boundary nodes). To map a tier-1 preorder changer x to a tier-1 preorder boundary node, there are two cases:

1. Node x is the root of the mini-tree it is in. In this case, we map x to itself, as it is also a tier-1 preorder boundary node.

2. Node x is not the root of the mini-tree it is in. In this case, by Lemma 5.5, node $(x-1)$ is the last node of the mini-tree it is in. Hence node $(x-1)$ is a tier-1 preorder boundary node. We map x to node $(x-1)$.

For example, in Figure 5.3, the tier-1 preorder changers 1, 2, 16, 22, and 30 are respectively mapped to the following tier-1 preorder boundary nodes: 1, 2, 15, 22 and 29. As seen above, our approach maps each tier-1 preorder changer to either itself or the node immediately preceding it in preorder. Therefore, at most two different tier-1 preorder changers can be mapped to the same tier-1 boundary node. As each mini-tree has at most two boundary nodes, this implies that the number of tier-1 preorder changers is at most four times the number of mini-trees.

To further prove that the number of tier-1 preorder changers is at most twice the number of mini-trees, we observe that each mini-tree of size larger than 1 has exactly two tier-1 boundary nodes. Hence it suffices to prove that if two preorder changers, y and z, are mapped to the same tier-1 preorder boundary node u under this mapping, then the mini-tree that u is in has only 1 node. We observe that y are z are mapped to the same tier-1 preorder boundary node only if they are visited consecutively during a preorder traversal. Without loss of generality, we assume that $y = z - 1$. By the description of the mapping, we conclude that $u = y$ and that y is the root of its mini-tree. We also have that z is not the root of its mini-tree. As z is a tier-1 preorder changer, by Lemma 5.5, we conclude that node y is the last node in preorder of the mini-tree it is in. As y is also the first node in preorder of its mini-tree, we conclude that the mini-tree that y is in is of size 1. This completes the proof. ☐

We now design auxiliary data structures to support `height`. We have the following lemma.

Lemma 5.7. *Using $o(n)$ additional bits, operation* `height` *can be supported in $O(1)$ time on* `TC`.

Proof. To compute $\texttt{height}(x)$, we compute x's number of descendants, d, using `nbdesc` [36, 37] (see Section 5.2.1). Then all the descendants of x are nodes $x+1, x+2, \cdots, x+d$. We have the formula: $\texttt{height}(x) = \max_{i=1}^{d}(\texttt{depth}(x+i)) - \texttt{depth}(x) + 1$. Therefore, the computation of $\texttt{height}(x)$ can be reduced to answering the range maximum query (see Section 5.3.2) on the conceptual array $D[1..n]$, where $D[j] = \texttt{depth}(j)$ for $1 \le j \le n$. For any node j, we can compute $\texttt{depth}(j)$ in $O(1)$ time [36, 37]. We now show how to support the range maximum query on D using $o(n)$ additional bits without storing D explicitly.

We construct the $o(n)$-bit auxiliary data structure of Lemma 5.2. We can use it to provide constant-time support for the range minimum/maximum query without accessing the array, when the starting and ending positions of the range are multiples of $\lg n$. To support the general case without explicitly storing the array D, we construct (we assume that the i^{th} tier-1 and tier-2 preorder changers are numbered y_i and z_i, respectively):

- A bit vector $B_1[1..n]$, where $B_1[i] = 1$ iff node i is a tier-1 preorder changer;
- A bit vector $B'_1[1..n]$, where $B'_1[i] = 1$ iff node i is a tier-2 preorder changer;
- An array $C_1[1..l_1]$ (l_1 denotes the number of tier-1 preorder changers), where $C_1[i] = \tau_1(y_i)$;
- An array $C'_1[1..l'_1]$ (l'_1 denotes the number of tier-2 preorder changers), where $C'_1[i]$ stores a pair of items $< \tau_2(z_i), \tau_3(z_i) >$;
- An array $E[1..l'_1]$, where $E[i]$ is the τ_3-name of the node, e_i, with maximum depth among the nodes between z_i and z_{i+1} (including z_i but excluding z_{i+1}) in preorder (we also consider the conceptual array $E'[1..l'_1]$, where $E'[i] = \texttt{depth}(e_i)$, but we do not store E' explicitly);
- A two-dimensional array M, where $M[i, j]$ stores the value δ such that $E'[i + \delta]$ is the maximum between and including $E'[i]$ and $E'[i + 2^j]$, for $1 \leq i < l'_1$ and $1 \leq j \leq \lceil \lg \lg n \rceil$;
- A table A_1, in which for each pair of nodes in each possible micro-tree, we store the τ_3-name of the node of the maximum depth between (inclusive) this pair of nodes in preorder. This table could be used for several trees. We show how to implement table A_1 later in this proof.

There are $O(n/\lg^4 n)$ tier-1 and $O(n/\lg n)$ tier-2 preorder changers, so B_1 and B'_1 can be stored in $o(n)$ bits using Part (b) of Lemma 2.1. C_1, C'_1 and E can also be stored in $o(n)$ bits (each element of C'_1 in $O(\lg \lg n)$ bits). As $M[i, j] \leq 2^{\lceil \lg \lg n \rceil}$, we can store each $M[i, j]$ in $\lceil \lg \lg n \rceil$ bits, so M takes $O(n/\lg n \times \lg \lg n \times \lg \lg n) = o(n)$ bits. To encode A_1, we use a modified version of the implicit representation of the micro-tree (see

Section 5.3.1 for a description of the implicit representations of micro-trees), together with the τ_3 names of any two pairs of nodes in the micro-tree, to index A_1. As each micro-tree has at most $3M'$ nodes, we can uniquely encode a micro-tree of m nodes using exactly $6M'$ bits, by inserting $6M' - 2m$ opening parentheses before its balanced parenthesis sequence. Thus the total number of entries of A_1 is $2^{6M'} \times 3M' \times 3M' = O(n^{1/4} \lg^2 n)$. Note that some combinations of micro-trees and the τ_3 names of the pairs of nodes are invalid. For example, some parenthesis sequences of length $6M'$ do not correspond to any micro-tree, or a τ_3 name may be greater than the size of the micro-tree. We store a -1 in each entry of A_1 corresponding to an invalid combination, and store the τ_3-names of the answers for the rest. Thus each entry occupies $O(\lg n)$ bits. Therefore, A_1 occupies $O(n^{1/4} \lg^3 n) = o(n)$ bits. Hence these auxiliary structures occupy $o(n)$ bits.

With the above auxiliary data structures, we can support the range maximum query on D. Given a range $[i, j]$, we divide it into up to three subranges: $[i, \lceil i/\lg n \rceil \lg n]$, $[\lceil i/\lg n \rceil \lg n, \lfloor j/\lg n \rfloor \lg n]$ and $[\lfloor j/\lg n \rfloor \lg n, j]$. The result is the largest among the maximum values of these three subranges. The range maximum query on the second subrange is supported by Lemma 5.2, so we consider only the first (the query on the third one, and the case where $[i, j]$ is indivisible using this approach can be supported similarly).

To support range maximum query for the range $[i, \lceil i/\lg n \rceil \lg n]$, we first use B'_1 to check whether there is a tier-2 preorder changer in this range. If not, then all the nodes in the range are in the same micro-tree. We use μ^k_l to denote this micro-tree. Then $k = C_1[\mathtt{bin\_rank}_{B_1}(1, i)]$ and l is the first item of the pair stored in $C'_1[\mathtt{bin\_rank}_{B'_1}(1, i)]$. The implicit representation of a micro-tree can be computed from its corresponding extended micro-tree in constant time. We further insert a number of opening parentheses before its balanced parenthesis representation (this can be done using a shift operation), and use it with the τ_3-names of the nodes i and $\lceil i/\lg n \rceil \lg n$ to index into the table A_1. This way we can retrieve the result in constant time.

If there are one or more tier-2 preorder changes in $[i, \lceil i/\lg n \rceil \lg n]$, let node z_u be the first one and z_v be the last. We further divide this range into three subranges: $[i, z_u]$, $[z_u, z_v)$ and $[z_v, \lceil i/\lg n \rceil \lg n]$. We can compute the maximum values in the first and the third subranges using the method described in the last paragraph, as the nodes in either of them are in the same micro-tree. To perform range maximum query on D with the range $[z_u, z_v)$, by the definition of E', we only need perform the query on E' with range $[u, v-1]$. we observe that $[u, v) = [u, u+2^s) \cup [v-2^s, v)$, where $s = \lceil \lg(v-1-u) \rceil$. As $v - u < z_v - z_u < \lg n$, we have $s \leq \lceil \lg \lg n \rceil$. Thus using $M[u, s]$ and $M[v-2^s, s]$, we can retrieve from E the τ_3-names of the nodes corresponding to the maximum values of E' in $[u, u+2^s]$ and $[v-2^s, v]$, respectively. To retrieve the τ_2-names and τ_3-names of these two nodes, we observe that it suffices to compute the τ_2-names and τ_3-names of the nodes z_u and z_{v-2^s}. As they correspond to the u^{th} 1 and the $(v-2^s)^{\text{th}}$ 1 in B'_1, we can easily compute their τ_2-names and τ_3-names using B_1, B'_1, C_1 and C'_1. Between the two nodes retrieved using the above process, the one with the larger depth is the node with the maximum depth in range $[z_u, z_v]$. $\square$

5.4.2 `LCA` and `distance` in $O(1)$ Time with $o(n)$ Extra Bits

We now show how to support `LCA` on `TC`.

Lemma 5.8. *Using $o(n)$ additional bits, operation* `LCA` *can be supported in $O(1)$ time on* `TC`.

Proof. We precompute a tier-1 macro tree as follows. First remove any node that is not a mini-tree root. For any two remaining nodes x and y, there is an edge from x to y iff among the remaining nodes, x is the nearest ancestor of y in T. The tier-1 macro tree has $O(n/\lg^4 n)$ nodes, and we store it using Lemma 5.3 in $O(n/\lg^4 n \times \lg n) = o(n)$ bits. For each mini-tree root of T, we also store its preorder number in the tier-1 macro tree. This also takes $O(n/\lg^4 n \times \lg n) = o(n)$ bits.

Similarly, for each mini-tree, we precompute a tier-2 macro tree which only has the micro-tree roots. Each tier-2 macro tree has $O(\lg^3 n)$ nodes, and we store it using Lemma 5.3 in $O(\lg^3 n \lg\lg n)$ bits. For the root of each mini-tree, we also store its preorder number in the corresponding tier-2 macro tree. This also takes $O(\lg^3 n \lg\lg n)$ bits for each mini-tree. As there are $O(n/\lg^4 n)$ mini-trees, the overall space used is $O(n \lg\lg n/\lg n) = o(n)$ bits.

We also construct a table A_2 to store, for each possible micro-tree and each pair of nodes in it (indexed by their τ_3-names), the τ_3-name of their lowest common ancestor. Similarly to the analysis in the proof of Lemma 5.7, A_2 occupies $o(n)$ bits.

Algorithm `LCA`(x, y)

1. If x and y are in the same micro-tree, retrieve their `LCA` using a constant-time lookup on A_2 and return.
2. If x and y are not in the same micro-tree, but are in the same mini-tree, retrieve the roots, u and v, of the micro-trees that x and y are in, respectively.
3. If $u = v$, return u as the result.
4. If $u \neq v$, retrieve their lowest common ancestor, w, in the tier-2 macro tree.
5. Retrieve the two children, i and j, of w in the tier-2 macro tree that are ancestors of x and y, respectively using `depth` and `level_anc`. Then retrieve the parents, k and l, of i and j in T, respectively.
6. If k and l are in two different micro-trees, return w as the result. Otherwise, return `LCA`(k, l).
7. If x and y are in two different mini-trees, retrieve the roots, p and q, of the two different mini-trees, respectively.
8. If $p = q$, return p as the result. Otherwise, similarly to Steps 4 and 5, retrieve two nodes a and b, such that a and b are the children of the lowest common ancestor, c, of p and q in the tier-1 macro tree, and they are also the ancestors of p and q, respectively. Retrieve the parents, r and s, of p and q in T, respectively.
9. If r and s are in two different mini-trees, return c. Otherwise, return `LCA`(r, s).

Figure 5.5: An algorithm for computing `LCA`.

Figure 5.5 presents the algorithm to compute `LCA`. The correctness is straightforward and it clearly takes $O(1)$ time. □

With the support for `LCA` and `depth`, the support for `distance` is trivial.

Corollary 5.1. *Operation* `distance` *can be supported on* `TC` *in* $O(1)$ *time.*

Proof. To compute $\texttt{distance}(x, y)$, first compute $z = \texttt{LCA}(x, y)$. Then use $\texttt{distance}(x, y) = (\texttt{depth}(x) - \texttt{depth}(z)) + (\texttt{depth}(y) - \texttt{depth}(z))$ to compute the result. □

5.4.3 `leftmost_leaf` and `rightmost_leaf` in $O(1)$ Time

Lemma 5.9. *Operations* `leftmost_leaf` *and* `rightmost_leaf` *can be supported on* `TC` *in* $O(1)$ *time.*

Proof. given a node x with preorder number i, postorder number j, and m descendants, we observe that the postorder number of its left-most leaf is $j - m$, and the preorder number of its rightmost leaf is $i + m$. Thus the support of these two operations follows the constant-time support for `nbdesc`, $\texttt{node\_rank}_{\texttt{PRE}}$, $\texttt{node\_rank}_{\texttt{POST}}$, $\texttt{node\_select}_{\texttt{PRE}}$ and $\texttt{node\_select}_{\texttt{POST}}$. □

5.4.4 `leaf_rank` and `leaf_size` in $O(1)$ Time with $o(n)$ Extra Bits

We first give the following definition.

Definition 5.3. *Each leaf of a mini-tree (or micro-tree) is a* **pseudo leaf** *of the original tree T. A pseudo leaf that is also a leaf of T is a* **real leaf**. *Given a mini-tree (or micro-tree), we mark the leftmost real leaf of the mini-tree (or micro-tree), and the first real leaf in preorder after each subtree of T rooted at a node that is not in the mini-tree (or micro-tree), but is a child of a node in it. These nodes are called* **tier-1** *(or* **tier-2***)* **marked leaves**.

For example, in Figure 5.3, nodes 6, 11 and 15 are pseudo leaves of micro-tree μ_1^2, among which nodes 6 and 15 are real leaves, while node 11 is not. Nodes 23 and 29 are tier-2 marked leaves. We observe the following property of the real leaves of a mini-tree (or micro-tree).

Property 5.1. *Given a mini-tree (or micro-tree) and a pair of its tier-1 (or tier-2) marked leaves such that there is no marked leaf between them in preorder, the real leaves visited in preorder between these two leaves (including the left one but excluding the right) have the property that, when listed from left to right, their* `leaf_rank`*s are consecutive integers. The real leaves that are to the right of (and including) the rightmost marked leaf have the same property.*

Geary *et al.* [36, 37] showed how to convert the preorder number of a given node to its τ-name. The algorithm and auxiliary data structures can be easily adapted to solve the easier problem of converting the preorder number of a given node to its preorder rank in its mini-tree.

Lemma 5.10. *Given a node x, its preorder number in its mini-tree can be computed in $O(1)$ time using $o(n)$ additional bits.*

Proof. We use the bit vector B_1 constructed in the proof of Lemma 5.7. In addition, we construct an array $P[1..l_1]$ (l_1 denotes the number of tier-1 preorder changers), where $P[i]$ stores the preorder number of y_i (y_i denotes the i^{th} tier-1 preorder changer) in its mini-tree. Thus P uses $O(n/\lg^4 n \times \lg\lg n) = o(n)$ bits. To compute the preorder number of x in its mini-tree, we first check whether x is a tier-1 preorder changer using B_1. If it is, then $P[\texttt{bin\_rank}_{B_1}(1, x)]$ is the answer. Otherwise, we locate the nearest tier-1 preorder changer y preceding x in preorder in constant time, using $y = \texttt{bin\_select}_{B_1}(1, \texttt{bin\_rank}_{B_1}(1, x))$. As x, y and the nodes visited between them in a preorder traversal are in the same mini-tree, we conclude that the preorder number of x in its mini-tree is $P[\texttt{bin\_rank}_{B_1}(1, x)]+x-y$. $\square$

Now we can present our result on supporting `leaf_rank`.

Lemma 5.11. *Using $o(n)$ additional bits, operation* `leaf_rank` *can be supported in $O(1)$ time on* `TC`.

Proof. For each mini-tree μ^i, we store the ranks of its tier-1 marked leaves in an array M_i, sorted by their preorder numbers. We also construct a bit vector K_i, whose length is the size of the mini-tree. The j^{th} bit of K_i is 1 iff the j^{th} node of μ^i in preorder is a tier-1 marked leaf. Observe that each tier-1 marked leaf (except perhaps the first real leaf in a mini-tree) corresponds to a distinct edge that leaves its mini-tree (which in turn corresponds to a distinct mini-tree root). Hence the number, m_1, of tier-1 marked leaves is at most twice as many as the number of mini-trees, which is $O(n/\lg^4 n)$. Therefore, the total space cost of all the M_is is $O(n/\lg^4 n \times \lg n) = o(n)$. We assume that μ^i has t_i nodes and l_i tier-1 marked leaves, and that there are n_1 mini-tree in total, then the total size of all the K_is (constructed using Part (b) of Lemma 2.1) is $\sum_{i=1}^{n_1} \lceil \log_2 \binom{t_i}{l_i} \rceil \leq \sum_{i=1}^{n_1} \log_2 \binom{t_i}{l_i} + n_1 \leq \lg \binom{n}{m_1} + n_1 = o(n)$.

Similarly, the number of tier-2 marked leaves is at most twice the number of micro-trees. For each micro-tree and each of its tier-2 marked leaf x, let y be the last tier-1 marked leaf visited before x during a preorder traversal (if x is a tier-1 marked leaf then set $y = x$). We compute the difference of $\texttt{leaf\_rank}(x)$ and $\texttt{leaf\_rank}(y)$. As we do not visit any node outside the mini-tree after y and before x in a preorder traversal, this value is $O(\lg^4 n)$. For each micro-tree μ^i_j, we store the difference values computed in the above way for its tier-2 marked leaves in an array D^i_j, sorted by the preorder numbers of the tier-2 marked leaves. We also construct a bit vector L^i_j, whose length is the size of μ^i_j. The k^{th} bit of L^i_j is 1 iff the k^{th} node of the micro-tree in preorder is a tier-2 marked leaf. The total space cost of all the D^i_js is $O(n \lg\lg n/\lg n)$. Similarly to the analysis of the space cost of all the bit vector K_is, we also have that the total space cost of bit vector L^i_js is $o(n)$.

In order to locate each D^i_j and each L^i_j, we need additional data structures, as we cannot afford storing pointers indicating their addresses in storage for each micro-tree (we can afford doing so for each mini-tree to locate the M_is and K_is though, as there are $O(n/\lg^4 n)$ mini-trees). We concatenate all the D^i_js, sorted by the lexicographic order of their micro-trees (we treat each micro-tree μ^i_j as a pair $< i, j >$ when sorting the micro-trees by lexicographic order), and store the resulting array, D, instead of storing each D^i_j separately. To locate the starting and ending positions of each D^i_j in D, we construct an additional bit vector D' to encode the number of elements of each D^i_j in unary. More precisely, if D^i_j has k elements (i.e. if there are k tier-2 marked leaves in micro-tree μ^i_j), we represent it by $1^{k-1}0$, and concatenate all such encodings by the lexicographic order of the corresponding micro-trees to construct D'. Thus the length of D' is the number of the tier-2 marked leaves in T, and the number of 1s in it is the number of micro-trees. Thus it can be stored in $o(n)$ bits using Part (b) of Lemma 2.1. To locate the starting and ending positions of each D^i_j in D, it suffices to computer the number, p, of micro-trees preceding

μ_j^i in lexicographic order, since $D_j^i = D[\texttt{bin\_select}_{D'}(0, p-1)+1, \texttt{bin\_select}_{D'}(0, p)]$. We store for each mini-tree μ^i the number of micro-trees in mini-trees $\mu^1, \mu^2, \cdots, \mu^{i-1}$, and this takes $O(n/\lg^4 n \times \lg n) = o(n)$ bits in total. Thus p is equal to the sum of j and this value stored in μ^i. The bit vectors L_j^i can be stored in a similar manner using $o(n)$ bits so that each of them can be located in constant time.

We construct a table A_4. For each possible micro-tree and each pair of its nodes (denoted by their τ_3-names), we store the number of pseudo leaves of the micro-tree between them. Similarly to the analysis in the proof of Lemma 5.7, we have that the space used by A_4 is $o(n)$ bits.

With the above data structures of $o(n)$ bits, we can now compute $\texttt{leaf\_rank}(x)$. We can assume that x is a leaf, because otherwise $\texttt{leaf\_rank}(x) = \texttt{leaf\_rank}(\texttt{leftmost\_leaf}(x))$. We assume that $\tau(x) =< i, j, k >$. We first compute the preorder number, r, of x in its mini-tree using Lemma 5.10. If $K_i(r) = 1$, then x is a tier-1 marked leaf, and $M_i[\texttt{bin\_rank}_{K_i}(1, r)]$ is the result. Otherwise, if $L_j^i(k) = 1$, then x is a tier-2 marked leaf, and $M_i[\texttt{bin\_rank}_{K_i}(1, r)] + D_j^i[\texttt{bin\_rank}_{L_j^i}(1, k)]$ is the result. Otherwise, we locate the closest tier-2 marked leaf, y, to the left of x in μ_j^i using L_j^i, and compute the number of pseudo leaves, p, between y and x using a table lookup on A_4. As there is no edge leaving the micro-tree between y and x, all the pseudo leaves between them are real leaves. Thus by Property 5.1, $M_i[\texttt{bin\_rank}_{K_i}(1, r)] + D_j^i[\texttt{bin\_rank}_{L_j^i}(1, k)] + p$ is the result. □

With the constant-time support for `leaf_rank`, `leftmost_leaf` and `rightmost_leaf`, the following corollary is immediate.

Corollary 5.2. *Operation* `leaf_size` *can be supported on* `TC` *in $O(1)$ time.*

5.4.5 `leaf_select` in $O(1)$ Time with $o(n)$ Extra Bits

We observe the following property.

Property 5.2. *Given a leaf x that is not a tier-1 (or tier-2) marked leaf, if the closest tier-1 (or tier-2) marked leaf to the left is node y (or node z), then $\tau_1(x) = \tau_1(y)$ (or $\tau_2(x) = \tau_2(z)$).*

Based on the above property, we can support `leaf_select` as follows.

Lemma 5.12. *Using $o(n)$ additional bits, operation* `leaf_select` *can be supported in $O(1)$ time on* TC.

Proof. Assume that T has l leaves. We list all the leaves from left to right, and number them $1, 2, ..., l$. We construct the following auxiliary data structures:

- A bit vector $B_5[1..l]$, where $B_5[i] = 1$ iff the i^{th} leaf is a tier-1 marked leaf;
- A bit vector $B'_5[1..l]$, where $B'_5[i] = 1$ iff the i^{th} leaf is a tier-2 marked leaf;
- An array $C_5[1..l_5]$ (l_5 denotes the number of tier-1 marked leaves), where $C_5[i]$ is the τ_1-name of the i^{th} tier-1 marked leaf;
- An array $C'_5[1..l'_5]$ (l'_5 denotes the number of tier-2 marked leaves), where $C'_5[i]$ is of the form $< q_i, r_i >$, such that q_i and r_i are the τ_2 and τ_3-names of the i^{th} tier-2 marked leaf, respectively;
- A table A_5, in which for each possible micro-tree, each of its pseudo leaves (denoted by its τ_3-name), and each integer between 1 and $3M'$, we store the τ_3-name of the pseudo leaf whose `leaf_rank` minus that of the given pseudo leaf is equal to the given integer.

As there are $O(n/\lg^4 n)$ tier-1 marked leaves and $O(n/\lg n)$ tier-2 marked leaves, B_5, B'_5, C_5 and C'_5 occupy $o(n)$ bits. Similarly to the analysis in the proof of Lemma 5.7, we have that the space used by A_5 is $o(n)$ bits. Thus the above auxiliary data structures occupy $o(n)$ bits.

With the above auxiliary data structures, we can now support `leaf_select`. To compute `leaf_select`(i) (x denotes the result), by Property 5.2, $\tau_1(x) = C_5[\texttt{bin\_rank}_{B_5}(1, i)]$, and $\tau_2(x)$ is the first item of the pair stored in $C'_5[\texttt{bin\_rank}_{B'_5}(1, i)]$. To compute $\tau_3(x)$, if $B'_5[i] = 1$, then $\tau_3(x)$ is the second item, j, of the same pair. Otherwise, we use x's micro-tree, j, and $i - \texttt{select}_1(B'_5, \texttt{bin\_rank}_{B'_5}(1, i))$ as parameters to perform a table lookup on A_5 to compute $\tau_3(x)$. □

5.4.6 $\texttt{node\_rank}_{\texttt{DFUDS}}$ in $O(1)$ Time with $o(n)$ Extra Bits

Lemma 5.13. *Using $o(n)$ additional bits, operation $\texttt{node\_rank}_{\texttt{DFUDS}}$ can be supported in $O(1)$ time on* TC.

Proof. We use the following formula proposed by Barbay *et al.* [7]: $\texttt{node\_rank}_{\texttt{DFUDS}}(x)$
$= \begin{cases} \texttt{child\_rank}(x) - 1 + \texttt{node\_rank}_{\texttt{DFUDS}}(\texttt{child}(\texttt{parent}(x), 1)) \text{ if } \texttt{child\_rank}(x) > 1; \\ \texttt{node\_rank}_{\texttt{PRE}}(x) + \sum_y (\texttt{degree}(\texttt{parent}(y)) - \texttt{child\_rank}(y)) \text{ otherwise.} \end{cases}$
where $y \in \texttt{anc}(x) \backslash r$ (we denote the set of ancestors of x by anc(x) and the root of T by r).

This formula reduces the support of $\texttt{node\_rank}_{\texttt{DFUDS}}$ to the support of computing $S(x) = \sum_{y \in \texttt{anc}(x) \backslash r} \texttt{degree}(\texttt{parent}(y)) - \texttt{childrank}(y)$ for any given node x [7]. We use $u(x)$ and $v(x)$ to denote the roots of the mini-tree and micro-tree that node x is in, respectively. Then we compute $S(x)$ as the sum of the following three values as suggested by Barbay *et al.* [7]: $S_1(x) = S(u(x))$, $S_2(x) = S(v(x)) - S(u(x))$, and $S_3(x) = S(x) - S(v(x))$. It is trivial to support the computation of $S_1(x)$ in constant time: for each mini-tree root i, we simply precompute and store $S(i)$ using $O(n/\lg^3 n) = o(n)$ bits. However, we cannot do the same to support the computation of $S_2(x)$. This is because there are $O(n/\lg n)$ micro-trees, and we need $O(\lg n)$ bits to store $S(j)$ for each micro-tree root j. The approach of Barbay *et al.* [7] does not solve the problem, either, because Property 1 in [7] does not hold. We propose the following approach.

We extend the mini-trees using the same method used to extend micro-trees (see Section 5.3.1). Thus, similarly to Lemma 5.1, we have:

Property 5.3. *Except for the roots of mini-trees, all the other original nodes have the property that (at least a promoted copy of) each of their children is in the same extended mini-tree as themselves.*

For node x that is not the root of a mini-tree ($S_2(x) = 0$ otherwise), we further divide $S_2(x)$ into two parts. Assume that $w(x)$ is the child of $u(x)$ that is also an ancestor of x ($w(x)$ can be computed in constant time using `depth` and `level_anc`). Then $S_2(x)$ is equal to the sum of $S(w(x)) - S(u(x))$ and $S(v(x)) - S(w(x))$ (denoted by $S'_2(x)$). As $S(w(x)) - S(u(x)) = \texttt{degree}(u(x)) - \texttt{childrank}(w(x))$, we can compute it in constant time. By Property 5.3, we conclude that $S'_2(x)$ is at most as large as the size of the extended mini-tree that x is in. We categorize extended mini-trees into two types: small extended mini-trees whose size is at most $\lg^5 n$, and large extended mini-trees whose size is greater than $\lg^5 n$. As each large extended mini-tree has $O(\lg^5 n)$ promoted nodes, and there are $O(n/\lg^4 n)$ promoted nodes in T, we have that there are $O(n/\lg^9 n)$ large extended mini-trees, which have $O(n/\lg^5 n)$ original nodes in total. For the roots of the micro trees that are original nodes of small extended mini-trees, we need $O(n \lg\lg n/\lg n) = o(n)$ bits to store their corresponding S'_2 values. For the roots of the micro trees that are original nodes of large extended mini-trees, we need $O(n/\lg^5 n \times \lg n) = o(n)$ bits to store their corresponding S'_2 values. An additional bit vector of $o(n)$ bits can tell us in which type of extended mini-tree a micro-tree root is presented. Thus we can support the constant-time computation of $S_2(x)$.

The constant-time support for the computation of $S_3(x)$ is similar. We also reduce the computation of $S_3(x)$ to the computation of $S'_3(x) = S(x) - S(q(x))$, where $q(x)$ is the child of $v(x)$ that is also an ancestor of x. $S'_3(x)$ is bounded by the size of the extended micro-tree that x is originally in. We categorize extended micro trees into three types: small extended micro-trees, whose size is at most $\frac{1}{4}\lg n$; medium extended micro-trees, whose size is greater than $\frac{1}{4}\lg n$, but not greater than $\lg^2 n$; and large extended micro-trees, whose size is greater than $\lg^2 n$. Similarly, there are $O(n/\lg n)$ original nodes in medium micro trees, and $O(n/\lg^2 n)$ original nodes in large micro trees. For all the small micro-trees, we can use an $o(n)$-bit table to support constant-time lookup of the S'_3 values of each node in it. For all the medium micro trees, we can store the S'_3 values of all their original nodes using $O(n \lg\lg n/\lg n) = o(n)$ bits. For all the large micro-trees, we can store the S'_3 values of all their original nodes using $O(n/\lg^2 n \times \lg n) = o(n)$ bits. Thus we can support the computation of $S_3(x)$ in constant time. □

5.4.7 $\texttt{node\_select}_{\texttt{DFUDS}}$ in $O(1)$ Time with $o(n)$ Extra Bits

In this section, we first define the τ^*-name of a node and show how to convert τ^*-names to τ-names. Then we show how to convert DFUDS numbers to τ^*-names. We define the τ^*-name of a node as follows:

Definition 5.4. *Given a node x whose τ-name is $\tau(x) =< \tau_1(x), \tau_2(x), \tau_3(x) >$, its* **$\tau^*$-name** *is $\tau^*(x) =< \tau_1(x), \tau_2(x), \tau^*_3(x) >$, if x is the $\tau^*_3(x)^{th}$ node of its micro-tree in* DFUDS *order.*

For example, in Figure 5.3, node 29 has τ-name $< 3, 1, 5 >$ and τ^*-name $< 3, 1, 4 >$. To convert the τ^*-name of a node to its τ-name, we have:

Lemma 5.14. *Given the τ^*-name of a node x, $\tau(x)$ can be computed in $O(1)$ time using $o(n)$ additional bits.*

Proof. We only need compute $\tau_3^*(x)$ and use table lookup for this purpose. For every micro-tree, and for every node (numbered by its DFUDS number) in the tree, we store its corresponding preorder number. Similarly to the analysis in the proof of Lemma 5.7, we have that the space used by this table is $o(n)$ bits. $\square$

The idea of computing the τ^*-name given a `DFUDS` number is to store the τ^*-names of some of the nodes, and compute the τ^*-names of the rest using these values. It is similar to the algorithm in Section 4.3.1 of [36, 37] to support $\texttt{node\_select}_{\texttt{PRE}}$. However, as we cannot define boundary nodes for `DFUDS` traversal, it is not trivial to apply the algorithm. We begin with the following definition.

Definition 5.5. *List the nodes in* `DFUDS` *order, numbered $1, 2, ..., n$. The i^{th} node in* `DFUDS` *order is a* **tier-1** *(or* **tier-2***)* `DFUDS` **changer** *if $i = 1$, or if the i^{th} and $(i-1)^{th}$ nodes in* `DFUDS` *order are in different mini-trees (or micro-trees).*

For example, in Figure 5.3, nodes 1, 2, 16, 3, 17, 22 and 30 are tier-1 `DFUDS` changers, and nodes 2, 16, 3, 6, 7, 11, 4 and others are tier-2 `DFUDS` changers. It is obvious that all the tier-1 `DFUDS` changers are also tier-2 `DFUDS` changers. We have the following lemma.

Lemma 5.15. *The number of tier-1 (or tier-2)* `DFUDS` *order changers in any tree is at most four times the number of mini-trees (or micro-trees).*

Proof. We only show how to bound the number of tier-1 `DFUDS` changers, and the number of tier-2 `DFUDS` changers can be bounded similarly.

List the mini-trees $t_1, t_2, \cdots$ in an order such that in a `DFUDS` traversal of T, either the root of t_i is visited before that of t_{i+1}, or if these two mini-trees share the same root, the children of the root of t_i are visited before the children of the root of t_{i+1}. We call the number assigned to each mini-tree in the above way the `DFUDS` number of each mini-tree.

In the rest of this proof, for simplicity, we use mini-tree i to refer to the mini-tree with DFUDS number i.

For a tier-1 DFUDS changer x, if it is in mini-tree i, and the node visited immediately before x in DFUDS traversal is in mini-tree j, we say that x is *related to* mini-trees i and j (if $x = 1$, then we set j to 0). We also say that x is *associated with* mini-tree $\max(i, j)$. We have:

Property 5.4. *If a tier-1* DFUDS *changer x is related to mini-trees i and j, and is associated with mini-tree i, then the root of mini-tree j cannot be a descendant of the root of mini-tree i.*

The correctness of this property is clear, because otherwise, we have $j > i$. Then x is associated with mini-tree j, which is a contradiction.

In the above notation, each tier-1 DFUDS changer is related to at most two mini-trees, but is associated with exactly one mini-tree. Therefore, we only need prove that for each mini-tree i, there are at most 4 changers that are associated with it. There are three cases.

Case 1: the root of mini-tree i is not shared with any other mini-tree. We first locate the DFUDS changers that are related to mini-tree i. It is clear that the root of mini-tree i (denoted by x) is related to it. In a DFUDS traversal, after visiting x, we visit x's right siblings if they exist, and then the descendants of x's left siblings if they exist. There nodes are outside mini-tree i. Thus the first node among these nodes (if they exist) in DFUDS order is a DFUDS changer related to mini-tree i. We then visit x's descendants, so the leftmost child of x (if x is not a leaf) is related to mini-tree i if it is a DFUDS changer (i.e. if x has right siblings or at least one of x's left siblings is not a leaf node) and it is in mini-tree i. When we visit x's descendants in DFUDS order, we may visit mini-trees whose roots are x's descendants, and this results in more DFUDS changers that are related to mini-tree i. Note that by Property 5.4, these nodes are not associated with mini-tree i. After visiting all of x's descendants that are in mini-tree i, the nodes we visit are outside this mini-tree. Thus the first node among these nodes (if there are any) in DFUDS order is also related to mini-tree i. Therefore, the only possible tier-1 DFUDS changers associated with mini-tree i are (note that not all of they are necessarily associated with mini-tree i, and some of them may not even be tier-1 DFUDS changers or may not even exist):

- Node x, the root of mini-tree i as defined above;
- Node y, the leftmost right sibling of x if it exists, or otherwise, node z, the first node visited in a `DFUDS` traversal among all the descendants of the left siblings of x;
- Node u, the leftmost child of x in mini-tree i;
- Node v, the node visited immediately after all the descendants of x are visited in a `DFUDS` traversal.

Case 2: the root of mini-tree i is shared with other mini-trees, but the `DFUDS` number of any other tree that shares the same root is larger than i. In this case, similarly to the analysis in the first case, there are at most 3 tier-1 `DFUDS` changers associated with mini-tree i. They are node x, either node y or node z, and node u, as defined in Case 1. Note that in this case, neither node v (defined above) or the leftmost right sibling of u is associated with mini-tree i, although they are related to mini-tree i.

Case 3: the root of mini-tree i is shared with another mini-tree whose `DFUDS` number is smaller than i. In this case, similarly to the analysis for the above two cases, there are at most 4 tier-1 `DFUDS` changers that may be associated with mini-tree i. They are:

- Node p, the leftmost child of x in mini-tree i;
- Node q, the first node visited in a `DFUDS` traversal among all the descendants of the left siblings of p;
- Node w, the leftmost child of p in mini-tree i;
- Node v, defined in Case 1. □

With the above definition and lemma, we can now support $\texttt{node\_select}_{\texttt{DFUDS}}$.

Lemma 5.16. *Using $o(n)$ additional bits, operation* $\texttt{node\_select}_{\texttt{DFUDS}}$ *can be supported in $O(1)$ time on* `TC`.

Proof. We design the following auxiliary data structures.

- A bit vector $B_7[1..n]$, where $B_7[i] = 1$ iff the i^{th} node in `DFUDS` order is a tier-1 `DFUDS` changer;

- A bit vector $B'_7[1..n]$, where $B'_7[i] = 1$ iff the i^{th} node in DFUDS order is a tier-2 DFUDS changer;
- An array $C_7[1..m_7]$ (m_7 denotes the number of tier-1 DFUDS changers), where $C_7[i]$ is the τ_1-name of the i^{th} tier-1 DFUDS changer in DFUDS order;
- An array $C'_7[1..m'_7]$ (m'_7 denotes the number of tier-2 DFUDS changers), where $C'_7[i]$ is of the form $< q_i, r_i >$, such that q_i and r_i are the τ_2 and τ_3-names of the i^{th} tier-2 DFUDS changer in DFUDS order.

The number of 1s in B_7 is m_7, which is $O(n/\lg^4 n)$ by Lemma 5.15. We can store B_7 using Part (b) of Lemma 2.1 in $O(\lg \binom{n}{n/\lg^4 n}) = o(n)$ bits. Similarly, $m'_7 = O(n/\lg n)$ and we can store B'_7 in $o(n)$ bits. C_7 occupies $O(n/\lg^4 n \times \lg n) = o(n)$ bits. C'_7 occupies $O(n/\lg n \times \lg\lg n) = o(n)$ bits. Therefore, the above data structures occupy $o(n)$ bits.

To support $\texttt{node\_select}_{\texttt{DFUDS}}(i)$, by Lemma 5.14, we only need compute the τ^*-name of the i^{th} node in DFUDS order. We use $< t_1, t_2, t_3 >$ to denote the result. By the definition of tier-1 DFUDS changers, we immediately have that t_1 is equal to the τ_1-name of the last tier-1 DFUDS changer up to and including the i^{th} node in DFUDS order, which is $C_7[\texttt{bin\_rank}_{B_7}(1, i)]$. Similarly, t_2 is equal to the τ_2-name of the last tier-2 DFUDS changer up to and including the i^{th} node in DFUDS order, which is the first item of the pair stored in $C'_7[\texttt{bin\_rank}_{B'_7}(1, i)]$ (we use k to denote the second item of this pair, which is the τ_3-name of this tier-2 DFUDS changer). Now we only need compute t_3. We first locate the last tier-2 DFUDS changer up to the i^{th} node in DFUDS order. We assume that it is the j^{th} node in DFUDS order. Then $j = \texttt{bin\_select}_{B'_7}(1, \texttt{bin\_rank}_{B'_7}(1, i))$. Finally we have $t_3 = k + i - j$. □

5.4.8 `level_leftmost` and `level_rightmost` in $O(1)$ Time with $o(n)$ Extra Bits

We define the i^{th} *level* of a tree to be the set of nodes whose depths are equal to i in the tree.

Lemma 5.17. *Using $o(n)$ additional bits, operations* `level_leftmost` *and* `level_rightmost` *can be supported in $O(1)$ time on* TC*.*

Proof. We only show how to support `level_leftmost`; `level_rightmost` can be supported similarly.

We first show how to compute the τ_1-name, u, of the node $\texttt{level\_leftmost}(i)$. Let h be the height of T. We construct a bit vector $B_8[1..h]$, in which $B_8[j] = 1$ iff the nodes $\texttt{level\_leftmost}(j-1)$ and $\texttt{level\_leftmost}(j)$ are in two different mini-trees, for $1 < j \leq h$ (we set $B_8[1] = 1$). Let l_8 be the number of 1s in B_8, and we construct an array $C_8[1..l_8]$ in which $C_8[k]$ stores the τ_1-name of the node $\texttt{level\_leftmost}(\texttt{bin\_select}_{B_8}(1, k))$. As the τ_1-name of the node $\texttt{level\_leftmost}(i)$ is the same as that of the leftmost node at level $\texttt{bin\_select}_{B_8}(1, \texttt{bin\_rank}_{B_8}(1, i))$, we have that $u = C_8[\texttt{bin\_rank}_{B_8}(1, i)]$. To analyze the space cost of B_8 and C_8, we observe that if a given value, p, occurs q times in C_8, then the mini-tree μ^p has at least $q - 1$ edges that leave μ^p. Thus the number of 1s in C_8 that correspond to the nodes in a given mini-tree is at most the number of edges that leave the mini-tree plus 1. Therefore, l_8 is at most the number of mini-trees plus the number of edges that leave a mini-tree, which is $O(n/\lg^4 n)$. Hence B_8 and C_8 occupy $o(n)$ bits.

To support the computation of the τ_2-name, v, of the node $\texttt{level\_leftmost}(i)$, we construct a bit vector $B'_8[1..h]$, in which $B'_8[j] = 1$ iff the nodes $\texttt{level\_leftmost}(j - 1)$ and $\texttt{level\_leftmost}(j)$ are in two different micro-trees, for $1 < j \leq h$ (we set $B'_8[1] = 1$). Let l'_8 be the number of 1s in B'_8, and we construct an array $C'_8[1..l'_8]$ in which $C'_8[l]$ stores the τ_2-name of the node $\texttt{level\_leftmost}(\texttt{bin\_select}_{B'_8}(1, l))$. Similarly, we have that $l'_8 = O(n/\lg n)$, so B'_8 and C'_8 occupy $o(n)$ bits. We also have $v = C'_8[\texttt{bin\_rank}_{B'_8}(1, i)]$, so we can compute v in constant time.

To compute the τ_3-name of this node in constant time, we construct a table A_8, which stores for each possible micro-tree and each integer l in the range $[1, 3M']$, the τ-name of the leftmost node at level l. Similarly to the analysis in the proof of Lemma 5.7, A_8 occupies $o(n)$ bits. ☐

5.4.9 `level_succ` and `level_pred` in $O(1)$ Time with $o(n)$ Extra Bits

We first give the following definition.

Definition 5.6. *A* **tier-1** *(or* **tier-2***)* **preorder segment** *is a sequence of nodes $x, (x + 1), \cdots, (x + i)$ that satisfies:*

- *Node x is a tier-1 (or tier-2) preorder changer;*
- *Node $(x+i+1)$ is a tier-1 (or tier-2) preorder changer if $x+i+1 \leq n$;*
- *None of the nodes $(x+1), (x+2), \cdots, (x+i)$ is a tier-1 (or tier-2) preorder changer.*

For example, in Figure 5.3, nodes 16, 17, 18, 19, 20 and 21 form a tier-1 preorder segment. Nodes 22, 23, 24 and 25 form a tier-2 preorder segment. We sort the tier-1 (or tier-2) preorder segments by the preorder numbers of the tier-1 (or tier-2) preorder changers that they respectively contain. We denote the i^{th} tier-1 (or tier-2) preorder segment in this order to be the i^{th} tier-1 (or tier-2) preorder segment of the tree. By Definition 5.6, the following properties of preorder segments are immediate.

Property 5.5. *The following basic facts hold:*

- *The nodes in the same tier-1 (or tier-2) preorder segments are in the same mini-tree (or micro-tree);*
- *Two different tier-1 (or tier-2) preorder segments do not share any node;*
- *The number of tier-1 (or tier-2) preorder segments is equal to the number of tier-1 (or tier-2) preorder changers;*
- *A tier-1 preorder segment can be divided into one or more tier-2 preorder segments.*

We now prove two lemmas on preorder segments.

Lemma 5.18. *Consider a node x at the i^{th} level of T. Let $L(x)$ be the set of nodes such that $y \in L(x)$ iff all the following conditions are satisfied:*

- *$y > x$;*
- *Node y is at the i^{th} level of T;*
- *Node x and node y are in the same tier-1 preorder segment.*

If $L(x)$ is not empty, then the node in $L(x)$ with the smallest preorder number is x's level successor.

The same claim is true for the set $L'(x)$ consisting of the nodes that are in the tier-2 preorder segment that x is in, and that satisfy the first two conditions above.

Proof. Let z be the node in $L(x)$ with the smallest preorder number. Let the level successor of x be v. Assume, contrary to what we are going to prove, that $v \neq z$. Then, by the definition of level successor, we have $x < v < z$. Therefore, $v \notin L(x)$. As node v satisfies the first two conditions, we conclude that node x and node v are not in the same tier-1 preorder segment. However, as node x and node z are in the same tier-1 preorder segment, the above inequality ($x < v < z$) contradicts Definition 5.6.

The same reasoning applies to the set $L'(x)$. □

Lemma 5.19. *Consider two different tier-1 (or tier-2) preorder segments A and B. If there are two nodes x and y in A and B respectively, such that x and y are at the same level of T and x is to the left of y, then any node of A at a given level (if such a node exists) is to the left of all the nodes of B at the same level (again if such nodes exist).*

Proof. First it is clear that $x < y$. As all the nodes in a given tier-1 (or tier-2) preorder segment are consecutive in preorder, we conclude that the preorder number of any node in A is smaller than the preorder number of any node in B. This lemma immediately follows. □

We say that a tier-1 (or tier-2) preorder segment A is *to the left of* another tier-1 (or tier-2) preorder segment B, if there are two nodes x and y in A and B respectively, such that x and y are at the same level of T and x is to the left of y. By Lemma 5.19, such a relationship always exits between two arbitrary tier-1 (or tier-2) preorder segments that have nodes at the same level.

We now consider the problem of retrieving the leftmost node in a given tier-1 (or tier-2) preorder segment at a given level.

Lemma 5.20. *There is an $o(n)$-bit auxiliary data structure that supports, given a node x and an integer i, the computation of the leftmost node at the i^{th} level in the tier-1 (or tier-2) preorder segment that contains x in $O(1)$ time on* `TC`.

Proof. We construct the following auxiliary data structures (p_1 denotes the total number of tier-1 preorder segments):

- A table A_9, which stores for each possible micro-tree and two integers k and l in the range $[1, 3M']$, the τ-name of the leftmost node after node k in preorder that is at level l of the micro-tree;
- An array E, in which $E[j]$ stores the preorder number of the node with the smallest depth in the j^{th} tier-1 preorder segment;
- B_1, B_1', C_1 and C_1' as in the proof of Lemma 5.7;
- A bit vector B_9^j for the j^{th} tier-1 preorder segment (for $j = 1, 2, \cdots, p_1$), in which $B_9^j[u] = 1$ iff the leftmost nodes at level $(s_j + u - 2)$ and at level $(s_j + u - 1)$ (s_j and l_j denote the minimum and maximum depths of the nodes in the j^{th} tier-1 preorder segment, respectively) are in two different tier-2 preorder segments, for $1 < u \leq l_j - s_j + 1$ (set $B_9^j[1] = 1$);
- An array C_9^j for the j^{th} tier-1 preorder segment (for $j = 1, 2, \cdots, p_1$), in which $C_9^j[w]$ is of the form $< q_w, r_w >$, such that q_w and r_w are the τ_2 and τ_3-names of the tier-2 preorder changer in the tier-2 preorder segment that contains the leftmost node of this tier-1 preorder segment at level $(\texttt{bin\_select}_{B_9^j}(1, w) + s_j - 1)$.

Similarly to the analysis in the proof of Lemma 5.7, A_9 occupies $o(n)$ bits. Array E has $O(n/\lg^4 n)$ entries, and each entry occupies $O(\lg\lg n)$ bits, so E occupies $o(n)$ bits. B_1, B_1', C_1 and C_1' occupy $o(n)$ bits by the proof of Lemma 5.7. To analyze the space costs of all the B_9^js and C_9^js, we claim that the total number of 1s in all the B_9^js is $m_1 = O(n/\lg n)$ (we will prove this fact later in this proof). Let d_j be the number of 1s in B_9^j, and f_j be the number of nodes in the j^{th} tier-1 preorder segment. Then the total size of all the B_9^js (constructed using Part (b) of Lemma 2.1) is $\sum_{j=1}^{p_1} \lceil \log_2 \binom{l_j - s_j + 1}{d_i} \rceil \leq \sum_{j=1}^{p_1} \lg \binom{f_j}{d_j} \leq \sum_{j=1}^{p_1} \log_2 \binom{f_j}{d_j} + p_1 \leq \lg \binom{n}{m_1} + p_1 = o(n)$. As there are m_1 elements in all the C_9^js and each of them occupies $O(\lg\lg n)$ bits, we have all the C_9^js occupy $o(n)$ bits. Thus these auxiliary data structures occupy $o(n)$ bits in total.

To compute the leftmost node at the i^{th} level in the tier-2 preorder segment that contains x, by Property 5.5, we can use the fact that the nodes in this tier-2 preorder segment are in the same micro-tree. Let y be the root of this micro-tree. Then the i^{th} level of the tree T is the $(v = i - \texttt{depth}(x))^{\text{th}}$ level of this micro-tree. As the node, z, with the smallest preorder number in this tier-2 preorder segment is a tier-2 preorder changer, we can locate z in constant time using B_1, B_1', C_1 and C_1'. We then use this micro-tree, integers z and v as parameters to retrieve the leftmost node at level v that is in this micro-tree and is after node z in preorder. If this node is in z's tier-2 preorder segment (this can be checked in constant time using B_1'), we return it as the answer. Otherwise, we return ∞.

We then show how to support the computation of the leftmost node at the i^{th} level in the tier-1 preorder segment that contains x. Assume that the tier-1 preorder segment that contains x is the j^{th} tier-1 preorder segment. Since we already know how to find the leftmost node at a given level for any given tier-2 preorder segment, it suffices to locate the tier-2 preorder changer in the tier-2 preorder segment that contains this node. This can be computed in constant time using B_9^j and C_9^j, as the τ_2 and τ_3-names of this tier-2 preorder changer is stored in $C_9^j[\texttt{bin\_rank}_{B_9^j}(1, i - s_j + 1)]$ (s_j can be computed in constant time using E, B_1, B_1', C_1, C_2').

It only remains to prove that $m_1 = O(n/\lg n)$. It suffices to prove that d_j (recall that it is the number of 1s in B_9^j) is at most twice the number of tier-2 preorder segments in the j^{th} tier-1 preorder segment. Assume that the tier-2 preorder segment A occurs more than once (in the form of the combination of the τ_2 and τ_3 names of the tier-2 preorder changer in it) in C_9^j. Consider the t^{th} occurrence of A in C_9^j for $t > 1$. Assume that the $(t-1)^{\text{th}}$ and the t^{th} occurrences of A correspond to the ${g_1}^{\text{th}}$ and the ${g_2}^{\text{th}}$ levels in the tree (i.e. the leftmost nodes at the ${g_1}^{\text{th}}$ and the ${g_2}^{\text{th}}$ levels of the tree are in A, but the leftmost nodes at the $(g_1 - 1)^{\text{th}}$ and the $(g_2 - 1)^{\text{th}}$ levels of the tree are not). Then there is one or more tier-2 preorder segments in the j^{th} tier-1 preorder segment to the left of A whose nodes are at levels between (but excluding) the ${g_1}^{\text{th}}$ and the ${g_2}^{\text{th}}$ levels of the tree. We map the t^{th} occurrence of A to the rightmost one (or one of the rightmost ones) among these tier-1 preorder segments. This way we can map each occurrence (except the first occurrence) of a tier-2 preorder segment in C_9^j to a tier-2 preorder segment in the same tier-1 preorder segment. This mapping is an injective function. Thus the total number of occurrences (except the first occurrences) of all the tier-2 preorder segments in C_9^j is at most the number of the tier-2 preorder segments in the j^{th} tier-1 preorder segment.

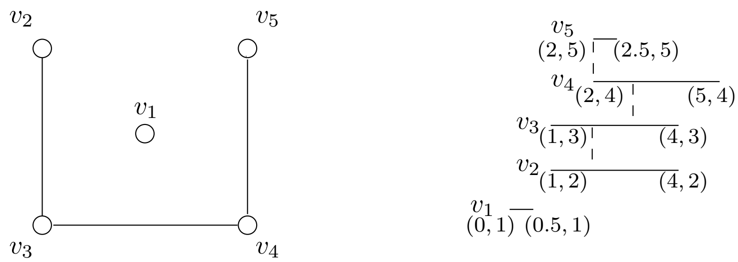


Figure 5.6: The tier-1 level successor graph of the tree in Figure 5.3 and its weak visibility representation.

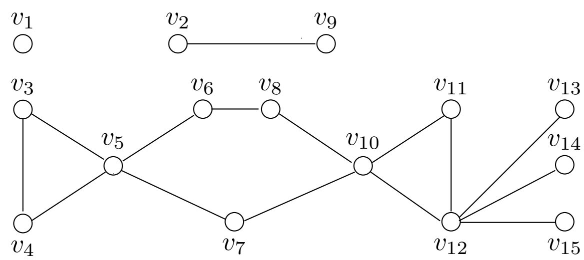


Figure 5.7: The tier-2 level successor graph of the tree in Figure 5.3.

Therefore, d_j is at most twice the number of these tier-2 preorder segments. ☐

We now define the notion of *level successor graphs.*

Definition 5.7. *The* **tier-1** *(or* **tier-2***)* **level successor graph** $G = \{V, E\}$ *is an undirected graph in which the* i^{th} *vertex,* v_i*, corresponds to the* i^{th} *tier-1 (or tier-2) preorder segment, and the edge* $(v_i, v_j) \in E$ *iff there exist nodes* x *and* y *in the* i^{th} *and* j^{th} *tier-1 (or tier-2) preorder segments, respectively, such that either* x *is* y*'s level successor, or* y *is* x*'s level successor.*

See Figure 5.6 for the tier-1 level successor graph of the tree in Figure 5.3. Figure 5.7 gives the tier-2 level successor graph of the same tree. We have the following lemma about level successor graphs.

Lemma 5.21. *A tier-1 (or tier-2) level successor graph is a planar graph.*

Proof. To prove the planarity of the tier-1 successor graph $G = \{V, E\}$, it suffices to construct a weak visibility representation for it (see Section 5.3.4). This is based on a similar idea in Section 2 of [74].

Let s_i and l_i be the minimum and maximum depths of the nodes in the i^{th} tier-1 preorder segment, respectively. We represent the vertex v_i of the tier-1 successor graph (recall that it corresponds to the i^{th} tier-1 preorder segment) using a vertex segment whose endpoints are (s_i, i) and (l_i, i) (if $l_i = s_i$, we increase l_i by a small constant less than 1, such as 0.5). For an edge $(v_i, v_j) \in E$, where $i < j$, we locate a node, x, in the i^{th} tier-1 preorder segment whose level successor is in the j^{th} tier-1 preorder segment. Let $v = \texttt{depth}(x)$. We then represent the edge (v_i, v_j) using an edge segment whose endpoints are (v, i) and (v, j). See Figure 5.6 for an example.

We need prove that none of the edge segments cross any vertex segment that does not contain its endpoints. Assume, contrary to what we are going to prove, that the edge segment for the edge (v_a, v_b), where $a < b$, crosses the vertex segment for the vertex v_c. Let the endpoints of this edge segment to be (k, a) and (k, b). Then we have $a < c < b$. Let y be the node at level k in the a^{th} tier-1 preorder segment whose level successor, z, is in the b^{th} tier-1 preorder segment. As the above edge segment crosses the vertex segment for v_c, then there exists at least one node in the c^{th} tier-1 preorder segment whose depth is k. Let u be one of such nodes. Then $y < u < z$, which is a contradiction.

The planarity of the tier-2 successor graph can be proved using the same approach. □

With these results, we now support `level_succ` and `level_pred`.

Lemma 5.22. *Using $o(n)$ additional bits, operations* `level_succ` *and* `level_pred` *can be supported in $O(1)$ time on* `TC`.

Proof. We only show how to support `level_succ`; `level_pred` can be supported similarly.

We construct the following auxiliary data structures (p_1 and p_2 denote the total number of tier-1 and tier-2 preorder segments, respectively):

- All the auxiliary data structures constructed in the proof of Lemma 5.20;
- An array E' of length p_2, in which $E'[i]$ stores the τ_3-name of the node with the smallest depth in the i^{th} tier-2 preorder segment;
- A table A'_9 that stores for each possible micro-tree and each node in it (identified by its τ_3-name), the τ_3-name of its level successor in the micro-tree if it exists, or 0 otherwise;

- A bit vector K_i for the i^{th} tier-1 preorder segment (for $i = 1, 2, \cdots, p_1$), in which $K_i[j] = 1$ iff the level successors of the rightmost nodes in the i^{th} tier-1 preorder segment at level $(s_i + j - 2)$ and at level $(s_i + j - 1)$ (s_i and l_i denote the minimum and maximum depths of the nodes in the i^{th} tier-1 preorder segment, respectively) are in two different tier-1 preorder segments, for $1 < j \leq l_i - s_i + 1$ (set $K_i[1] = 1$);

- An array G_i for the i^{th} tier-1 preorder segment (for $i = 1, 2, \cdots, p_1$), in which $G_i[j]$ stores the preorder number of the tier-1 preorder changer in the tier-1 preorder segment that contains the level successor of the rightmost node of the i^{th} tier-1 preorder segment at level $(\texttt{bin\_select}_{K_i}(1, j) + s_i - 1)$, for $1 < j \leq l_i - s_i + 1$;

- A bit vector K'_i for the i^{th} tier-2 preorder segment (for $i = 1, 2, \cdots, p_2$), in which $K'_i[j] = 1$ iff the level successors of the rightmost nodes in the i^{th} tier-2 preorder segment at level $(s'_i + j - 2)$ and at level $(s'_i + j - 1)$ (s'_i and l'_i denote the minimum and maximum depths of the nodes in the i^{th} tier-2 preorder segment, respectively) are in two different tier-2 preorder segments, for $1 < j \leq l'_i - s'_i + 1$ (set $K'_i[1] = 1$);

- An array G'_i for the i^{th} tier-2 preorder segment (for $i = 1, 2, \cdots, p_2$), in which $G'_i[j]$ is of the form $< q_j, r_j >$, such that q_j and r_j are the τ_2 and τ_3-names of the tier-2 preorder changer in the tier-2 preorder segment that contains the level successor of the rightmost node of the i^{th} tier-2 preorder segment at level $(\texttt{bin\_select}_{K'_i}(1, j) + s_i - 1)$, for $1 < j \leq l'_i - s'_i + 1$.

Array E' takes $O(n/\lg n \times \lg\lg n) = o(n)$ bits. Similarly to the analysis in the proof of Lemma 5.7, table A'_9 occupies $o(n)$ bits. To analyze the space costs of all the K_is, G_is, K'_is and G'_is, we claim that the total number of 1s in all the K_is (or K'_is) is $k_1 = O(n/\lg^4 n)$ (or $k_2 = O(n/\lg n)$). We will prove this fact later in this proof. With this we can prove that all the K_is, G_is, K'_is and G'_is occupy $o(n)$ bits following the same approach used to analyze the space costs of all the B^i_9s and C^i_9s in the proof of Lemma 5.20. We also concatenate all the K'_is and G'_is and construct an $o(n)$-bit auxiliary structure similar to the structure constructed in the proof of Lemma 5.11 so that we can locate each of them in constant time. Therefore, all the auxiliary data structures occupy $o(n)$ bits.

To compute $\texttt{level\_succ}(x)$, we first check whether its level successor is in the same tier-2 preorder segment that x is in (assume that x is in the j^{th} tier-2 preorder segment), and if it is, compute its preorder number. We perform a constant-time lookup on table A'_9 to compute x's level successor, y, in its micro-tree. If y does not exist, of if y is not in the j^{th} tier-2 preorder segment (this can be checked using B'_1 in constant time), then x's level successor is not in the same tier-2 preorder segment. Otherwise, by Lemma 5.18, we can return y as the result.

If x's level successor is not in the j^{th} tier-2 preorder segment, we locate the node with the smallest depth in this tier-2 preorder segment and compute its depth (we denote the result by t). This can be done in constant time as shown in the proof of Lemma 5.20 (we use E' instead of E). We then compute the τ_2 and τ_3-names (denoted by u and v respectively) of the tier-2 preorder changer in the tier-2 preorder segment that contains x's level successor. These are stored in $G'_j[\texttt{bin\_rank}_{K_j}(1, \texttt{depth}(x) - t + 1)]$. This information is sufficient to determine whether x and its level successor are in the same tier-1 preorder segment using the following approach. Check where there exists a node z such that $\tau(z) =< \tau_1(x), u, v >$ using the approach that converts τ-names to preorder numbers [36, 37](see Section 5.3.1). If z does not exist, then x and its level successor are not in the same tier-1 preorder segment. If z exists but it is not a tier-2 preorder changer (i.e. $B'_1[z] = 0$), or if $z \leq x$, then x and its level successor are not in the same tier-1 preorder segment. Otherwise, we use Lemma 5.20 to locate the leftmost node of the tier-2 preorder segment that contains z at level $\texttt{depth}(x)$. If such a node does not exist, or if its appears to the left of x, then x and its level successor are not in the same tier-1 preorder segment. Otherwise, we return this node as the result.

The case when x and its level successor are not in the same tier-1 preorder segment can be handled using a similar approach as described in the above paragraph using the K_is and G_is.

It now remains to prove that $k_1 = O(n/\lg^4 n)$ and $k_2 = O(n/\lg n)$. We only show how to bound k_1; k_2 can be bounded similarly. There are two types of 1s in K_i. A 1 of the first type corresponds to the first occurrence of a tier-1 preorder changer in G_i. The second type consists of the remaining 1s. We first prove that the number of the 1s in all the K_is of the first type is $O(n/\lg^4 n)$. We map the i^{th} tier-1 preorder segment to vertex v_i in the tier-1 level successor graph. Give a 1 of the first type, if it is in K_a and the corresponding item in G_a stores the preorder number of a tier-1 preorder changer in the

b^{th} tier-1 preorder segment, we map it to the edge (v_a, v_b). This mapping is a bijection. Thus the number of 1s of the first type is equal to the number of edges in the tier-1 level successor graph, which is $O(n/\lg^4 n)$ as the graph is planar. To prove that the number of the 1s in all the K_is of the second type is $O(n/\lg^4 n)$, we show how to map each of them to a distinct edge segment in the weak visibility representation of the tier-1 level successor graph. Consider the two 1s in K_i that correspond to the $(w-1)^{\text{th}}$ and the w^{th} occurrences of a tier-1 preorder changer in G_i (assume that this tier-1 preorder changer is in the j^{th} tier-1 preorder segment), where $w > 1$. Assume that these two 1s correspond to the g^{th} and the l^{th} level of T. Then there exists at least one vertex segment such that the x-coordinates of its endpoints are between but excluding g and l, and they are between the vertex segments that correspond to v_i and v_j. We map the 1 in K_i that corresponds to the w^{th} occurrences of the above tier-1 preorder changer in G_i to the lowest one among these vertex segments. It is clear that each 1 is mapped to a distinct vertex segment this way. This completes the proof. □

With the new operations supported in this section, we can now present the main result of this chapter.

Theorem 5.1. *An ordinal tree of n nodes can be represented using $2n+o(n)$ bits to support all the operations listed in Section 5.1 in $O(1)$ time.*

5.5 Computing a Subsequence of BP and DFUDS

We now consider the efficient computation of any $O(\lg n)$ consecutive bits of the BP or DFUDS sequence of a given tree represented by TC (Theorem 5.2). This result shows that any operation to be supported by BP or DFUDS in the future, can be supported by TC efficiently.

5.5.1 $O(\lg n)$-bit Subsequences of BP in $O(f(n))$ Time with $n/f(n)+o(n)$ Extra Bits

Lemma 5.23. *For any $f(n)$ such that $f(n) = O(\lg n)$ and $f(n) = \Omega(1)$, using $n/f(n) + O(n \lg\lg n/\lg n)$ additional bits, any $O(\lg n)$-bit subsequence of BP can be computed from TC in $O(f(n))$ time.*

Proof. We use $\texttt{BP}[1..2n]$ to denote the $\texttt{BP}$ sequence. Recall that each opening parenthesis in $\texttt{BP}$ corresponds to the preorder number of a node, and each closing parenthesis corresponds to the postorder. Thus the number of opening parentheses corresponding to tier-1 (or tier-2) preorder changers is $O(n/\lg^4 n)$ (or $O(n/\lg n)$), and we call them *tier-1* (or *tier-2*) *marked opening parentheses*.

We first show how to compute the subsequence of $\texttt{BP}$ starting from a tier-2 marked opening parenthesis up to (but not including) the next tier-2 marked opening parenthesis. We use j and k to denote the positions of these two parentheses in $\texttt{BP}$, respectively, and thus our goal is to compute $\texttt{BP}[j..k-1]$. We construct the following auxiliary data structures:

- A bit vector B_{10} of length $2n$, whose i^{th} bit is 1 iff $\texttt{BP}[i]$ corresponds to a tier-1 marked opening parenthesis;
- A bit vector B'_{10} of length $2n$, whose i^{th} bit is 1 iff $\texttt{BP}[i]$ corresponds to a tier-2 marked opening parenthesis;
- An array C_{10} of length m_1 (m_1 denotes the number of tier-1 marked opening parentheses), where $C_{10}[i]$ stores the τ_1-name of the node corresponding to the i^{th} tier-1 marked opening parenthesis;
- An array C'_{10} of length m_2 (m_2 denotes the number of tier-2 marked opening parentheses), where $C'_{10}[i]$ is of the form $< q_i, r_i >$, where q_i and r_i are the τ_2 and τ_3-names of the node corresponding to the i^{th} tier-2 marked opening parenthesis, respectively;
- A table A_{10}, in which for each possible micro-tree and each one of its nodes, we store the subsequence of the balanced parentheses sequence of the micro-tree that starts from the opening parenthesis corresponding to this node to the end of this sequence, and we also store the length of such a subsequence.

As $m_1 = O(n/\lg^4 n)$ and $m_2 = O(n/\lg n)$, the bit vectors B_{10} and B'_{10} can be stored in $O(n \lg\lg n/\lg^4 n)$ and $O(n \lg\lg n/\lg n)$ bits, respectively, using Part (b) of Lemma 2.1. C_{10} and C'_{10} occupy $O(n/\lg^3 n)$ and $O(n \lg\lg n/\lg n)$ bits, respectively. As the length of the balanced parentheses of each micro-tree is at most $6M'$, similarly to the analysis in the proof of Lemma 5.7, we have that the space used by A_{10} is $O(2^{6M'} \times M' \times M') = O(n^{1/4} \lg^2 n)$ bits. Therefore, these auxiliary data structures occupy $O(n \lg\lg n/\lg n)$ bits in total.

To compute $\texttt{BP}[j..k-1]$, we first compute, in constant time, the τ-names of the tier-2 preorder changers, x and y, whose opening parenthesis are stored in $\texttt{BP}[j]$ and $\texttt{BP}[k]$, respectively, using B_{10}, B'_{10}, C_{10} and C'_{10}. The algorithm is similar to the one used in the proof of lemma 5.16. We then perform a constant-time lookup on A_{10} to retrieve the subsequence of the balanced parenthesis sequence of x's micro-tree, starting from the opening parenthesis corresponding to x, to the end of this sequence. Let $P[1..l]$ denote the result. As there is no edge leaving x's micro-tree between and including nodes x and $y-1$, $P[1..k-j]$ is the result if $l \geq k-j$. Otherwise, x must correspond to the last marked tier-2 opening parenthesis of its micro-tree, so there are $k-j-l$ closing parentheses between the subsequence $P[1..l]$ and the tier-2 marked opening parenthesis that corresponds to y. Thus, $\mathtt{BP}[j..k-1]$ can either be computed in constant time if its length is at most $\lg n$, or any $\lg n$-bit subsequence of it can be computed in constant time.

To compute any $O(\lg n)$-bit subsequence of BP, we conceptually divide BP into blocks of size $\lg n$. As any $O(\lg n)$-bit subsequence spans a constant number of blocks, it suffices to support the computation of a block. For a given block with u tier-2 marked opening parentheses, we can run the algorithm described in the last paragraph at most $u+1$ times to retrieve the result. To facilitate this process, we choose a function $f(n)$ where $f(n) = O(\lg n)$ and $f(n) = \Omega(1)$. We explicitly store the blocks that have $2f(n)$ or more tier-2 marked opening parentheses, which takes at most $2n/(\lg n \times 2f(n)) \times \lg n = n/f(n)$ bits. We concatenate the blocks that are explicitly stored, and in order to retrieve any of these blocks in constant time, we construct a bit vector L of length $n/\lg n$, where $L[i] = 1$ iff the i^{th} block is stored explicitly. L occupies $n/\lg n + o(n/\lg n)$ using Part (a) of Lemma 2.1. If the i^{th} block is explicitly stored, its starting position in the concatenated sequence constructed above is $\lg n(\mathtt{bin\_rank}_L(1,i) - 1) + 1$. Thus, a block explicitly stored can be computed in $O(1)$ time, and a block that is not can be computed in $O(f(n))$ time as it has less than $2f(n)$ tier-2 marked opening parentheses. The total space cost of all the auxiliary data structures now becomes $n/f(n) + O(n \lg\lg n/\lg n)$ bits. □

5.5.2 $O(\lg n)$-bit Subsequences of DFUDS in $O(f(n))$ Time with $n/f(n) + o(n)$ Extra Bits

To support the computation of a word of the DFUDS sequence, recall that the DFUDS sequence can be considered as the concatenation of the DFUDS subsequences of all the nodes in preorder (See Section 5.2.1). Thus the techniques used in the proof of Lemma 5.23 can be modified to support the computation of a subsequence of `DFUDS`. We first prove the following lemma.

Lemma 5.24. *Consider a node x that is not the root of any micro-tree. Then its* `DFUDS` *subsequence in the extended micro-tree that contains it as an original node is the same as its* `DFUDS` *subsequence in T.*

Proof. Recall that the `DFUDS` subsequence of a node of degree d consists of d opening parenthesis following by a closing parentheses (See Section 5.2.1). Thus it suffices to prove that the degrees of x in T and in the extended micro-tree that contains it as an original node are the same. This is true by Lemma 5.1. □

We now show how to support the computation of a subsequence of `DFUDS`.

Lemma 5.25. *For any $f(n)$ such that $f(n) = O(\lg n)$ and $f(n) = \Omega(1)$, using $n/f(n) + O(n \lg\lg n/ \lg n)$ additional bits, any $O(\lg n)$-bit subsequence of* `DFUDS` *can be computed from* `TC` *in $O(f(n))$ time.*

Proof. We use $U[1..2n]$ to denote the `DFUDS` sequence. We define the *tier-1* (or *tier-2*) *marked positions* of U to be the starting positions of the `DFUDS` subsequences of the tier-1 (or tier-2) preorder changers in U. Thus the number of tier-1 (or tier-2) marked positions is $O(n/\lg^4 n)$ (or $O(n/\lg n)$).

We first show how to compute the subsequence of U starting from a tier-2 marked position up to (but not including) the next tier-2 marked position in U. We use j and k to denote these two positions in U, respectively, and thus our goal is to compute $U[j..k-1]$. We construct the following auxiliary data structures:

- A bit vector B_{11} of length $2n$, whose i^{th} bit is 1 iff $U[i]$ corresponds to a tier-1 marked position;
- A bit vector B'_{11} of length $2n$, whose i^{th} bit is 1 iff $U[i]$ corresponds to a tier-2 marked position;
- An array C_{11} of length m_1 (m_1 denotes the number of tier-1 marked positions), where $C_{11}[i]$ stores the τ_1-name of the node corresponding to the i^{th} tier-1 marked positions;

- An array C'_{11} of length m_2 (m_2 denotes the number of tier-2 marked positions), where $C'_{11}[i]$ is of the form $< q_i, r_i >$, where q_i and r_i are the τ_2 and τ_3-names of the node corresponding to the i^{th} tier-2 marked positions, respectively;
- A table A_{11}, in which for each possible type 1 extended micro-tree and each one of its nodes, we store the subsequence of the DFUDS sequence of the extended micro-tree that starts from the starting position of the DFUDS subsequence of this node in this extended micro-tree, to the end of this sequence, and we also store the length of such a subsequence.

As $m_1 = O(n/\lg^4 n)$ and $m_2 = O(n/\lg n)$, the bit vectors B_{11} and B'_{11} can be stored in $O(n \lg\lg n/\lg^4 n)$ and $O(n \lg\lg n/\lg n)$ bits, respectively, using Part (b) of Lemma 2.1. C_{11} and C'_{11} occupy $O(n/\lg^3 n)$ and $O(n \lg\lg n/\lg n)$ bits, respectively. As the size of a type 1 extended micro-tree is at most $\frac{1}{4}\lg n$, the length of the DFUDS sequence of each type 1 extended micro-tree is at most $\frac{1}{2}\lg n$. Similarly to the analysis in the proof of Lemma 5.7, we have that the space used by A_{11} is $O(2^{\frac{1}{2}\lg n} \times \lg n \times \lg n) = O(n^{1/2}\lg^2 n)$ bits. Therefore, these auxiliary data structures occupy $O(n \lg\lg n/\lg n)$ bits in total.

To compute $U[j..k-1]$, we first compute, in constant time, the τ-names of the tier-2 preorder changers, x and y, whose DFUDS subsequences start at positions j and k in U, respectively, using B_{11}, B'_{11}, C_{11} and C'_{11}. We then compute the subsequence V of the DFUDS sequence of x's extended micro-tree (x is an original node in it) that starts from the starting position of x's DFUDS subsequence in this extended micro-tree, to the end of this sequence. If x is in a type 1 extended micro-tree, we can retrieve V in constant time using A_{11}. If not, then the DFUDS representation of this extended micro-tree (it is now a type 2 extended micro-tree) is explicitly stored (see Section 5.3.1). We can compute the starting position of V in the DFUDS sequence of x's extended micro-tree in constant time using the algorithm that converts the preorder number of a node to the starting position of its DFUDS subsequence [56] (note that this is what exactly the operation `node_select`$_{\texttt{PRE}}$ does on DFUDS representations; see Section 5.2.1). Therefore, we can either compute V in constant time if its length is at most $\lg n$, or compute any $\lg n$-bit subsequence of it in constant time. Note that the nodes that are represented by $U[j..k-1]$ is in the same micro-tree. Thus if x is not a root of any micro-tree, then by Lemma 5.24, we have $U[j..k-1] = V[1..k-j]$. If x is the root of its extended micro-tree, then we compute its degree d, and replace its representation in V by d opening parentheses followed by a closing parenthesis. We use V' to denote this modified version of V. As all the nodes in V' are represented by their `DFUDS` subsequences in T, we have $U[j..k-1] = V'[1..k-j]$. Therefore, $U[j..k-1]$ can either be computed in constant time if its length is at most $\lg n$, or any $\lg n$-bit subsequence of it can be computed in constant time.

To compute any $O(\lg n)$-bit subsequence of U, we conceptually divide U into blocks of size $\lg n$. Without the loss of generality, we assume that the starting position of the subsequence is greater than 1 (as $U[1] =' (')$). As any $O(\lg n)$-bit subsequence spans a constant number of blocks, it suffices to support the computation of a block. For a given block with u tier-2 marked positions, we can run the algorithm described in the last paragraph at most $u + 1$ times to retrieve the result. We use the exact same approach presented in the proof of Lemma 5.23 to speed up this process, and the result of this lemma follows. □

Combining Lemma 5.23 and Lemma 5.25, we have:

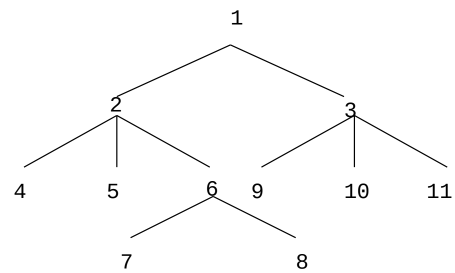
Theorem 5.2. *Given a tree represented by* `TC`*, any $O(\lg n)$ consecutive bits of its* `BP` *or* `DFUDS` *sequence can be computed in $O(f(n))$ time, using at most $n/f(n) + O(n \lg\lg n/\lg n)$ extra bits, for any $f(n)$ where $f(n) = O(\lg n)$ and $f(n) = \Omega(1)$.*

5.6 Multi-Labeled Trees

5.6.1 Definitions

We now consider a multi-labeled tree. Recall that n denotes the number of nodes in the tree, $[\sigma]$ denotes the label alphabet, and t denotes the total number of node-label pairs. As with binary relations, we adopt the assumption that each node is associated with at least one label. To design succinct indexes for multi-labeled trees, we define the interface of the ADT of a multi-labeled tree through the following operator: `node_label`(x, i), which returns the i^{th} label associated with node x in lexicographic order.

We store the tree structure as part of the index (as it takes negligible space), and hence do not assume the support for any navigational operation in the ADT. Recall that we refer to nodes by their preorder numbers (i.e. node x is the x^{th} node in the preorder traversal).



DFUDS: `((()((()))(()))((())))`

Figure 5.8: An ordinal tree (where each node is assigned its rank in `DFUDS` order) and its `DFUDS` representation [9].

The navigational operations we need support on ordinal trees to construct succinct indexes for multi-labeled trees include `child`, `child_rank`, `depth`, `level_anc`, `nbdesc`, `degree`, `LCA`, $\texttt{node\_rank}_{\texttt{DFUDS}}$ and $\texttt{node\_select}_{\texttt{DFUDS}}$. To support these operations, we have two options. One option is to use Theorem 5.1 to encode an ordinal tree using $2n+o(n)$ bits. The other option, which was the original approached that we used in [6], is to augment the `DFUDS` representation [10, 9, 56] to support all these operations. We prove the following lemma.

Lemma 5.26. *Using the* `DFUDS` *representation [10, 9, 56], an ordinal tree with n nodes can be encoded in $2n+o(n)$ bits to support* `child`*,* `child_rank`*,* `depth`*,* `level_anc`*,* `nbdesc`*,* `degree`*,* `LCA`*,* $\texttt{node\_rank}_{\texttt{DFUDS}}$ *and* $\texttt{node\_select}_{\texttt{DFUDS}}$ *in $O(1)$ time.*

Proof. As it is shown in [10, 9, 56] how to support all the navigational operations listed in the lemma except $\texttt{node\_rank}_{\texttt{DFUDS}}$ and $\texttt{node\_select}_{\texttt{DFUDS}}$, we only need provide support for these two operations. We use the operations supported by Lemma 5.4, as it is used to encode the `DFUDS` sequence[10, 9, 56].

In the balanced parentheses representation of the `DFUDS` sequence of the tree [9], each node corresponds to an opening parenthesis and a closing parenthesis. We observe that in the sequence, the opening parentheses correspond to `DFUDS` order, while the closing parentheses correspond to the preorder. For example, in Figure 5.8, the 6^{th} node in `DFUDS` order (which is the 5th node in preorder) corresponds to the 6^{th} opening parenthesis, and the 5^{th} closing parenthesis.

With this observation, $\texttt{node\_select}_{\texttt{DFUDS}}(x)$ means that for the node x (recall that it corresponds to the x^{th} closing parenthesis), we need to compute the rank of the corresponding opening parenthesis among all the opening parentheses. To compute this value, we consider the subsequence of the DFUDS representation of the tree that represents a node and all its descendants. In this subsequence, the number of closing parentheses minus the number of opening parentheses is equal to 1. Therefore, if x is the r^{th} child of its parent, then the closing parenthesis that comes before the DFUDS subsequence of node x matches the opening parenthesis that is r positions before the closing parenthesis in the DFUDS subsequence of x's parent. To make use of this fact, we first find the opening parenthesis that matches the closing parenthesis that comes before the DFUDS subsequence of node x. Its position in the sequence is: $j = \texttt{find\_open}(\texttt{select\_close}(x-1))$. With j, we can compute the starting position of the subsequence of the parent of x, which is $p = \texttt{select\_close}(\texttt{rank\_close}(j))+1$, and $\texttt{child\_rank}(x)$ (denoted by r as above), which is $r = \texttt{select\_close}(\texttt{rank\_close}(p)+1) - j$. Finally, $\texttt{rank\_open}(p+r-1)$ is the result.

The computation of $\texttt{node\_rank}_{\texttt{DFUDS}}(r)$ is exactly the inverse of the above process. □

We now define permuted binary relations and present a related lemma that we use to design succinct indexes for multi-labeled trees.

Definition 5.8. *Given a permutation π on $[n]$ and a binary relation $R \subset [n] \times [\sigma]$, the* **permuted binary relation** *$\pi(R)$ is the relation such that $(x, \alpha) \in \pi(R)$ if and only if $(\pi^{-1}(x), \alpha) \in R$.*

Lemma 5.27. *Consider a permutation π on $[n]$, such that the access to $\pi(i)$ and $\pi^{-1}(i)$ is supported in $O(1)$ time. Given a binary relation $R \subset [n] \times [\sigma]$ of cardinality t, and support for* `object_access` *on R in $f(n, \sigma, t)$ time, there is a succinct index using $t \cdot o(\lg \sigma)$ bits that supports on both R and $\pi(R)$:*

- `label_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t) + \lg\lg\sigma))$ time;*
- `label_select` *for any label $\alpha \in [\sigma]$ in $O(\lg\lg\lg\sigma(f(n,\sigma,t) + \lg\lg\sigma))$ time;*
- `label_pred` *and* `label_succ` *for any character $\alpha \in [\sigma]$ in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t) + \lg\lg\sigma))$ time, and these two operations for any literal $\alpha \in [\bar{\sigma}]$ in $O(f(n,\sigma,t) + \lg\lg\sigma)$ time;*

- `object_rank` *and* `label_access` *for any literal* $\alpha \in [\sigma] \cup [\bar{\sigma}]$ *in* $O(\lg\lg\lg\sigma f(n,\sigma,t) + \lg\lg\sigma)$ *time;*

- `label_nb` *for any literal* $\alpha \in [\sigma] \cup [\bar{\sigma}]$ *and* `object_nb` *in* $O(1)$ *time.*

Proof. As $\texttt{object\_access}_{\pi(R)}(x,r) = \texttt{object\_access}_R(\pi^{-1}(x),r)$, we can support the operator `object_access` on $\pi(R)$ in $f(n,\sigma,t)$ time. Therefore, we can use Theorem 3.3 to construct succinct indexes for R and $\pi(R)$ and use the combined data structure to support the above operations. The total space cost is thus $t \cdot (\lg\sigma)$ bits. ☐

To efficiently find all the α-ancestors of any given node, for each node and for each of its labels α we encode the number of α-ancestors of x. To measure the maximum number of such ancestors, we define the *recursivity* of a node, motivated by the notion of *document recursion level* of a given XML document [87].

Definition 5.9. *The* **recursivity $\boldsymbol{\rho_\alpha}$ of a label** α *in a multi-labeled tree is the maximum number of occurrences of* α *on any rooted path of the tree. The* **average recursivity $\boldsymbol{\rho}$ of a multi-labeled tree** *is the average recursivity of the labels weighted by the number of nodes associated with each label* α *(denoted by* t_α*):* $\rho = \frac{1}{t}\sum_{\alpha\in[\sigma]}(t_\alpha\rho_\alpha)$.

Note that ρ is usually small in practice, especially for XML trees. Zhang *et al.* [87] observed that in practice the document recursion level (when translated to our more precise definition, it is the maximum value of all ρ_αs minus one, which can be easily used to bound ρ) is often very small: in their data sets, it was never larger than 10.

5.6.2 Succinct Indexes

Theorem 5.3. *Consider a multi-labeled tree on* n *nodes and* σ *labels, associated in* t *relations, of average recursivity* ρ*. Given support for* `node_label` *in* $f(n,\sigma,t)$ *time, there is a succinct index using* $t \cdot o(\lg\sigma)$ *bits that supports (for a given node* x*) the enumeration of:*

- *the set of* α*-descendants of* x *(denoted by* D*) in* $O(|D|(\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ *time;*

- *the set of α-children of x (denoted by C) in $O(|C|(\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ time;*
- *the set of α-ancestors of x (denoted by A) in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma)+|A|(\lg\lg\rho_\alpha+\lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma)))$ time using $t(\lg\rho+o(\lg\rho))$ bits of extra space.*

Proof. We encode the underlying ordinal tree structure in $2n+o(n)$ bits using either Theorem 5.1 or Lemma 5.26. The sequence of nodes referred by their preorder (`DFUDS` order) numbers and the associated label sets form a binary relation R_p (R_d). Operations `node_rank`$_{\texttt{DFUDS}}$ and `node_select`$_{\texttt{DFUDS}}$ provide constant-time conversions between the preorder numbers and the `DFUDS` order numbers, and `node_label` supports `object_access` on R_p. By Lemma 5.27, we can construct succinct indexes for R_p and R_d using $t\cdot o(\lg\sigma)$ bits, and support `label_rank`, `label_select` and `label_access` operations on either of them efficiently.

Using the technique of Barbay *et al.* [5, Corollary 1], the succinct index for R_p enables us to enumerate all the descendants of node x matching label α in $O(|D|(\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ time (we can alternatively use the succinct index for R_d to achieve the same result). More precisely, we keep using `label_succ` to retrieve the nodes after but not including x that are associated with α, till we reach a node whose preorder number is greater than or equal to $(x+\texttt{nbdesc}(x))$. Similarly, the succinct index of R_d enables us to enumerate all the children of node x matching α in $O(|C|(\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ time, as the `DFUDS` order traversal lists the children of any given node consecutively.

As there is no order in which the ancestors of each node are consecutive, we store for each label α of a node x the number of ancestors of x (including x) matching α. To be specific, for each label α such that $\rho_\alpha>1$, we represent those numbers in one string $S_\alpha\in[\rho_\alpha]^{t_\alpha}$ (see Definition 5.9 for the definitions of ρ_α and t_α), where the i^{th} number of S_α corresponds to the i^{th} node labeled α in preorder. The lengths of the strings S_αs are implicitly encoded in R_p. We also encode for each label α its recursivity ρ_α in unary, using at most $t+\sigma+o(t+\sigma)$ bits. We use the encoding of Golynski *et al.* [41, Theorem 2.2](see Section 3.2.1) to encode each string S_α in $t_\alpha(\lg\rho_\alpha+o(\lg\rho_\alpha))$ bits to support `string_rank` and `string_access` in $O(\lg\lg\rho_\alpha)$ time and `string_select` in constant time. The total space used by these strings is $\sum_{\alpha\in[\sigma]}t_\alpha(\lg\rho_\alpha+o(\lg\rho_\alpha))$ bits. By concavity of the logarithmic function, this is at most $\left(\sum_{\alpha\in[\sigma]} t_\alpha\right)\left(\lg\left(\frac{\sum_{\alpha\in[\sigma]} t_\alpha\rho_\alpha}{\sum_{\alpha\in[\sigma]} t_\alpha}\right)+o\left(\frac{\sum_{\alpha\in[\sigma]} t_\alpha\rho_\alpha}{\sum_{\alpha\in[\sigma]} t_\alpha}\right)\right)=t(\lg\rho+o(\lg\rho))$.

To support the enumeration of all the α-ancestors of a node x, we first find from R_p the number, p_x, of nodes labeled α preceding x in preorder using `label_rank`. Then we iterate i from 1. In each iteration, we first find the position p_i in S_α of the character i immediately preceding position p_x: it corresponds to the ${p_i}^{\text{th}}$ node labeled α in preorder (this can be located using `label_select` on R_p). If this node is an ancestor of x (this can be checked using `depth` and `level_anc` in constant time), output it, increment i and iterate, otherwise stop. Each iteration consists of a `label_select` operation on R_p and some rank and select operations on S_α, so each is performed in $O(\lg\lg\rho_\alpha+\lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma))$ time. Hence it takes $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma)+|A|(\lg\lg\rho_\alpha+\lg\lg\lg\sigma(f(n,\sigma,t)+\lg\lg\sigma)))$ time to enumerate A. □

We can also support the retrieval of the first α-descendant, child or ancestor of node x that appears after node y in preorder.

Corollary 5.3. *The structure of Theorem 5.3 also supports (for any two given nodes x and y) the selection of:*

- *the first α-descendant of x after y in preorder in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ time;*
- *the first α-child of x after y in preorder in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ time;*
- *the first α-ancestor of x after y in preorder in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma)+\lg\lg\rho_\alpha)$ time.*

Proof. Using the index in Theorem 5.3, we can easily support the first operation; we merely need retrieve the first node labeled α after y using `label_succ` and then check whether it is a descendant of x. The support for the second operation is nontrivial only when y is a descendant of x. (otherwise, the result is either the first α-child of x or ∞). In this case, we first locate the child of x, node u, that is also an ancestor of y using `depth` and `level_anc`. Then the problem is reduced to the selection of the first α-child of x after u in preorder, which can be computed by performing `label_succ` on R_d.

To support the search for the first α-ancestor of x after y, we assume that y precedes x in preorder (otherwise the operator returns ∞), and that y is an ancestor of x (if not, the problem can be reduced to the search for the first α-ancestor of node x after node $\texttt{LCA}(x, y)$). Using `label_succ` on the relation R_p and some navigational operators, we can find the first α-descendant z of y in preorder in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t) + \lg\lg\sigma))$ time. Node z is not necessarily an ancestor of x, but it has the same number, i, of α-ancestors as the node we are looking for. We can retrieve i from the string S_α in $O(\lg\lg\rho_\alpha)$ time. Finally, the first α-ancestor of x after y is the α-node corresponding to the value i immediately preceding the position corresponding to x in S_α, found in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma)+\lg\lg\rho_\alpha)$ time. □

The operations on multi-labeled trees are important for the support of XPath queries for XML trees [5, 3]. The main idea of our algorithms is to construct indexes for binary relations for different traversal orders of the trees. Note that without succinct indexes, we would encode different binary relations separately and waste a lot of space.

We finally show how to use these succinct indexes to design a succinct integrated encoding of multi-labeled trees.

Corollary 5.4. *Consider a multi-labeled tree on n nodes and σ labels, associated in t relations, of average recursivity ρ. It can be represented using $\lg\binom{n\sigma}{t} + t\cdot o(\lg\sigma)$ bits to support (for a given node x) the enumeration of:*

- *the set of α-descendants of x (denoted by D) in $O(|D|(\lg\lg\lg\sigma)^2\lg\lg\sigma)$ time;*
- *the set of α-children of x (denoted by C) in $O(|C|(\lg\lg\lg\sigma)^2\lg\lg\sigma)$ time;*
- *the set of α-ancestors of x (denoted by A) in $O((\lg\lg\lg\sigma)^2\lg\lg\sigma + |A|(\lg\lg\rho_\alpha + \lg\lg\sigma\lg\lg\lg\sigma))$ time using $t(\lg\rho + o(\lg\rho))$ bits of extra space.*

It also supports (for any two given nodes x and y) the selection of:

- *the first α-descendant of x after y in preorder in $O((\lg\lg\lg\sigma)^2\lg\lg\sigma)$ time;*
- *the first α-child of x after y in preorder in $O((\lg\lg\lg\sigma)^2\lg\lg\sigma)$ time;*
- *the first α-ancestor of x after y in preorder in $O((\lg\lg\lg\sigma)^2\lg\lg\sigma + \lg\lg\rho_\alpha)$ time using $t(\lg\rho + o(\lg\rho))$ bits of extra space.*

Proof. In the proof of Theorem 5.3 and Corollary 5.3, we use Theorem 3.5 to encode the binary relation R_p, and construct a succinct index for R_d using Lemma 5.27. This corollary immediately follows. □

The discussions in Chapter 3 on the more general case of binary relations when each object may be associated with zero or more labels (instead of at least one label) also apply to the more general case for multi-labeled trees when each node may be associated with zero or more labels.

5.7 Discussion

In this chapter, we design a succinct representation of ordinal trees, based on that of Geary *et al.* [36, 37], that supports all the navigational operations supported by various succinct tree representations while requiring only $2n + o(n)$ bits. It also supports efficient level-order traversal, a useful ordering previously supported only with a very limited set of operations [55]. Our second contribution expands on the notion of a single succinct representation supporting more than one traversal ordering, by showing that our method supports two other encoding schemes as abstract data types. In particular, it supports extracting a word ($O(\lg n)$ bits) of the balanced parenthesis sequence [71] or depth first unary degree sequence [10, 9] in $O(f(n))$ time, using at most $n/f(n)+o(n)$ additional bits, for any $f(n)$ in $O(\lg n)$ and $\Omega(1)$. We then further design succinct indexes and integrated encodings of multi-labeled trees to support the efficient retrieval of α-children/descendants/ancestors of a given node.

There are a few open problems. The first open problem is whether we can compute any $O(\lg n)$-bit subsequence of `BP` or `DFUDS` in constant time using $o(n)$ additional bits for `TC`. Our result in Theorem 5.2 is in the form of time/space tradeoff and we do not know whether it is optimal. Other interesting open problems include the support of the operations that are not previously supported by `BP`, `DFUDS` or `TC`. One is to support rank/select operations on the level-order traversal of the tree. It is not supported by previous research except the trivial support for it on Jacobson's representation directly based on the level-order traversal of the trees, in which each node is identified by its level-order number [55]. Another one is to support `level_leftmost` (`level_rightmost`) on an arbitrary subtree of T, which is not supported by previous research, either. Finally, for multi-labeled trees, as it requires $t(\lg \rho + o(\lg \rho))$ bits of extra space to support the efficient retrieval of α-ancestors of a given node, an open problem is to reduce this additional storage cost, or to prove that it is necessary.

Chapter 6

Planar Graphs and Related Classes of Graphs

This chapter deals with the problem of designing succinct representations of unlabeled and multi-labeled graphs. The chapter starts with an introduction in Section 6.1, followed by a brief review of previous work in Section 6.2. We describe existing results that we use and/or improve upon in Section 6.3. We present succinct indexes for triangulated planar graphs with labels associated with their vertices or edges in Section 6.4, and use them to design succinct indexes for multi-labeled planar graphs. To achieve these results, we describe a succinct representation of unlabeled planar triangulations which supports the rank/select of edges in ccw (counter clockwise) order in addition to the other operations supported in previous work. We present a succinct encoding for k-page graphs with labels associated with their edges in Section 6.5. To achieve this result, we design a succinct representation for a k-page graph when k is large which supports various navigational operations more efficiently. We conclude with a discussion of our results in Section 6.6.

6.1 Introduction

Graphs are fundamental combinatorial objects in mathematics and in computer science. They are widely used to represent various types of data, such as the link structure of the web, geographic maps, and surface meshes in computer graphics. As modern applications often process large graphs, the problem of designing space-efficient data structures to represent graphs has attracted a great deal of attention. In particular the idea of succinct data structures has been applied to various classes of graphs [55, 70, 18, 17, 12, 15, 16].

Previous work focused on succinct graph representations which support efficiently testing the adjacency between two vertices and listing the edges incident to a vertex [70, 15, 16]. However, in many applications, such connectivity information is associated with labels on the edges or vertices of the graph, and the space required to encode those labels dominates the space used to encode the connectivity information, even when the encoding of the labels is compressed [54]. For example, when surface meshes are associated with properties such as color and texture information, more bits per vertex are required to encode those labels than to encode the graph itself. We address this problem by designing succinct representations of *labeled graphs*, where labels from alphabet $[\sigma]$ are associated with edges or vertices. These representations efficiently support label-based connectivity queries, such as retrieving the neighbors associated with a given label. We assume that all the graphs are simple graphs.

We investigate three important classes of graphs: planar triangulations, planar graphs and k-page graphs. Planar graphs, and in particular planar triangulations, correspond to the connectivity information underlying surface meshes. Triangle meshes are one of the most fundamental representations for geometric objects: in computational geometry they are one natural way to represent surface models, and in computer graphics triangles are the basic geometric primitive for efficient rendering. k-page graphs have applications in several areas, such as sorting with parallel stacks [83], fault-tolerant processor arrays [77] and VLSI (very large scale integration) design [20].

6.2 Previous Work

Here we briefly review related work on succinct unlabeled graphs. As most graphs in practice have particular combinatorial properties, researchers usually exploit these properties to design succinct representations.

Jacobson [55] was the first to propose a succinct representation of planar graphs. His approach is based on the concept of *book embedding* by Bernhart and Kainen [11]. A *k-page* *embedding* is a topological embedding of a graph with the vertices along the spine and edges distributed across k pages, each of which is an outerplanar graph. The minimum number of pages, k, for a particular graph has been called the *pagenumber* or *book thickness*. Jacobson showed how to represent a k-page graph using $O(kn)$ bits to support adjacency tests in $O(\lg n)$ bit probes, and listing the neighbors of a vertex x in $O(d(x)\lg n + k)$ bit probes, where $d(x)$ is the degree of x.

Munro and Raman [70, 71] improved his results under the word RAM model by showing how to represent a graph using $2kn + 2m + o(kn + m)$ bits to support adjacency tests and the computation of the degree of a vertex in $O(k)$ time, and the listing of all the neighbors of a given vertex in $O(d + k)$ time. Gavoille and Hanusse [35] proposed a different tradeoff. They proposed an encoding in $2(m+i)\lg k + 4(m+i) + o(km)$ bits, where i is the number of isolated vertices, to support the adjacency test in $O(k)$ time. As any planar graph can be embedded in at most 4 pages [86], these results can be applied directly to planar graphs. In particular, a planar graph can be represented using $8n + 2m + o(n)$ bits to support adjacency tests and the computation of the degree of a vertex in $O(1)$ time, and the listing of all the neighbors of a given vertex x in $O(d(x))$ time [70, 71].

A different line of research based on the canonical ordering of planar graphs was taken by Chuang *et al.* [18, 19]. They designed a succinct representation of planar graphs of n vertices and m edges in $2m + (5+\epsilon)n + o(m+n)$ bits, for any constant $\epsilon > 0$, to support the operations on planar graphs in asymptotically the same amount of time as the approach described in the previous paragraph. Chiang *et al.* [17] further reduced the space cost to $2m + 3n + o(m + n)$ bits. When a planar graph is triangulated, Chuang *et al.* [18, 19] showed how to represent it using $2m + 2n + o(m + n)$ bits.

Based on a partition algorithm, Castelli Aleardi *et al.* [15] proposed a succinct representation of planar triangulations with a boundary. Their data structure uses 2.175 bits per triangle to support various operations efficiently. Castelli Aleardi *et al.* [16] further extended this approach to design succinct representations of 3-connected planar graphs and triangulations using 2 bits per edge and 1.62 bits per triangle respectively, which asymptotically match the respective entropy of these two types of graphs.

Finally, Blandford *et al.* [12] considered the problem of representing graphs with small separators (the graph separator considered in their main result is a *vertex separator*, i.e.

a set of vertices whose removal separates the graph into two approximately equally sized parts). This is useful because many graphs in practice, including planar graphs [61], have small separators. They designed a succinct representation using $O(n)$ bits that supports adjacency tests and the computation of the degree of a vertex in $O(1)$ time, and the listing of all the neighbors of a given vertex x in $O(d(x))$ time.

6.3 Preliminaries

6.3.1 Multiple Parentheses

Chuang *et al.* [18, 19] proposed the succinct representation of *multiple parentheses*, a string of $O(1)$ types of parentheses that may be unbalanced. Thus a multiple parenthesis sequence of p types of parentheses is a sequence over the alphabet $\{'(_1', ')_1', '(_2', ')_2', \cdots, '(_p', ')_p'\}$. We call $'(_i'$ and $')_i'$ *type-i opening parenthesis* and *type-i closing parenthesis*, respectively. The operations considered are:

- $\texttt{m\_rank}_S(i, \alpha)$: the number of parentheses α in $S[1..i]$;
- $\texttt{m\_select}_S(i, \alpha)$: the position of the i^{th} parenthesis α;
- $\texttt{m\_first}_S(\alpha, i)$ ($\texttt{m\_last}_S(\alpha, i)$): the position of the first (last) parenthesis α after (before) $S[i]$;
- $\texttt{m\_match}_S(i)$: the position of the parenthesis matching $S[i]$;
- $\texttt{m\_enclose}_S(k, i_1, i_2)$: the position of the closest matching parenthesis pair of type k which encloses $S[i_1]$ and $S[i_2]$.

We omit the subscript S when it is clear from the context.

Chuang *et al.* [18, 19] showed how to to support the above operations

Lemma 6.1 ([18, 19]). *Consider a string S of $O(1)$ types of parentheses that is stored explicitly. Then there is an auxiliary data structure using $o(|S|)$ bits that supports the operations listed above in $O(1)$ time.*

We show how to improve this result in Lemma 6.8, and propose an encoding for the case when the number of types of parentheses is non-constant in Theorem 6.7.

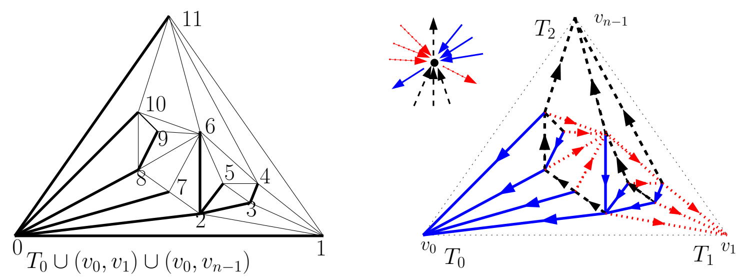


Figure 6.1: A triangulated planar graph of 12 vertices with its canonical spanning tree $\overline{T}_0$ (on the left). On the right, it shows the triangulation induced with a realizer, as well as the local condition.

6.3.2 Realizers and Planar Triangulations

Our general approach, in much of this chapter, is based on the idea of realizers of planar triangulations (see Figure 6.1 for an example).

Definition 6.1 ([81]). *A* **realizer** *of a planar triangulation $\mathcal{T}$ is a partition of the set of the internal edges into three sets T_0, T_1 and T_2 of directed edges, such that for each internal vertex v the following conditions hold:*

- *v has exactly one outgoing edge in each of the three sets T_0, T_1 and T_2;*
- **local condition:** *the edges incident to v in counterclockwise (ccw) order are: one outgoing edge in T_0, zero or more incoming edges in T_2, one outgoing edge in T_1, zero or more incoming edges in T_0, one outgoing edge in T_2, and finally zero or more incoming edges in T_1.*

A fundamental property of realizers (characterizing very finely the planarity of planar triangulations) that we use extensively in Section 6.4 is:

Lemma 6.2 ([81]). *Consider a planar triangulation $\mathcal{T}$ of n vertices, with exterior face (v_0, v_1, v_{n-1}). Then $\mathcal{T}$ always admits a realizer $R = (T_0, T_1, T_2)$ and each set of edges in T_i is a spanning tree of all internal vertices. More precisely:*

- *T_0 is a spanning tree of $\mathcal{T} \setminus \{v_1, v_{n-1}\}$;*
- *T_1 is a spanning tree of $\mathcal{T} \setminus \{v_0, v_{n-1}\}$;*

- *T_2 is a spanning tree of $\mathcal{T} \setminus \{v_0, v_1\}$.*

As we consider undirected planar triangulations, we orient each internal edge when we compute the realizers. For each edge in T_i, if we reverse its direction, we get a different set of directed edges. We use T_i^{-1} to denote this set. We also use the following lemma in this chapter.

Lemma 6.3 ([81])**.** *If T_0, T_1 and T_2 are realizers of a planar triangulation $\mathcal{T}$, then for $i \in \{0, 1, 2\}$, there is no directed cycle in the set $T_i \cup T_{i+1}^{-1} \cup T_{i+2}^{-1}$ (indices are modulo 3).*

6.4 Planar Triangulations

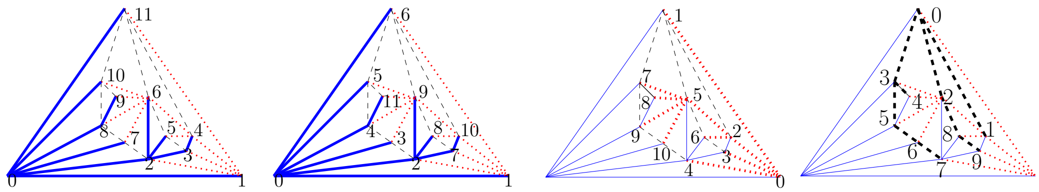
6.4.1 Three New Traversal Orders on a Planar Triangulation

A key notion in the development of our results is that of three new traversal orders of planar triangulations based on realizers. Let $\mathcal{T}$ be a planar triangulation of n vertices and m edges, with exterior face (v_0, v_1, v_{n-1}). We denote its realizer by (T_0, T_1, T_2) following Definition 6.1. By Lemma 6.2, T_0, T_1 and T_2 are three spanning trees of the internal nodes of $\mathcal{T}$, rooted at v_0, v_1 and v_{n-1}, respectively. We add the edges (v_0, v_1) and (v_0, v_{n-1}) to T_0, and call the resulting tree, $\overline{T}_0$, the *canonical spanning tree* of $\mathcal{T}$ [18, 19]. In this section, we denote each vertex by its number in *canonical ordering*, which is the ccw preorder number in $\overline{T}_0$ (i.e. vertex i or v_i denotes the i^{th} vertex in canonical ordering). We use (x, y) to denote the edge between two vertices x and y.

Definition 6.2. *The* **zeroth order***, π_0, is defined on all the vertices of $\mathcal{T}$ and is simply given by the preorder traversal of $\overline{T_0}$ starting at v_0 in* counterclockwise order *(ccw order).*

The **first order***, π_1, is defined on the vertices of $\mathcal{T} \setminus v_0$ and corresponds to a traversal of the edges of T_1 as follows. Perform a preorder traversal of the contour of $\overline{T_0}$ in a ccw manner. During this traversal, when visiting a vertex v, we enumerate consecutively its incident edges $(v, u_1), \ldots, (v, u_i)$ in T_1, where v appears before u_i in π_0. The traversal of the edges of T_1 naturally induces an order on the nodes of T_1: each node (different from v_1) is uniquely associated with its parent edge in T_1.*

The **second order***, π_2, is defined on the vertices of $\mathcal{T} \setminus \{v_0, v_1\}$ and can be computed in a similar manner by performing a preorder traversal of T_0 in* clockwise order *(cw order).*



π_0 $\quad DFUDS \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 1 & 2 & 7 & 8 & 10 & 11 & 3 & 5 & 6 & 4 & 9 \end{bmatrix}$ $\quad \pi_1 \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 11 & 4 & 3 & 2 & 6 & 5 & 10 & 9 & 8 & 7 \end{bmatrix}$ $\quad \pi_2 \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 4 & 6 & 10 & 9 & 8 & 7 & 2 & 5 & 3 \end{bmatrix}$

Figure 6.2: A planar triangulation induced with one realizer. The three orders π_0, π_1 and π_2, as well as the order induced by a `DFUDS` traversal of $\overline{T_0}$ are also shown.

When visiting in cw order the contour of $\overline{T_0}$, the edges in T_2 incident to a node v are listed consecutively to induce an order on the vertices of T_2.

Note that the orders π_1 and π_2 do not correspond to previously studied traversal orders on the trees T_1 and T_2, as they are dependent on $\overline{T_0}$ through π_0 (see Figure 6.2). To show that all the internal nodes are listed in π_2 and π_3, it suffices to prove the following lemma.

Lemma 6.4. *Consider an edge (v_i, v_j) in T_1 (or T_2). If $i < j$ (or $i > j$), then v_i is v_j's parent in T_1 (or T_2).*

Proof. We only consider the case when the edge (v_i, v_j) is in T_1; the claim for the case when (v_i, v_j) is in T_2 can be proved similarly.

As the case when $i = 1$ is trivial, and there is no edge in T_1 that is incident to v_0 or v_{n-1}, we only need consider the case when v_i and v_j are internal vertices.

We first prove that v_i is not v_j's ancestor in T_0. Assume, contrary to what we are going to prove, that v_i is v_j's ancestor in T_0. Recall that the edges in T_0, T_1 and T_2 are oriented toward the parent nodes incident to them. Then there is a directed path from v_j to v_i in T_0. As there is an edge in T_1 between v_i and v_j, there is a directed cycle from v_j to v_i and then back to v_j using edges from the set $T_0 \cup T_1^{-1}$ or the set $T_1^{-1} \cup T_0^{-1}$ (depending on the direction of the edge (v_i, v_j)), which contradicts Lemma 6.3.

Let v_k be the lowest common ancestor of v_i and v_j in T_0. As $i < j$ and v_i is not v_j's ancestor in T_0, the path from v_i to v_k in T_0, the path from v_j to v_k in T_0 and the edge (v_i, v_j) define a closed region R (see Figure 6.3). Assume, contrary to what we are going to

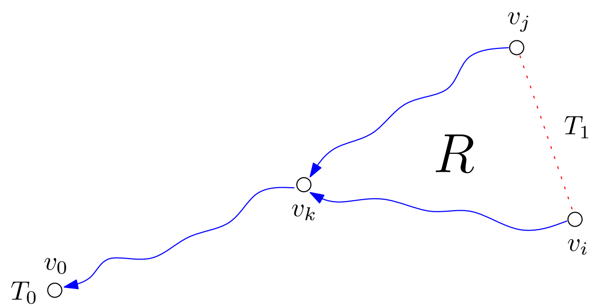


Figure 6.3: Region R in the proofs of Lemma 6.4 and Lemma 6.6.

prove, that v_j is v_i's parent in T_1. Then, according to the local condition in Definition 6.1, the parent of v_j in T_2 in either inside R or on the boundary of R. As there is a directed path from v_j to v_{n-1} and v_{n-1} is neither inside R or on its boundary, we conclude that there exists a node v_t either in the path from v_i to v_k in T_0, or in the path from v_j to v_k, that is v_j's ancestor. In the former case, there is a directed cycle $v_i, \cdots, v_t, \cdots, v_j, v_i$ consisting of edges in the set $T_0 \cup T_1^{-1} \cup T_2^{-1}$. In the latter case, there is a directed cycle $v_j, \cdots, v_t, \cdots, v_j$ consisting of edges in the set $T_0 \cup T_2^{-1}$. Either of these two observations contradicts Lemma 6.3. $\square$

The following lemma is crucial, as it puts in correspondence the labels of the neighbors of a vertex with a finite number of substrings.

Lemma 6.5. *For any node x, its children in T_1 (or T_2), listed in ccw order (or cw order), have consecutive numbers in π_1 (or π_2). In the case of $\overline{T_0}$, the children of x are listed consecutively by a* `DFUDS` *traversal of $\overline{T_0}$.*

Proof. This lemma directly follows the local condition and the ccw traversal we perform on T_0 to construct π_1 and π_2. To be specific, the edges between x and its children in T_1 are all incoming edges incident to x in T_1, and because of the local condition in Definition 6.1, they are encountered consecutively when listing the edges incident to x in ccw order (and just before visiting the outgoing edge of x in T_0). A similar argument holds for π_2 and π_0. $\square$

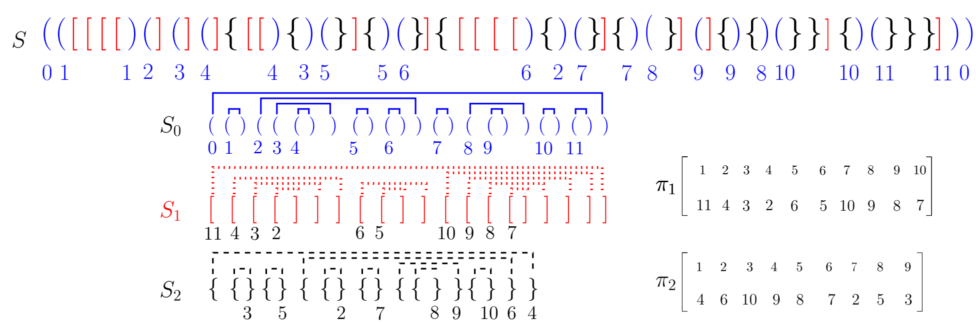


Figure 6.4: The multiple parenthesis string encoding of the planar triangulation in Figure 6.2.

6.4.2 Representing Planar Triangulations

We consider the following operations on unlabeled planar triangulations:

- `adjacency`(x, y), whether vertices x and y are adjacent;
- `degree`(x), the degree of vertex x;
- `select_neighbor_ccw`(x, y, r), the r^{th} neighbor of vertex x starting from vertex y in ccw order if x and y are adjacent, and ∞ otherwise;
- `rank_neighbor_ccw`(x, y, z), the number of neighbors of vertex x between (and including) the vertices y and z in ccw order if y and z are both neighbors of x, and ∞ otherwise;
- $\Pi_j(i)$, given the number of a vertex in π_0, returns the number of the same vertex in π_j (for $j \in \{1, 2\}$);
- $\Pi_j^{-1}(i)$, given the number of a vertex in π_j, returns the number of the same vertex in π_0 (for $j \in \{1, 2\}$).

To represent a planar triangulation $\mathcal{T}$, we compute a realizer (T_0, T_1, T_2) of $\mathcal{T}$ following Lemma 6.2. We then encode the three trees T_0, T_1 and T_2 using a multiple parenthesis sequence S of length $2m$ consisting of three types of parenthesis. S is obtained by performing a preorder traversal of the canonical spanning tree $\overline{T}_0 = T_0 \cup (v_0, v_1) \cup (v_0, v_{n-1})$

and using different types of parentheses to describe the edges of $\overline{T}_0$, T_1 and T_2. We use parentheses of the first type, namely $'('$ and $')'$, to encode the tree $\overline{T}_0$, and other types of parentheses, $'['$, $']'$, $'\{'$, $'\}'$, to encode the edges of T_1 and T_2. We use S_0, S_1 and S_2 to denote the subsequences of S that contain all the first, second, and the third types of parentheses, respectively. We construct S as follows (see Figure 6.4 for an example).

Let $v_0, \ldots, v_{n-1}$ be the ccw preorder of the vertices of $\overline{T}_0$. Then the string S_0 is simply the balanced parenthesis encoding of the tree $\overline{T}_0$ [70, 71] (see Section 5.2.1): S_0 can be obtained by performing a ccw preorder traversal of the contour of $\overline{T}_0$, writing down an opening parenthesis when an edge of $\overline{T}_0$ is traversed for the first time, and a closing parenthesis when it is visited for the second time. During the traversal of $\overline{T}_0$, we insert in S a pair of parentheses $'['$ and $']'$ for each edge of T_1, and a pair of parentheses $'\{'$ and $'\}'$ for each edge in T_2. More precisely, when visiting in ccw order the edges incident to a vertex v_i, we insert:

- A $'['$ for each edge (v_i, v_j) in T_1, where $i < j$, before the parenthesis $')'$ corresponding to v_i;
- A $']'$ for each edge (v_i, v_j) in T_1, where $i < j$, after the parenthesis $'('$ corresponding to v_j;
- A $'\}'$ for each edge (v_i, v_j) in T_2, where $i > j$, after the parenthesis $'('$ corresponding to v_i;
- A $'\{'$ for each edge (v_i, v_j) in T_2, where $i > j$, before the parenthesis $')'$ corresponding to v_j.

The relative order of the parentheses $'['$, $']'$, $'\}'$ and $'\{'$ inserted between two consecutive parentheses of the other type, i.e. $'('$ or $')'$, does not matter. For the simplicity of the proofs in this section, we assume that when we insert a parenthesis pair for an edge in T_1 (or T_2), we always ensure that the positions of this pair in S either enclose or are enclosed in those for an edge that shares the same parent node in the same tree. Thus the string S is of length $2m$, consisting of three types of parenthesis. It is easy to observe that the subsequences S_1 and S_2 are balanced parenthesis sequences of lengths $2(n-1)$ and $2(n-2)$, respectively.

We first observe some basic properties of the string S. Recall that a node v_i can be referred to by its preorder number in $\overline{T}_0$, and by the position of the matching parenthesis pair of the first type corresponding to it (let p_i and q_i respectively denote the positions of the opening and closing parentheses of this pair in S). Let be p_f (or q_l) be the position of the opening (or closing) parenthesis in S corresponding to the first (or last) child of node v_i in $\overline{T}_0$.

Property 6.1. *The following basic facts hold:*

- *Two nodes v_i and v_j are adjacent if and only if there is one common incident edge (v_i, v_j) in exactly one of the trees $\overline{T}_0$, T_1 or T_2;*
- $p_i < p_f < q_l < q_i$;
- *The number of edges incident to v_i and not belonging to the tree T_0 is $(p_f - p_i - 1) + (q_i - q_l - 1)$;*
- *If v_i is not a leaf in $\overline{T_0}$, between the occurrences of the $'('$ that correspond to the vertices v_i and v_{i+1} (note that the $'('$ corresponding to v_{i+1} is at position p_f), there is exactly one $']'$. Similarly, there is exactly one $'\{'$ between the $')'$ that correspond to the vertices v_i and the $')'$ at position q_l.*

We now prove following lemma that is important to our representation.

Lemma 6.6. *In the process of constructing S, the two parentheses of the second type (or the third type) inserted for each edge in T_1 (or T_2) form a matching parenthesis pair in S.*

Proof. We only prove the lemma for parentheses of the second type; the claim for parentheses of the third type can be proved similarly.

Recall that for each edge (v_i, v_j) in T_1, where $i < j$, we insert a $'['$ and a $']'$ into S. We first prove that between these two parentheses, the position of parenthesis $'['$ in S is before that of parenthesis $']'$. As the case when $i = 1$ is trivial, and no edge in T_1 is incident to v_0 or v_{n-1}, we only need consider the case when v_i and v_j are internal vertices. By the process of constructing S, it suffices to prove that the parenthesis $')'$ corresponding to v_i appears before the parenthesis $'('$ corresponding to v_j. Assume, contrary to what we are going to prove, that this is not true. As the parenthesis $'('$ corresponding to v_i appears before the parenthesis $'('$ corresponding to v_j (this is because $i < j$), we conclude that the parenthesis pair corresponding to v_i in S_0 encloses the pair corresponding to v_j. Thus v_i is an ancestor of v_j in T_0. Therefore, there is directed path from v_j to v_i using edges from the set T_0. By Lemma 6.4, v_i is v_j's parent in T_1, so the edge (v_i, v_j) is oriented toward v_i. Hence there is a directed cycle $v_j, \cdots, v_i, v_j$ using edges from the set $T_0 \cup T_1^{-1}$, which contradicts Lemma 6.3.

We now only need prove that the two parenthesis pairs of the second type inserted for two different edges in T_1 either do not intersect, or one is enclosed in the other. Let (v_i, v_j) $(i < j)$ and (v_p, v_q) $(p < q)$ be two edges that are not incident to the same node in T_1 (the case when these two edges are incident to the same node in T_1 is trivial). As shown in the proof of Lemma 6.4, we have that v_i is not v_j's ancestor in T_0, and that v_p is not v_q's ancestor in T_0. Let v_k be the lowest common ancestor of v_i and v_j. We define the region R as in the proof of Lemma 6.4 (see Figure 6.3). Without the loss of generality, we assume that $p < i$. There are two cases.

We first consider the case when v_p is v_i's ancestor. If v_p is also v_j's ancestor, then the parenthesis pair of the first type corresponding to v_p encloses the pairs corresponding to v_i and v_j. By the process we use to construct S, the two parenthesis pairs of the second type inserted for edges (v_i, v_j) and (v_p, v_q) do not intersect. If v_p is not v_j's ancestor, then it is in the path from v_i to v_k in T_0, excluding v_i and v_k. We observe that the parenthesis $'['$ inserted for (v_p, v_q) is after that inserted for (v_i, v_j). By the local condition in Definition 6.1 and the planarity of the graph, we also have the vertex v_q is either inside region R, or is in the path from v_j to v_k in T_0. Therefore, $q < j$. Thus the parenthesis $'('$ corresponding to v_q is before the parenthesis $'('$ corresponding to v_j. Hence the parenthesis $']'$ inserted for (v_p, v_q) is before that inserted for (v_i, v_j). Therefore, the parenthesis pair of the second type inserted for (v_p, v_q) is enclosed in that inserted for (v_i, v_j).

We then consider the case when v_p is not v_i's ancestor. In this case, the parenthesis $')'$ corresponding to v_p appears before that corresponding to v_i. Thus the parenthesis $'['$ inserted for (v_p, v_q) appears before that inserted for (v_i, v_j). As v_p is outside R, by the planarity of the graph, v_q is either outside R, or on R's boundary. We also observe that v_q cannot be in the path from v_j to v_k in T_0, because otherwise, by the local condition in

Definition 6.1, v_p is either in R or in the path from v_i to v_k in T_0. Therefore, v_q is either before v_i in canonical order, or is a descendant of v_i, or is after v_j in canonical order. In the first two cases, the parenthesis pairs of the second type inserted for (v_p, v_q) and (v_i, v_j) do not intersect. In the last case, the parenthesis pair of the second type inserted for (v_i, v_j) is enclosed by that for (v_p, v_q). $\square$

Observe that S_0 is the balanced parenthesis encoding of the tree $\overline{T_0}$ [70, 71], so that if we store S_0 and construct the auxiliary data structures for S_0 as in [70, 71, 17, 74, 62], we can support a set of navigational operators on $\overline{T_0}$ (see Section 5.2.1). S can be represented using Lemma 6.1 in $2m \lg 6 + o(m) = 2m\lceil \log_2 6 \rceil + o(m) = 6m + o(m)$ bits. However, this encoding does not support the computation of an arbitrary word in S_0, so that we cannot navigate in the tree $\overline{T_0}$ without storing S_0 explicitly, which will cost essentially 2 additional bits per node. To reduce this space redundancy, and to decrease the item $2m\lceil \log_2 6 \rceil$ to $2m \log_2 6 + o(m)$, we have the following lemma.

Lemma 6.7. *The string S can be stored in $2m \log_2 6 + o(m)$ bits to support the operators listed in Section 6.3.1 in $O(1)$ time, as well as the computation of an arbitrary word, or $\Theta(\lg(n))$ bits of the balanced parenthesis sequence of $\overline{T_0}$ in $O(1)$ time.*

Proof. We construct a conceptual bit vector B_1 of $2m$ bits, so that $B_1[i] = 1$ iff $S[i] =' ('$ or $S[i] =')'$. We construct another conceptual bit vector B_2 of $2m - 2n$ bits for the 0s in B_2 (recall that there are $2n$ parentheses of the first type), so that $B_2[i] = 1$ iff the parenthesis corresponds to the i^{th} 0 in B_1 is either $'['$ or $']'$. We store B_1 and B_2 using Part (b) of Lemma 2.1 to support rank/select operations on them. The space cost of storing B_1 and B_2 is thus $\lg \binom{2m}{2n} + o(m) + \lg \binom{2m-2n}{n} + o(m)$. To analyze the above space cost, we use the equality $\log_2 n! = n \log_2 n - n \log_2 e + \frac{1}{2} \log_2 n + O(1)$ as in Section 4.6.4. We have (note that $m = 3n - 3$):

$$\begin{aligned} & \log_2 \binom{2m}{2n} + \log_2 \binom{2m-2n}{2n-2} \\ = \; & \log_2 \binom{6n-6}{2n} + \log_2 \binom{4n-6}{2n-2} \\ = \; & \log_2 \Big(\frac{(6n-6)!}{(2n)!(4n-6)!} \times \frac{(4n-6)!}{(2n-2)!(2n-4)!} \Big) \end{aligned}$$

$$
\begin{aligned}
&= \log_2 \frac{(6n-6)!}{(2n)!(2n-2)!(2n-4)!} \\
&< \log_2 \frac{(6n)!}{((2n)!)^3} \\
&= \log_2(6n)! - 3\log_2(2n)! \\
&= 6n\log_2(6n) - 6n\log_2 e + \frac{1}{2}\log_2(6n) - 3(2n\log_2(2n) - 2n\log_2 e + \frac{1}{2}\log_2(2n)) + O(1) \\
&= 6n\log_2 3 - \log_2 n + O(1) \\
&< 6n\log_2 3 + O(1) \\
&= 2m\log_2 3 + O(1)
\end{aligned}
$$

Therefore, the two bit vectors B_1 and B_2 occupy $2m\log_2 3 + o(m)$ bits.

In addition, we store S_0, S_1 and S_2 using Lemma 5.4. The space cost of storing these three sequences is $2n + o(n) + 2(n-1) + o(n) + 2(n-2) + o(n) = 2m + o(m)$ bits. Thus the total space cost is $2m\log_2 6 + o(m)$ bits.

B_1 and B_2 can be used to compute the rank/select operations over S if we treat each type of (opening and closing) parentheses as the same character. For example, to compute the number of parentheses of the third type in $S[1..i]$, we can first compute the number of 0s in $S[1..i]$ (j denotes the result). Then we have the number of parentheses of the third type in $S[1..i]$ is $\texttt{bin\_rank}_{B_2}(0, j)$. Other rank/select operations can be supported similarly. On the other hand, S_0, S_1 and S_2 can be used to support operations on the parentheses of the same type. By representing all these data structures, the operations listed in Section 6.3.1 can be easily supported in constant time. As we store S_0 explicitly in our representation, we can trivially support the computation of an arbitrary word of S_0. □

The same approach can be directly applied to a sequence of $O(1)$ types of parentheses that may be unbalanced.

Lemma 6.8. *Consider a multiple parenthesis sequence M of n parenthesis of p types, where $p = O(1)$. M can be stored using $n\log(2p) + o(n)$ bits to support the operators listed in Section 6.3.1 in $O(1)$ time, as well as the computation of an arbitrary word, or $\Theta(\lg(n))$ bits of the balanced parenthesis sequence of the parentheses of a given type in M in $O(1)$ time.*

Proof. Let n_i be the number of parenthesis of type i in M. Let $l_i = \sum_{j=i}^{p} n_j$. Thus $l_1 = n$ and $l_p = n_p$. For $i = 1, 2, \cdots, p-1$, we construct a bit vector $B_i[1..l_i]$, where $B_i[k] = 1$ iff the k^{th} parenthesis among the parentheses of types $i, i+1, \cdots, p$ in M is of type i. We store all the B_is using Part (b) of Lemma 2.1. Thus the space cost of all the B_is is $\sum_{i=1}^{p-1}[\lg \binom{l_i}{n_i} + o(l_i)] < \sum_{i=1}^{p-1}[\log_2 \binom{l_i}{n_i} + 1 + o(n)] = \sum_{i=1}^{p-1} \log_2 \binom{l_i}{n_i} + o(n)$. To analyze the above space cost, we again use the equality $\log_2 n! = n\log_2 n - n\log_2 e + \frac{1}{2}\log_2 n + O(1)$ as in Section 4.6.4 (Let $H_0^*(M)$ be the zeroth order entry of M when we replace each occurrence of the parentheses of the same type by one distinct character). We have:

$$
\begin{aligned}
& \sum_{i=1}^{p} \log_2 \binom{l_i}{n_i} \\
= \ & \log_2 \prod_{i=1}^{p} \binom{l_i}{n_i} \\
= \ & \log_2 \prod_{i=1}^{p} \frac{l_i!}{n_i!(l_i - n_i)!} \\
= \ & \log_2 \prod_{i=1}^{p} \frac{l_i!}{n_i! l_{i+1}!} \\
= \ & \log_2 \left(\frac{l_1!}{n_1! l_2!} \times \frac{l_2!}{n_2! l_3!} \times \cdots \times \frac{l_{p-1}!}{n_{p-1}! l_p!}\right) \\
= \ & \log_2 \frac{n!}{n_1! \times n_2! \times \cdots \times n_p!} \\
= \ & \log_2 n! - \sum_{i=1}^{p} \log_2(n_i!) \\
= \ & n\log_2 n - n\log_2 e + \frac{1}{2}\log_2 n - \sum_{i=1}^{p}(n_i \log_2 n_i - n_i \log_2 e + \frac{1}{2}\log_2 n_i) + O(1) \\
= \ & n\log_2 n - n\log_2 e + \frac{1}{2}\log_2 n - [\sum_{i=1}^{p}(n_i \log_2 n_i) - n\log_2 e + \sum_{i=1}^{p}\frac{1}{2}\log_2 n_i] + O(1) \\
= \ & n\log_2 n - \sum_{i=1}^{p}(n_i \log_2 n_i) + O(\log_2 n)
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{i=1}^{p}(n_i \log_2 \frac{n}{n_i}) + O(\log_2 n) \\
&= nH_0^*(M) + O(\log_2 n) \\
&\leq n \log_2 p + O(\log_2 n)
\end{aligned}
$$

Thus the space cost of all the B_is is $n \log_2 p + o(n)$ bits.

Let M_i be the subsequence of M that contains all the parentheses of type i. Note that M_i may be unbalanced. We use the approach of Chuang *et al.* [18, 19] to encode all the M_is while supporting all the operations on balanced parentheses listed in Section 5.3.5 on them. More precisely, let u_i be the difference between the number of opening and closing parentheses in M_i. We can either insert u_i opening parentheses before the beginning of M_i or append u_i closing parentheses to the end of M_i to make it a balanced parenthesis sequence (M'_i denotes such a sequence and n'_i denotes its length). We have $n'_i \leq 2n_i$. To encode M'_i while allowing the computation of any $O(\lg n)$-bit substring of $M'_i[j]$ in constant time, we only need store M_i and u_i which occupies $n_i + \lg n$ bits. We also build the auxiliary data structures for M'_i using Lemma 5.4. Thus it takes $n_i + \lg n + o(n'_i) = n_i + o(n_i)$ bits to encode M_i while supporting all the operations on balanced parentheses listed in Section 5.3.5 on M_i. Hence the space cost of all the M_is is $n + o(n)$ bits. Therefore, the total space cost of all the data structures is $n \log_2(2p) + o(n)$ bits.

As we can perform rank/select operations for each type of parentheses in M using B_is, and we can support all the operations on balanced parentheses listed in Section 5.3.5 on M_is, the algorithms used in the proof of Lemma 6.7 can be used to support in $O(1)$ time the operators listed in Section 6.3.1 on M. An arbitrary word of the parenthesis sequence of type i in M can be computed using M_i. □

The following theorem shows how to support the navigational operations on triangulations. While the space used here is a little more than that of Chiang *et al.* [17] (see Section 6.2), the explicit use of the three parenthesis sequences seems crucial to exploiting the realizers to support $\Pi_j(i)$ and $\Pi_j^{-1}(i)$ efficiently (for $j \in \{1, 2\}$).

Theorem 6.1. *A planar triangulation $\mathcal{T}$ of n vertices and m edges can be represented using $2m \log_2 6 + o(m)$ bits to support operators* `adjacency`, `degree`, `select_neighbor_ccw`, `rank_neighbor_ccw` *as well as the* $\Pi_j(i)$ *and* $\Pi_j^{-1}(i)$ *(for* $j \in \{1, 2\}$*) in* $O(1)$ *time.*

Proof. We construct the string S for $\mathcal{T}$ as shown in this section, and store it using $2m \log_2 6 + o(m)$ bits by Lemma 6.7. Recall that S_0 is the balanced parenthesis encoding of $\overline{T_0}$, and that we can compute an arbitrary word of S_0 from S. Thus we can construct additional auxiliary structures using $o(n) = o(m)$ bits [70, 71, 17, 74, 62] to support the navigational operations on $\overline{T_0}$ (see Section 5.2.1). As each vertex is denoted by its number in canonical ordering, vertex x corresponds to the x^{th} opening parenthesis in S_0. We now show that these data structures are sufficient to support the navigational operations on $\mathcal{T}$.

To compute `adjacency`(x, y), recall that x and y are adjacent iff one is the parent of the other in one of the trees $\overline{T_0}$, T_1 and T_2. As S_0 encodes the balanced parenthesis sequence of $\overline{T_0}$, we can trivially check whether x (or y) is the parent of y (or x) using existing algorithms on S_0 [70, 71] (see Section 5.2.1). To test adjacency in T_1, we recall that x is the parent of y iff the (only) outgoing edge of y, denoted by a $'{]}'$, is an incoming edge of x, denoted by a $'{[}'$. It then suffices to retrieve the first $'{]}'$ after the y^{th} $'('$ in S, given by `m_first`$('{[}', \texttt{m\_select}(y, '('))$, and compute the index, i, of its matching closing parenthesis, $'{[}'$, in S. We then check whether the nearest succeeding closing parenthesis $')'$ of the $'{[}'$ retrieved, located using `m_first`$(')', i)$, matches the x^{th} opening parenthesis $'('$ in S. If it does, then x is the parent of y in T_1. We use a similar approach to test the adjacency in T_2.

To compute `degree`(x), let d_0, d_1 and d_2 be the degrees of x in the trees $\overline{T_0}$, T_1 and T_2 (in this proof, we denote the degree of a node in a tree as the number of nodes adjacent to it), respectively, so that the sum of these three values is the answer. To compute d_0, we use S_0 and the algorithm to compute the degree of a node in an ordinal tree using its balanced parenthesis representation by Chiang *et al.* [17] (see Section 5.2.1). To compute $d_1 + d_2$, if x has children in $\overline{T_0}$, we first compute the indices, i_1 and i_2, of the x^{th} and the $x + 1^{\text{th}}$ $'('$ in S, and the indices, j_1 and j_2, of the $(n-x)^{\text{th}}$ and the $(n-x+1)^{\text{th}}$ $')'$ in S in constant time. By the third item of Property 6.1, we have the property $d_1 + d_2 = (i_2 - i_1 - 1) + (j_2 - j_1 - 1)$. The case when x is a leaf in $\overline{T_0}$ can be handled similarly.

To support `select_neighbor_ccw` and `rank_neighbor_ccw`, we make use of the local condition of realizers in Definition 6.1. The local condition tells us that, given a vertex x, its neighbors, when listed in ccw order, form the following six types of vertices: x's parent in $\overline{T_0}$, x's children in T_2, x's parent in T_1, x's children in $\overline{T_0}$, x's parent in T_2, and x's children in T_1. The i^{th} child of x in ccw order in $\overline{T_0}$ can be computed in constant time, and the number of siblings before a given child of x in ccw order can also be computed in constant time using the algorithms of Lu and Yeh [62] (see Section 5.2.1). The children of x in T_1 corresponds to the parentheses $'['$ between the $(n-x)^{\text{th}}$ and the $(n-x+1)^{\text{th}}$ $')'$ in S. In addition, by the construction of S, if u and v are both children of x, and u occurs before v in π_1, then u is also before v in ccw order among x's children in T_1. The children of x in T_2 have a similar property. Thus the operators supported on S allow us to perform rank/select on x's children in T_1 and T_2 in ccw order. As we can also compute the number of each type of neighbors of x in constant time, this allows us to support `select_neighbor_ccw` and `rank_neighbor_ccw` in $O(1)$ time.

To compute $\Pi_1(i)$, we first locate the position, j, of the i^{th} occurrence of $'('$ in S, which is $\texttt{m\_select}(i, '(')$. We then locate the position, k, of the first $']'$ after position j, which is $\texttt{m\_first}(']', j)$. After that, we locate the matching parenthesis of $S[j]$ using $\texttt{m\_match}(j)$ (p denotes the result). $S[p]$ is the parenthesis $'['$ that corresponds to the edge between v_i and its parent in T_1, and by the construction algorithm of S, the rank of $S[p]$ is the answer, which is $\texttt{m\_rank}(p, '[')$. The computation of Π_1^{-1} is exactly the inverse of the above process. Π_2 and Π_2^{-1} can be supported similarly. $\square$

6.4.3 Vertex Labeled Planar Triangulations

We now consider a vertex labeled planar triangulation. Let n and m respectively denote the numbers of its vertices and edges, σ denote the number of labels, and t denote the total number of node-label pairs. Same as binary relations, we adopt the assumption that each vertex is associated with at least one label.

In addition to unlabeled operators, we present a set of operators that allow efficient navigation in a vertex labeled planar triangulation (these are natural extensions to navigational operators on multi-labeled trees):

- $\texttt{lab\_degree}(\alpha, x)$, the number of neighbors of vertex x that are labeled α;
- $\texttt{lab\_select\_ccw}(\alpha, x, y, r)$, the r^{th} vertex labeled α among neighbors of vertex x after vertex y in ccw order, if y is a neighbor of x, and ∞ otherwise;

- lab_rank_ccw(α, x, y, z), the number of neighbors of vertex x labeled α between vertices y and z in ccw order if y and z are neighbors of x, and ∞ otherwise.

We define the interface of the ADT of vertex labeled planar triangulations through the operator node_label(v, r), which returns the r^{th} label in lexicographic order associated with vertex v (i.e. the v^{th} vertex in canonical ordering).

Recall that Lemma 6.7 encodes the string S constructed in Section 6.4.2 to support the computation of an arbitrary word of S_0, which is the balanced parenthesis sequence of the tree $\overline{T_0}$. In this section, we consider the DFUDS sequence of $\overline{T_0}$, as the DFUDS order traversal visits the children of a node consecutively. We have the following lemma.

Lemma 6.9. *The string S can be stored in $(2\log_2 6 + \epsilon)m + o(m)$ bits, for any ϵ such that $0 < \epsilon < 1$, to support the operators listed in Section 6.3.1 in $O(1)$ time, as well as the computation of an arbitrary word, or $\Theta(\lg n)$ bits of the balanced parenthesis sequence, and of the* DFUDS *sequence of $\overline{T_0}$ in $O(1)$ time.*

Proof. We construct the same data structures as in Lemma 6.7, except when we encode S_0, we use Theorem 5.2 to represent the tree $\overline{T_0}$ (choose $f(n) = 1/\epsilon$). More precisely, we encode S_0 using $(2 + \epsilon)n + o(n)$ bits, for any ϵ such that $0 < \epsilon < 1$, and this encoding supports the computation of an arbitrary word of the balanced parenthesis sequence, and the DFUDS sequence of $\overline{T_0}$ in constant time. As we can compute an arbitrary word of the original sequence of S_0 in constant time and all the other structures are the same as in Lemma 6.7, we can still support the operators listed in Section 6.3.1 in constant time. □

We now construct succinct indexes for vertex labeled planar triangulations.

Theorem 6.2. *Consider a multi-labeled planar triangulation $\mathcal{T}$ of n vertices, associated with σ labels in t pairs ($t \geq n$). Given the support of* node_label *in $f(n, \sigma, t)$ time on the vertices of $\mathcal{T}$, there is a succinct index using $t \cdot o(\lg \sigma)$ bits which supports* lab_degree, lab_select_ccw *and* lab_rank_ccw *in $O((\lg\lg\lg\sigma)^2(f(n, \sigma, t) + \lg\lg\sigma))$ time.*

Proof. The main idea is to combine our succinct representation of planar triangulations with three instances of the succinct indexes for related binary relations.

We represent the combinatorial structure of $\mathcal{T}$ using Theorem 6.1, in which we use Lemma 6.9 to store S. Thus we can construct the auxiliary data structures for the DFUDS representation of $\overline{T_0}$ using Lemma 5.26. Observe that the sequence of the vertices (for simplicity we only consider internal vertices) referred by their numbers in three different orders, namely the DFUDS order of the nodes of $\overline{T_0}$, π_1 and π_2, form three binary relations, R_0, R_1 and R_2, with their associated labels.

We adopt the same strategy used previously for multi-labeled trees in Section 5.6.2. We can convert between the ranks of the vertices between π_0, π_1 and π_2 in constant time by Theorem 6.1. We can also convert between the preorder numbers of the nodes in $\overline{T_0}$ (note that they are in the the order of π_0) and the DFUDS numbers of the nodes in $\overline{T_0}$ in constant time using node_rank$_{\texttt{DFUDS}}$ and node_select$_{\texttt{DFUDS}}$. Therefore, we can use the operator node_label to support the ADT of R_0, R_1 and R_2. Thus, for each of the binary relations R_1, R_2 and R_0 we construct a succinct index of $t \cdot o(\lg \sigma)$ bits using Theorem 3.3.

To compute lab_degree(α, x), we first check whether x's parents in $\overline{T_0}$, T_1 and T_2 are labeled α. The DFUDS number of x's parent in $\overline{T_0}$ can be computed in constant time. The number in π_1 (or π_2) of x's parent in T_1 (or T_2) can also be computed in constant time, as shown in the proof of Theorem 6.1. Thus this can be checked by performing label_access operation on R_0, R_1 and R_2. We now need compute the numbers of x's children in $\overline{T_0}$, T_1 and T_2 that are associated with label α. By Lemma 6.5, x's children in $\overline{T_0}$, T_1 and T_2 are listed consecutively in DFUDS order of $\overline{T_0}$, π_1 and π_2, respectively. Compute the DFUDS number, s, of x's first child in $\overline{T_0}$ in constant time. Then the DFUDS numbers of x's children in $\overline{T_0}$ are in the range $[s+1, s+\texttt{degree}(x)]$. Thus we can compute the number of x's children in $\overline{T_0}$ that are associated with label α by performing label_rank on R_0. To get the numbers in π_1 of x's children in T_1, we locate the first and last occurrences of parenthesis $'['$ inserted for the edges in T_1 whose parent node is x, and compute their ranks, f and l, among all the occurrences of parenthesis $'['$ in S. As the numbers in π_2 of x's children in T_1 is in the range $[f, l]$, we can compute the number of x's children in T_1 that are associated with label α by performing label_rank on R_1. The number of x's children in T_2 that are associated with label α can be computed similarly.

To support lab_select_ccw and lab_rank_ccw, by the local condition in Definition 6.1 and the algorithms in the above paragraph, it suffices to show that we can support the label-based rank/select of the children of a given node in ccw order in the three trees $\overline{T_0}$, T_1 and T_2, respectively. As we can compute the ranges of the DFUDS numbers in $\overline{T_0}$, the numbers in π_1 and the numbers in π_2 of x's children in $\overline{T_0}$, T_1 and T_2, respectively, these operations can be supported by performing `label_rank` and `label_select` operations on R_0, R_1 and R_2.

Finally, we observe that the space requirement of our representation is dominated by the cost of the succinct indexes for the binary relations, each using $t \cdot o(\lg \sigma)$ bits. $\square$

To design a succinct representation of vertex labeled graphs using the above theorem, we have the following corollary.

Corollary 6.1. *A multi-labeled planar triangulation $\mathcal{T}$ of n vertices, associated with σ labels in t pairs ($t \geq n$) can be represented using $\lg \binom{n\sigma}{t} + t \cdot o(\lg \sigma)$ bits to support* `node_label` *in $O(1)$ time, and* `lab_degree`*,* `lab_select_ccw` *and* `lab_rank_ccw` *in $O((\lg\lg\lg\sigma)^2 \lg\lg\sigma)$ time.*

Proof. We use the approach in the proof of Theorem 3.5 to encode the binary relation between the vertices in canonical order and the set of labels in $n + t + o(n+t) + \lg \binom{n\sigma}{t} + o(t) + O(\lg\lg(n\sigma))$ bits to support `object_select` on it in constant time. Observe that the above operator directly supports `node_label` on $\mathcal{T}$. We then build the succinct indexes of $t \cdot o(\lg \sigma)$ bits for $\mathcal{T}$ using Theorem 6.2 and the corollary directly follows. $\square$

6.4.4 Edge Labeled Planar Triangulations

We now consider an edge labeled planar triangulation. Let n and m respectively denote the numbers of its vertices and edges, σ denote the number of labels, and t denote the total number of edge-label pairs. Same as binary relations, we adopt the assumption that each edge is associated with at least one label. We define the interface of the ADT of edge labeled planar triangulations through the operator `edge_label`(x, y, r), which returns the r^{th} label associated to the edge between the vertices x and y in lexicographic order if they are adjacent, or 0 otherwise.

We consider the following operations:

- `lab_adjacency`(α, x, y), whether there is an edge labeled α between vertices x and y;
- `lab_degree_edge`(α, x), the number of edges incident to vertex x that are labeled α;

- `lab_select_edge_ccw`(α, x, y, r), the r^{th} edge labeled α among edges incident to vertex x after edge (x, y) in ccw order, if y is a neighbor of x, and ∞ otherwise;
- `lab_rank_edge_ccw`(α, x, y, z), the number of edges incident to vertex x labeled α between edges (x, y) and (x, z) in ccw order if y and z are neighbors of x, and ∞ otherwise.

We construct the following succinct index for edge labeled planar triangulations.

Theorem 6.3. *Consider a multi-labeled planar triangulation $\mathcal{T}$ of n vertices and m edges, in which the edges are associated with σ labels in t pairs ($t \geq m$). Given the support of* `edge_label` *in $f(n, \sigma, t)$ time on the edges of $\mathcal{T}$, there is a succinct index using $t \cdot o(\lg \sigma)$ bits which supports* `lab_adjacency` *in $O(\lg \lg \lg \sigma f(n, \sigma, t) + \lg \lg \sigma)$ time, and* `lab_degree_edge`*,* `lab_select_edge_ccw` *and* `lab_rank_edge_ccw` *in $O((\lg \lg \lg \sigma)^2 \cdot (f(n, \sigma, t) + \lg \lg \sigma))$ time.*

Proof. We represent the combinatorial structure of $\mathcal{T}$ using Theorem 6.1, in which we use Lemma 6.9 to store S. We also construct the auxiliary data structures for the `DFUDS` representation of $\overline{T_0}$ using Lemma 5.26.

We number the edges in $\overline{T_0}$, T_1 and T_2 by the numbers of their child nodes in `DFUDS` order of $\overline{T_0}$, π_1 and π_2, respectively, and denote these three orders of edges by π'_0, π'_1 and π'_2, respectively. For example, in Figure 6.2, the 8^{th} edge in π'_0 is the edge between v_5 (i.e. the 8^{th} node in `DFUDS` order of $\overline{T_0}$) and v_2. We observe that the numbers of the edges in π'_0, π'_1 and π'_2 have numbers in $[n-1]$, $[n-2]$ and $[n-3]$, respectively. Thus the edges in π'_0, π'_1 and π'_2 and the label set $[\sigma]$ form three binary relations R'_0, R'_1 and R'_2, respectively. To support `object_select`(x, r) on R'_0, let y be the node whose `DFUDS` number in $\overline{T_0}$ is x. We locate y's parent z, and `edge_label`(y, z, r) is the result. The support for `object_select` on R'_1 and R'_2 is similar. Therefore, we can use `edge_label` to support the ADT of R'_0, R'_1 and R'_2. For each of these three binary relations, we construct a succinct index using $t \cdot o(\lg \sigma)$ bits using Theorem 3.3.

To compute `lab_adjacency`(α, x, y), we first use the algorithm in the proof of Theorem 6.1 to check whether x and y are adjacent, and if they are, which of the three trees ($\overline{T_0}$, T_1 and T_2) has the edge (x, y). If x is y's parent in $\overline{T_0}$, we compute y's `DFUDS` number (i.e. the number of edge (x, y) in π'_0), u, in $\overline{T_0}$, and `label_access`$_{R'_0}(u, \alpha)$ is the answer.

The cases when x is y's child in $\overline{T_0}$, and when the edge (x, y) is in T_1 or T_2 can be handled similarly. Thus, we can support `lab_adjacency` in $O(\lg\lg\lg\sigma f(n,\sigma,t) + \lg\lg\sigma)$ time.

To support the other three operations, we observe that the edges between a given node x and its children in $\overline{T_0}$, T_1 and T_2 have consecutive numbers in π'_0, π'_1 and π'_2, respectively. We also have x's children in $\overline{T_0}$ and T_1 are listed in ccw order in π'_0 and π'_1, respectively, and x's children in T_2 are listed in cw order in π_2. Thus we can use algorithms similar to those in Theorem 6.2 to support these operations.

Finally, we observe that the space requirement of our representation is dominated by the cost of the succinct indexes for the binary relations, each using $t \cdot o(\lg\sigma)$ bits. $\square$

To design a succinct representation of edge labeled graphs using the above theorem, we have the following corollary.

Corollary 6.2. *A multi-labeled planar triangulation $\mathcal{T}$ of n vertices and m edges, in which the edges are associated with σ labels in t pairs ($t \geq m$), can be represented using* $\lg\binom{m\sigma}{t} + t \cdot o(\lg\sigma)$ *bits to support* `edge_label` *in $O(1)$ time,* `lab_adjacency` *in $O(\lg\lg\sigma)$ time, and* `lab_degree_edge`*,* `lab_select_edge_ccw` *and* `lab_rank_edge_ccw` *in $O((\lg\lg\lg\sigma)^2\lg\lg\sigma)$ time.*

Proof. We represent the combinatorial structure of $\mathcal{T}$ using Theorem 6.1, in which we use Lemma 6.9 to store S. Let m_1, m_2 and m_3 denote the number of edges in $\overline{T_0}$, T_1 and T_2, respectively. Let t_1, t_2 and t_3 denote the total numbers of edge-label pairs in $\overline{T_0}$, T_1 and T_2, respectively. We encode the three binary relations R'_0, R'_1 and R'_3 defined in Theorem 6.3 using Theorem 3.5. The space cost of encoding them in bits is $\sum_{i=1}^{2}[\lg\binom{m_i\sigma}{t_i} + t_i \cdot o(\lg\sigma)] = \sum_{i=0}^{2}[\log_2\binom{m_i\sigma}{t_i} + t_i \cdot o(\lg\sigma)] < \lg\binom{m\sigma}{t} + t \cdot o(\lg\sigma)$, which dominates the overall space cost.

To support `edge_label`(x, y, r), we check which of the three trees, $\overline{T_0}$, T_1 and T_2, contains (x, y), and we compute the number of this edge in π'_0, π'_1 or π'_3. We then perform `object_select` on R'_0, R'_1 or R'_2 to compute the result. The other operations can be supported using Theorem 6.3. $\square$

6.4.5 Extensions to Planar Graphs

We now extend the techniques of Sections 6.4.2, 6.4.3 and 6.4.4 to general planar graphs. By Fáry's theorem [27], any planar graph admits a *straight line embedding* (i.e. a drawing of a planar graph such that all its edges are straight line segments that do not cross). Thus it suffices to represent straight-line embedded planar graphs.

Consider a straight-line embedded planar graph G of n vertices and m edges. To use our results on planar triangulations, we construct a planar triangulation $\mathcal{T}$ for G using the following approach (this approach was used by Kirkpatrick [58] to reduce the point location query problem on general planar subdivisions to that on triangular subdivisions). We first surround G with a large triangle such that all the vertices and edges of G are in the interior of this triangle. We add the three vertices of this triangle and the three edges between them into G, and denote the resulting graph G'. Finally, we triangulate each interior face of G' that is a polygon with more than three vertices. The resulting graph is the planar triangulation $\mathcal{T}$.

Let n' and m' be the number of vertices and edges of $\mathcal{T}$, respectively. Then we have $n' = n + 3$ and $m' = 3n + 3$. We denote the three nodes on the exterior face of $\mathcal{T}$ by v_0, v_1 and v_n. We denote the nodes of G by their numbers in the canonical ordering of $\mathcal{T}$. Thus the nodes of G are $v_2, v_3, \cdots, v_n$. The three orders π_0, π_1 and π_2 on the vertices of G are simply given by these three orders on the vertices of $\mathcal{T}$. Recall that we use (T_0, T_1, T_2) to denote the realizer of $\mathcal{T}$, and $\overline{T_0}$ to denote its canonical spanning tree.

We now extend Theorem 6.1 to represent unlabeled planar graphs.

Theorem 6.4. *A straight-line embedded planar graph G of n vertices and m edges can be represented using $3n(\log_2 3 + 3 + \epsilon) + o(n)$ bits to support operators* `adjacency`, `degree`, `select_neighbor_ccw`, `rank_neighbor_ccw` *as well as $\Pi_j(i)$ and $\Pi_j^{-1}(i)$ (for $j \in \{1, 2\}$) in $O(1)$ time.*

Proof. We construct the planar triangulation $\mathcal{T}$ for G using the above approach. We then represent $\mathcal{T}$ using Theorem 6.1, in which we use Lemma 6.9 to encode the string S constructed to encode $\mathcal{T}$. Thus $\mathcal{T}$ is encoded in $m'(2\log_2 6 + \epsilon) + o(m') = 3n(2\log_2 6 + \epsilon) + o(n)$ bits. In addition, we construct the following three bit vectors to indicate which edge in $\mathcal{T}$ is present in G:

- A bit vector $B_0[2..n]$, where $B_0[i] = 1$ iff the edge between the i^{th} node in `DFUDS` order of $\overline{T_0}$ and its parent in $\overline{T_0}$ is present in G;

- A bit vector $B_1[2..n]$, where $B_1[i] = 1$ iff the edge between the i^{th} node in π_1 and its parent in T_1 is present in G;
- A bit vector $B_2[1..n-1]$, where $B_2[i] = 1$ iff the edge between the i^{th} node in π_2 and its parent in T_2 is present in G.

We encode these three bit vectors in $3n + o(n)$ bits using Part (a) of Lemma 2.1. Thus the total space cost is $3n(2\log_2 6 + \epsilon) + 3n + o(n) = 3n(\log_2 3 + 3 + \epsilon) + o(n)$ bits.

To compute `adjacency`(x, y), we first check whether x and y are adjacent in $\mathcal{T}$. If they are not, we return false. If they are, the algorithm in the proof of Theorem 6.1 also tells us which of the three trees, $\overline{T_0}$, T_1 and T_2, has the edge (x, y) of $\mathcal{T}$. If x is y's parent in $\overline{T_0}$, we compute y's `DFUDS` number, j, in $\overline{T_0}$. If $B_0[j] = 1$, then the edge (x, y) is in G, so we return true. We return false otherwise. The case when y is x's parent in $\overline{T_0}$, and the case when the edge (x, y) of $\mathcal{T}$ is in T_1 or T_2 can handled similarly.

To compute `degree`(x), we observe that the algorithm in the above paragraph can be used to check whether x and its parents in $\overline{T_0}$, T_1 and T_2 are adjacent in G. Thus it suffices to compute the number of x's children in $\overline{T_0}$, T_1 and T_2 that are adjacent to x in G. To count the number, u, of x's children in $\overline{T_0}$ that are adjacent to x in G, we compute the `DFUDS` numbers, p and q, of the first and the last child of x in $\overline{T_0}$. Then u is equal to the number of 1s in $B_0[p..q]$, which can be computed in constant time by performing `bin_rank` on B_0. The number of x's children in T_1 or T_2 that are adjacent to x in G can be computed similarly.

To use the algorithms in the proof of Lemma 6.1 to support `select_neighbor_ccw` and `rank_neighbor_ccw`, it suffices to support these two operations: given a vertex x, select its i^{th} child in $\overline{T_0}$ (T_1 or T_2) that is adjacent to it in G; given a vertex x and a child, y, of it in $\overline{T_0}$ (T_1 or T_2) that are adjacent to x, compute the number of y's left siblings that are adjacent to x. To support these two operations, we first compute the `DFUDS` numbers in $\overline{T_0}$ (numbers in π_1 or π_2) of the first and last children of x in $\overline{T_0}$ (T_1 or T_2). From these, we can locate the substring of B_0 (B_1 or B_2) corresponding to the children of x in $\overline{T_0}$ (T_1 or T_2), and perform rank/select operations on it to support these two operations in constant time.

Finally, we observe that the algorithms in the proof of Lemma 6.1 to support Π_j and Π_j^{-1} on planar triangulations can be used directly here. $\square$

We construct the following succinct indexes for vertex labeled planar graphs.

Theorem 6.5. *Consider a multi-labeled, straight-line embedded planar graph G of n vertices, associated with σ labels in t pairs ($t \geq n$). Given the support of* `node_label` *in $f(n, \sigma, t)$ time on the vertices of G, there is a succinct index using $t \cdot o(\lg \sigma)$ bits which supports* `lab_degree`*,* `lab_select_ccw` *and* `lab_rank_ccw` *in $O((\lg\lg\lg \sigma)^2(f(n, \sigma, t) + \lg\lg \sigma))$ time.*

Proof. We represent the combinatorial structure of G using Theorem 6.4. The vertices of G in canonical order and the set of labels $[\sigma]$ form a binary relation L. As `node_label` directly supports `object_select` on L, we construct a succinct index of $t \cdot o(\lg \sigma)$ bits using Theorem 3.3 for L.

In addition, we construct three binary relations, L_0, L_1 and L_2, between the vertices and the set of labels. In L_0, the i^{th} object corresponds to the i^{th} vertex in `DFUDS` order of $\overline{T_0}$. If this vertex and its parent in $\overline{T_0}$ are adjacent in G, we associate its labels with the i^{th} object. Otherwise, we do not associate any label with this object. As we can perform constant time conversions between the canonical order of a vertex and its `DFUDS` number in $\overline{T_0}$, and we can also check whether a vertex and its parent in $\overline{T_0}$ are adjacent in G, we can use `node_label` to support `object_select` on L_0. We construct L_1 and L_2 using the same approach, except that the i^{th} object in L_1 and L_2 corresponds to the i^{th} vertex in π_1 and π_2, respectively. We can also use `node_label` to support `object_select` on them. We construct a succinct index of $t \cdot o(\lg \sigma)$ bits using Theorem 3.3 for each of these three binary relations. Note that although Theorem 3.3 assumes that each object is associated with at least one label, it still applies to the more general case when an object is associated with zero or more label, as stated in the paragraph after the proof of Theorem 3.3. Furthermore, the result is the same if $t > n$, which is true here.

As we can perform conversions between the `DFUDS` number of $\overline{T_0}$, π_0, π_1 and π_2, we can perform `label_access` on L to check whether the parent of a given vertex in $\overline{T_0}$, T_1 or T_2 is associated with a given label (if this vertex and the parent are adjacent in G). We also observe that if a vertex and one of its child in $\overline{T_0}$, T_1 or T_2 are not adjacent in G, then the object in L_0, L_1 and L_2 that corresponds to the child is not associated with any label. Thus we can use the algorithms in the proof of Theorem 6.2 to support `lab_degree`, `lab_select_ccw` and `lab_rank_ccw`, and this theorem follows. □

To design a succinct representation for a vertex labeled planar graph based on the above theorem, we can use the approach in the proof of Corollary 6.1, and the following corollary is immediate.

Corollary 6.3. *A multi-labeled, straight-line embedded planar graph G of n vertices, associated with σ labels in t pairs ($t \geq n$) can be represented using $\lg \binom{n\sigma}{t} + t \cdot o(\lg \sigma)$ bits to support* `node_label` *in $O(1)$ time, and* `lab_degree`*,* `lab_select_ccw` *and* `lab_rank_ccw` *in $O((\lg\lg\lg \sigma)^2 \lg\lg \sigma)$ time.*

We now design succinct indexes for edge labeled planar graphs.

Theorem 6.6. *Consider a multi-labeled, straight-line embedded planar graph G of n vertices and m edges, in which the edges are associated with σ labels in t pairs ($t \geq m$). Given the support of* `edge_label` *in $f(n, \sigma, t)$ time on the edges of $\mathcal{T}$, there is a succinct index using $t \cdot o(\lg \sigma) + O(n)$ bits which supports* `lab_adjacency` *in $O(\lg\lg\lg \sigma f(n, \sigma, t) + \lg\lg \sigma)$ time, and* `lab_degree_edge`*,* `lab_select_edge_ccw` *and* `lab_rank_edge_ccw` *in $O((\lg\lg\lg \sigma)^2(f(n, \sigma, t) + \lg\lg \sigma))$ time.*

Proof. We add labels to the edges of the planar triangulation $\mathcal{T}$ constructed in this section for G as follows. For each edge of $\mathcal{T}$ that is in G, we label it with its labels in G. For each edge of $\mathcal{T}$ that is not in G, we do not associate it with any label. We also construct B_0, B_1 and B_2. As we can check whether an edge of $\mathcal{T}$ is in G in constant time, we can support `edge_label` on $\mathcal{T}$ in $f(n, \sigma, t)$ time using B_0, B_1 and B_2. We use Theorem 6.3 to construct a succinct index for the edge-labeled version of $\mathcal{T}$.

To analyze the space cost, we observe that to encode the succinct indexes for the three binary relations in the proof of Theorem 6.6, we need $t \cdot o(\lg \sigma) + m' + o(m')$ bits, according to the discussions in the paragraph after the proof of Theorem 3.3. It requires $O(n)$ bits to encode the combinatorial structure of $\mathcal{T}$. B_0, B_1 and B_2 occupy $3n + o(n)$ bits. Thus the overall space cost is $t \cdot o(\lg \sigma) + O(n)$ bits.

Observe that although we add more edges when constructing $\mathcal{T}$, none of them is associated with any labels. Therefore, the operations `lab_adjacency`, `lab_degree_edge`, `lab_select_edge_ccw` and `lab_rank_edge_ccw` can be used directly to support the same operations on G, and the theorem follows. □

To design a succinct representation for an edge labeled planar graph based on the above theorem, we have the following corollary.

Corollary 6.4. *A multi-labeled planar graph G of n vertices and m edges, in which the edges are associated with σ labels in t pairs ($t \geq m$), can be represented using* $\lg \binom{m\sigma}{t} + t \cdot o(\lg \sigma) + O(n)$ *bits to support* `edge_label` *in $O(1)$ time,* `lab_adjacency` *in $O(\lg \lg \sigma)$ time, and* `lab_degree_edge`*,* `lab_select_edge_ccw` *and* `lab_rank_edge_ccw` *in $O((\lg \lg \lg \sigma)^2 \cdot \lg \lg \sigma)$ time.*

Proof. We encode the combinatorial structure of the planar triangulation $\mathcal{T}$ using Theorem 6.1. We construct an edge-labeled version of $\mathcal{T}$ as in the proof of Theorem 6.6. Compute the realizer (T_0, T_1, T_2) of $\mathcal{T}$. Let m_1', m_2' and m_3' denote the numbers of edges of $\mathcal{T}$ in $\overline{T_0}$, T_1 and T_2, respectively. Let m_1, m_2 and m_3 denote the numbers of edges of G in $\overline{T_0}$, T_1 and T_2, respectively. Let t_1, t_2 and t_3 denote the total numbers of edge-label pairs in $\overline{T_0}$, T_1 and T_2, respectively. We use the notion of the three orders, π_0', π_1' and π_2' defined on the edges of $\mathcal{T}$ as in the proof of Theorem 6.6. We construct the three bit vectors B_0, B_1 and B_2 as in the proof of Theorem 6.4.

Consider the edges of G that are in $\overline{T_0}$. Observe that each of them corresponds to a 1 in B_0. These edges in the order of π_0' and the set of labels form a binary relation and we use E_0' to denote it. We use the approach in the proof of Theorem 3.5 to encode E_0' in $m_1 + t_1 + o(m_1 + t_1) + \lg \binom{m_1\sigma}{t_1} + O(\lg \lg(n\sigma))$ bits to support `object_select` on it in constant time. Similarly, we define two binary relations E_1' and E_2' between the edges of G in T_1 and T_2 in the orders of π_1' and π_2', respectively, and the set of labels. We use the same approach to encode them. Thus the total space used to encode these three binary relations is $\sum_{i=0}^{2}(m_i+t_i+o(m_i+t_i)+\lg \binom{m_i\sigma}{t_i}+O(\lg \lg(n\sigma)) \leq m+t+o(m+t)+\lg \binom{m\sigma}{t}+O(\lg \lg(n\sigma))$.

To use Theorem 6.6 to prove this corollary, it suffices to support `edge_label`(x, y, r) on G. As the case when x and y are not adjacent in G is trivial, we only consider the case when they are adjacent. We first consider the case when the edge (x, y) is in $\overline{T_0}$. Assume, without the loss of generality, that x is y's parent in $\overline{T_0}$. Let j be y's `DFUDS` number in $\overline{T_0}$. In this case, the edge (x, y) is numbered j in π_0', which corresponds to the $(k = \texttt{bin\_rank}_{B_0}(1, j))^{\text{th}}$ edge in E_0'. Thus $\texttt{object\_select}_{E_0'}(k, r)$ is the result. The case when the edge (x, y) is in T_1 or T_2 can be handled similarly.

The overall space is $t \cdot o(\lg \sigma) + O(n) + m + t + o(m+t) + \lg \binom{m\sigma}{t} + O(\lg\lg(n\sigma)) = \lg \binom{m\sigma}{t} + t \cdot o(\lg \sigma) + O(n)$ bits. □

6.5 k-Page Graphs

6.5.1 Multiple Parentheses

To present our result on multiple parentheses, we first consider the following operation on strings: $\texttt{string\_rank}'_S(\alpha, i)$, which returns the number of characters α in $S[1..i]$ if $S[i] = \alpha$. We have the following lemma.

Lemma 6.10. *A string S of length n over alphabet $[\sigma]$ can be represented using $n(H_0(S) + o(\lg \sigma))$ bits to support* `string_access` *and* `string_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg\lg \sigma)$ time, and* `string_rank′` *and* `string_select` *for any character $\alpha \in [\sigma]$ in $O(1)$ time. Given a character $\alpha \in [\sigma]$, this representation also supports in $O(1)$ time the computation of the number of characters of S that are lexicographically smaller than α.*

Alternatively, S can be represented using $n(H_0(S) + \epsilon \lg \sigma + o(\lg \sigma))$ bits for any ϵ such that $0 < \epsilon < 1$ to support `string_access` *in $O(1)$ time, while providing the same support for all the other operations above.*

Proof. To prove the result in the first paragraph of this lemma, we use Lemma 4.11 to encode string S in $n(H_0(S) + o(\lg \sigma))$ bits. Thus we only need show how to support $\texttt{string\_rank}'(\alpha, i)$, and how to compute the number of characters of S that are lexicographically smaller than a given character α. We use the bit vector $E[\alpha]$ defined in the proof of Lemma 4.11. It is shown in the same proof how to support `bin_rank′` on $E[\alpha]$ in constant time. As $\texttt{string\_rank}'(\alpha, i) = \texttt{bin\_rank}'_{E[\alpha]}(1, i)$, we can support `string_rank′` in constant time. To compute the number of characters of S that are lexicographically smaller than α, recall that in Lemma 4.11, we treat S as a conceptual table E. Thus we only need compute the number of 1s in the first $\alpha - 1$ rows of E. This can be computed by performing rank/select operations on the bit vector B (see the proof of Theorem 3.2 for the definition of B), as it is exactly the number of 1s in B before the $(n(\sigma-1)/\sigma)^{\text{th}}$ 0.

To achieve the second result, observe that the proof of Lemma 4.11 uses the succinct indexes for strings as in Theorem 3.2. If we increase the size of the auxiliary data structures for the permutations defined in the proof of Theorem 3.2 to $\epsilon n \lg \sigma$, then we can support `string_access` on S in constant time when we apply the result of Lemma 4.11 to our problem. $\square$

We now consider the succinct representations of multiple parenthesis sequences of p types of parentheses, where p is not a constant. We consider the following operation on a multiple parenthesis sequence $S[1..2n]$ in addition to those defined in Section 6.3.1: $\texttt{m\_rank}'_S(i)$, the rank of the parenthesis at position i among parentheses of the same type in S. We have the following theorem.

Theorem 6.7. *A multiple parenthesis sequence of $2n$ parentheses of p types, in which the parentheses of any given type are balanced, can be represented using $2n \lg p + n \cdot o(\lg p)$ bits to support* `m_access`*,* $\texttt{m\_rank}'$*,* `m_findopen` *and* `m_findclose` *in $O(\lg\lg p)$ time, and* `m_select` *in $O(1)$ time. Alternatively, $(2+\epsilon)n \lg p + n \cdot o(\lg p)$ bits are sufficient to support these operations in $O(1)$ time, for any constant ϵ such that $0 < \epsilon < 1$.*

Proof. We store the sequence as a string P over alphabet $\{'(_1', ')_1', '(_2', ')_2', ..., '(_p', ')_p'\}$ using the result in the first paragraph of Lemma 6.10. P occupies at most $2n(\lg p + o(\lg p))$ bits.

For each integer i such that $1 \le i \le p$, we construct a balanced parenthesis sequence B_i, where $B_i[j]$ is an opening (closing) parenthesis iff the j^{th} parenthesis of type i in P is an opening (closing) parenthesis. We denote the number of parentheses of type i by n_i. Thus the length of B_i is n_i. We store each B_i using part (a) of Lemma 2.1. Thus the total space cost of these bit vectors is $\sum_{i=1}^{p}(n_i + o(n_i)) = 2n + o(n)$ bits. To store all these bit vectors, we concatenate them to get a bit vector B. In order to locate B_i in B, it suffices to compute the numbers of characters in P that are lexicographically smaller than $'(_i'$ and $'(_{i+1}'$, which is supported in constant time by Lemma 6.10.

The operation `m_access` can be supported by calling `string_access` on P once, so it can be supported in $O(\lg\lg p)$ time. To support $\texttt{m\_rank}'(i)$, we first compute the parenthesis, α, at position i using `m_access` in $O(\lg\lg p)$ time. Then $\texttt{m\_rank}'(i) = \texttt{string\_rank}'(\alpha, i)$. We also have $\texttt{m\_select}(\alpha, i) = \texttt{string\_select}(\alpha, i)$. Finally, to support $\texttt{m\_match}(i)$, we first find out which parenthesis is at position i using `m_access`. Assume, without loss of generality, it is an closing parenthesis, and let $')_j'$ be this parenthesis. Then we have $\texttt{m\_match}(i) = \texttt{m\_select}('(_j', \texttt{find\_close}_{B_j}(\texttt{string\_rank}'('(_p', i)))$.

To support all these operations in constant time, it suffices to support `string_access` on P in constant time. This can be achieved by using the result in the second paragraph of Lemma 6.10. The total space is thus increased by $\epsilon n \lg p$ bits. □

6.5.2 k-Page Graphs for large k

On unlabeled k-page graphs, we consider the operators `adjacency` and `degree` defined in Section 6.4.2, and the operator `neighbors`(x), returning the neighbors of x.

As shown in Section 6.2, previous results on succinctly representing k-page graphs [70, 71, 35] support `adjacency` in $O(k)$ time. The lower-order term in the space cost of the result of Gavoille and Hanusse [35] is $o(km)$, which is dominant when k is large. Thus previous results mainly deal with the case when k is small. We consider large k.

In this section, we denote each vertex of a k-page graph by its rank along the spine of the book (i.e. vertex x is the x^{th} vertex along the spine). We define the *span* of an edge between vertices x and y to be $|y - x|$. An edge between vertices x and y is a *left edge* (or *right edge*) of x if $y > x$ (or $y < x$). We show the following result.

Theorem 6.8. *A k-page graph G of n vertices and m edges can be represented using $n + 2m \lg k + m \cdot o(\lg k)$ bits to support* `adjacency` *in $O(\lg k \lg\lg k)$ time,* `degree` *in $O(1)$ time, and* `neighbors`(x) *in $O(d(x) \lg\lg k)$ time where $d(x)$ is the degree of x. Alternatively, it can be represented in $n + (2 + \epsilon)m \lg k + m \cdot o(\lg k)$ bits to support* `adjacency` *in $O(\lg k)$ time,* `degree` *in $O(1)$ time, and* `neighbors`(x) *in $O(d(x))$ time, for any constant ϵ such that $0 < \epsilon < 1$.*

Proof. We construct a bit vector B of $n + m$ bits to encode the degree of each node in unary as in [55], in which vertex x corresponds to the x^{th} 1 followed by $d(x)$ 0s. We encode B in $n + m + o(n + m)$ bits using part (a) of Lemma 2.1 to support rank/select operations. We construct a multiple parenthesis sequence S of $2m$ parentheses of k types as follows. For each node $x \in \{1, 2, \cdots, n\}$ and for each page $i \in \{1, 2, \cdots, k\}$:

1. If there are j left edges of x on page i where $j > 0$, we write down $j - 1$ copies of the symbol $')_i'$.

2. Assume that the left edges of x are on pages $p_1, p_2, ..., p_l$. We sort the sequence $')_{p_1}', ')_{p_2}', ..., ')_{p_l}'$ by the maximum span of the left edges of x on these pages and write down the sorted sequence, i.e. in the sorted sequence, $')_{p_u}'$ appears before $')_{p_v}'$ if the maximum span of the left edges of x on page p_u is less than the maximum span of the left edges of x on page p_v.

3. Similarly, we assume that the right edges of x are on pages $q_1, q_2, ..., q_r$. We sort the sequence $'(_{q_1}', '(_{q_2}', ..., '(_{p_l}'$ by the maximum span of the right edges of x on these pages and write down the sorted sequence in descending order.

4. If there are j' right edges of x on page i where $j' > 0$, we write down $j' - 1$ copies of $'(_i'$.

Although the sequence, S, appears to be similar to the sequence in Theorem 2 of [35], it differs in the order we store the parentheses corresponding to the edges of a given vertex. It also has $2m$ parentheses of k types, and we encode it using Theorem 6.7 in $2m \lg k + m \cdot o(\lg k)$ bits. Finally we construct a bit vector B' of $2m$ bits in which $B[i] = 1$ iff $S[i]$ is a closing parentheses, and encoding it in $2m + o(m)$ using part (a) of Lemma 2.1 bits to support rank/select operations. Thus the total space cost is $n + 2m \lg k + m \cdot o(\lg k)$ bits.

With the above definitions and structures, the algorithm [35] which checks whether there is an edge between vertices x and y on page p can be described as follows (assume, without loss of generality, that $x < y$). Let w be the index of the parenthesis in S that corresponds to the right edge of x with the largest span on page p. Observe that this occurrence is the first occurrence of the character $(_p$ in S after position $\texttt{bin\_rank}_B(0, \texttt{bin\_select}_B(1, x))$. Assume that w is given (to use this to support `adjacency`, it suffices to assume that w is given, as shown in the next paragraph). We retrieve the index, t, of the closing parenthesis that matches $B[w]$ in $O(\lg \lg k)$ time, and if it corresponds to a left edge of y (this is true iff $\texttt{bin\_rank}_B(1, t) = y$), then there is an edge between x and y. Similarly, we retrieve the parenthesis in S that corresponds to the left edge of y with the largest span on page p (again, we assume that the index of the occurrence of the parenthesis corresponding to it is given), and if its matching opening parenthesis corresponds to a right edge of x, then x and y are adjacent. If the above process cannot find an edge between x and y, then x and

y are not adjacent on page p. All these steps take $O(\lg \lg k)$ time.

To compute `adjacency`(x, y) (assume, without loss of generality, that $x < y$), we first observe that by Step 2 of the construction algorithm of S, the opening parentheses that correspond to the right edges of x with the largest spans among the right edges of x on the same pages form a substring of S. We can compute the starting position of this substring using B and B' in constant time. Because these parentheses are sorted by the spans of the edges they correspond to, we can perform a doubling searching to check whether one of these edges connects x and y. In each step of the doubling search, we perform the algorithm in the last paragraph in $O(\lg \lg k)$ time. There are at most k such parentheses, so we perform the algorithm $O(\lg k)$ times. Similarly, we perform doubling search on the left edges of y with the largest spans among the left edges of y on the same pages. Thus we can test the adjacency between two vertices in $O(\lg k \lg \lg k)$ time.

The degree of any vertex can be easily computed in constant time using B. We can also perform the algorithms in previous work [35] to compute `neighbors`(x). More precisely, for each opening (or closing) parenthesis corresponding to a right (or left) edge incident to x, we find its matching parenthesis to locate the other vertex that it is incident to. This takes $O(d(x) \lg \lg k)$ time on our data structures.

Finally, to improve the time efficiency, we can store S using $(2 + \epsilon)m \lg k + m \cdot o(\lg k)$ bits using Theorem 6.7 to achieve the other tradeoff. □

6.5.3 Edge Labeled k-Page Graphs

On edge labeled k-page graphs, we consider `lab_adjacency` and `lab_degree_edge` defined in Section 6.4.4, as well as the following operation: `lab_edges`(α, x), the edges incident to vertex x that are labeled α. We define the interface of the ADT of labeled k-page graphs through the operator `edge_label`, as defined in Section 6.4.4.

We first design a succinct index for an edge labeled graph with one page.

Lemma 6.11. *Consider a multi-labeled outerplanar graph G of n vertices and m edges, in which the edges are associated with σ labels in t pairs ($t \geq m$). Given the support of* `edge_label` *in $f(n, \sigma, t)$ time on the edges of G, there is a succinct index using $t \cdot o(\lg \sigma) + n + o(n)$ bits which supports* `lab_adjacency` *in $O(\lg \lg \lg \sigma f(n, \sigma, t) + \lg \lg \sigma)$*

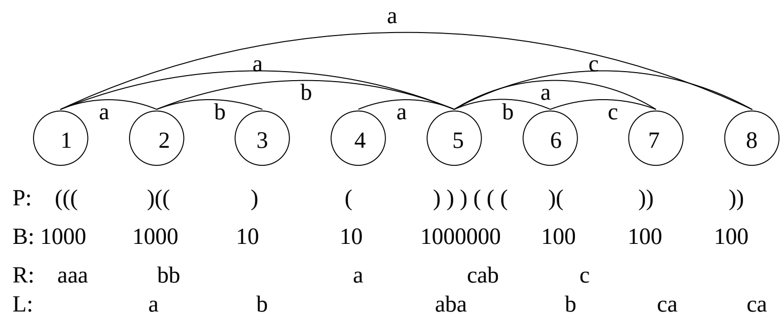


Figure 6.5: An example of the succinct representation of a labeled graph with one page. For simplicity, each edge is associated with exact one label in this example.

time, `lab_degree_edge` *in* $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ *time, and* `lab_edges`(α,x) *in* $O(d(\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ *time, where* $d=$ `lab_degree_edge`(α,x).

Proof. We construct a bit vector B of $n+m$ bits to encode the degree of each node in unary as in the proof of Theorem 6.8, and use part (a) of Lemma 2.1 to encode it. We construct a balanced parenthesis sequence P as follows. List the vertices from left to right along the spine, and each node is represented by zero or more closing parentheses followed by zero or more opening parentheses, where the number of closing (or opening) parentheses is equal to the number of its left (or right) edges. The edges sorted by the positions of the corresponding opening parentheses and the set of labels form a binary relation R. Similarly, the edges sorted by the positions of the corresponding closing parentheses and the set of labels form a binary relation L. See Figure 6.5 for an example.

To compute `object_select`$_R(x,r)$, we first find the two vertices y and z $(y<z)$ that the edge corresponding to the x^{th} opening parenthesis in P is incident to. As this parenthesis corresponds to the i^{th} 0 in B, where $i=$ `bin_select`$_B(0,x)$, we have $y=$ `bin_rank`$_B(1,i)$. We find the closing parenthesis that matches this opening parenthesis in P using `find_close`, and z can be computed similarly. As `object_select`$_R(x,r)=$ `edge_label`(y,z,r), we can support `object_select` on R in $f(n,\sigma,t)$ time. The support for `object_select` on L is similar. We then build a succinct index of $t\cdot o(\lg\sigma)$ bits for either of L and R using Theorem 3.5. These data structures occupy $t\cdot o(\lg\sigma)+n+o(n)$ bits in total as $t\geq m$.

To compute `lab_adjacency`(α,x,y), we first use the algorithm by Jacobson [55] to check whether x and y are adjacent. If they are, we retrieve the position of the opening parenthesis in P that corresponds to the edge between x and y, compute its rank, v, among opening parenthesis, and we return the result of `label_access`(v, α) on R. This takes $O(\lg\lg\lg\sigma f(n,\sigma,t) + \lg\lg\sigma)$ time.

To compute `lab_degree_edge`(α, x), we need compute the number, l, of the left edges of x that are labeled α, and the number, r, of the right edges of x that are labeled α. To compute l, we first compute the positions l_1 and l_2 such that each parenthesis in the substring $P[l_1..l_2]$ is a closing parentheses that corresponds to a left edge of x, using rank/select operations on B and P in constant time. We then use `label_rank` and `label_select` on L to compute the number of objects associated with α between and including objects l_1 and l_2 in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t) + \lg\lg\sigma))$ time. Similarly we can compute r by performing rank/select operations on B, P and R, and the sum of l and r is the answer. To further list all the edges of x that is labeled α, we need perform `label_succ` on L and R to retrieve the positions of the corresponding parentheses in P, and perform rank/select operations on B to retrieve the vertices that these edges are incident to. □

We now use the above lemma to design a succinct representation of edge labeled outerplanar graph.

Lemma 6.12. *An outerplanar graph of n vertices and m edges in which the edges are associated with σ labels in t pairs ($t \geq m$) can be represented using $n + o(n) + \lg\binom{m\sigma}{t} + t \cdot o(\lg\sigma)$ bits to support:*

- `edge_label` *in $O(1)$ time;*
- `lab_adjacency` *in $O(\lg\lg\sigma)$ time;*
- `lab_degree_edge` *in $O((\lg\lg\lg\sigma)^2 \lg\lg\sigma)$ time;*
- `lab_edges`(α, x) *in $O(d(\lg\lg\lg\sigma)^2 \lg\lg\sigma)$ time, where $d =$ `lab_degree_edge`(α, x).*

Proof. We construct B and P as in the proof of Lemma 6.11. We use Theorem 3.5 to represent the binary relation R defined in the proof of Lemma 6.11. This costs $\lg\binom{m\sigma}{t} + t \cdot o(\lg\sigma)$ bits. Given two adjacent vertices x and y, we can locate the opening parenthesis corresponding to the edge between x and y using P and B in constant time. Thus we can use `object_select` on R to directly support `edge_label`, which in turn supports `object_select` on L. Hence we can construct a succinct index of $t \cdot \lg \sigma$ bits for the binary relation L defined in the proof of Lemma 6.11, and this lemma immediately follows. □

To represent an edge labeled k-page graph, we can use Lemma 6.12 to represent each page and combine all the pages represented in this way to support operations. Alternatively, we can use Theorem 6.8 and an approach similar to Lemma 6.12 to achieve a different tradeoff to improve the time efficiency for large k. As we consider general k, the auxiliary data structures may occupy more space than the labels themselves. Thus we choose to directly show our succinct representations instead of presenting a succinct index first. Note that for sufficiently small k, this approach can still be used to construct a succinct index.

Theorem 6.9. *A k-page graph G of n vertices and m edges, in which the edges are associated with σ labels in t pairs ($t \geq m$), can be represented using $k(n+o(n))+\lg \binom{m\sigma}{t}+t\cdot o(\lg \sigma)$ bits to support:*

- `edge_label` *in $O(k)$ time;*
- `lab_adjacency` *in $O(\lg\lg\sigma + k)$ time;*
- `lab_degree_edge` *in $O(k(\lg\lg\lg\sigma)^2 \lg\lg\sigma)$ time;*
- `lab_edges`(α, x) *in $O(d(\lg\lg\lg\sigma)^2 \lg\lg\sigma+k)$ time, where $d =$* `lab_degree_edge`(α, x).

Alternatively, it can be represented using $n+o(n)+2m(\lg k+o(\lg k))+\lg \binom{m\sigma}{t}+m\cdot o(\lg \sigma)$ bits to support:

- `edge_label` *in $O(1)$ time;*
- `lab_adjacency` *in $O(\lg\lg\sigma + \lg k)$ time;*
- `lab_degree_edge` *in $O((\lg\lg\lg\sigma)^2 \lg\lg\sigma)$ time;*
- `lab_edges`(α, x) *in $O(d(\lg\lg\lg\sigma)^2 \lg\lg\sigma)$ time, where $d =$* `lab_degree_edge`(α, x).

Proof. To prove the first result, we use Lemma 6.12 to represent each page. Assume that m_i pages are embedded in the i^{th} page, and that there are t_i edge-label pairs between them and the alphabet set. The total space cost in bits is $k(n+o(n))+\sum_{i=1}^{k}(\lg\binom{m_i\sigma}{t_i}+t_i\cdot o(\lg\sigma))\leq k(n+o(n))+\lg\binom{m\sigma}{t}+t\cdot o(\lg\sigma)$. To support the above operations on G, we perform them on each page. Note that to perform `lab_adjacency` on a page, it only takes constant time if the edge between these two vertices is not embedded in this page. It is then easy to show the above running time of each operation is correct.

To prove the second result, we use the second result of Theorem 6.8 to encode the combinatorial structure of G. Recall that in its proof, we construct a multiple parenthesis sequence S, and two bit vectors B and B'. To encode the labels, we use an approach similar to that used in the proof of Lemma 6.12. We observe that the edges of G sorted by the positions of the corresponding opening parentheses (of any type) in S and the set of labels form a binary relation, and we denote this relation by R'. Similarly, the edges of G sorted by the positions of the corresponding closing parentheses (of any type) in S and the set of labels form a binary relation L'. We use Theorem 3.5 to represent the binary relation R' in $\lg\binom{m\sigma}{t}+t\cdot o(\lg\sigma)$ bits. Observe that for the i^{th} closing parenthesis in S, we can locate the position of the matching opening parenthesis using `m_match`, and compute the number of opening parentheses preceding it in S using B'. This can be performed in constant time. Thus we can use `object_select` on R' to directly support `object_select` on L'. We construct a succinct index of $t\cdot\lg\sigma$ bits using Theorem 3.3 for the binary relation L'. All these data structures occupy $n+o(n)+(2m+\epsilon)\lg k+m\cdot o(\lg k)+\lg\binom{m\sigma}{t}+m\cdot o(\lg\sigma)=n+o(n)+2m(\lg k+o(\lg k))+\lg\binom{m\sigma}{t}+m\cdot o(\lg\sigma)$ bits, as $k\leq m^2$.

With the above data structures, the algorithms in the proof of Lemma 6.11 can be easily modified to support the operations on G, and the theorem follows. □

As a planar graph can be embedded in at most 4 pages [86], we have the following corollary.

Corollary 6.5. *An edge-labeled planar graph of n vertices and m edges, in which the edges are associated with σ labels in t pairs ($t\geq m$), can be represented using $n+o(n)+\lg\binom{m\sigma}{t}+t\cdot o(\lg\sigma)$ bits to support:*

- `edge_label` *in $O(1)$ time;*

- `lab_adjacency` *in* $O(\lg\lg\sigma)$ *time;*
- `lab_degree_edge` *in* $O((\lg\lg\lg\sigma)^2\lg\lg\sigma)$ *time;*
- `lab_edges`(α, x) *in* $O(d(\lg\lg\lg\sigma)^2\lg\lg\sigma)$ *time, where* $d =$ `lab_degree_edge`(α, x).

Proof. When we prove the second result in Theorem 6.9, we use the second result in Theorem 6.7 to encode the multiple parenthesis sequence S. Theorem 6.7 applies to the case when the number of types of parentheses is non-constant. To prove this theorem, as a planar graph can be embedded in at most 4 pages, the number of type of parentheses in S is 4. Thus we can use Lemma 6.1 to represent S and the corollary directly follows. □

An alternative approach to achieve a similar result is to compute a straight-line embedding of the planar graph first, and then use Corollary 6.4 to represent it. The space cost is increased to $O(n) + \lg\binom{m\sigma}{t} + t \cdot o(\lg\sigma)$ bits. Thus Corollary 6.4 is more suitable when we need to use label-based rank/select operations in ccw order on edge labeled planar graphs that are already straight-line embedded in the plane.

6.6 Discussion

In this chapter, we present a framework for succinct representation of properties of graphs in the form of labels. Our main results are the succinct representations of labeled and multi-labeled graphs (we consider vertex/edge labeled planar triangulations, vertex/edge labeled planar graphs, as well as edge labeled k-page graphs) to support various label queries efficiently. The additional space cost to store the labels is essentially the information-theoretic minimum. As far as we know, our representations are the first succinct representations of labeled graphs. We also present two preliminary results on unlabeled graphs to achieve the main results. First, we design a succinct representation of unlabeled planar triangulations and straight-line embedded planar graphs to support the rank/select of edges in ccw (counter clockwise) order in addition to the other operations supported in previous work [18, 19, 17, 15, 16]. Second, we design a succinct representation for a k-page graph when k is large to support various navigational operations more efficiently. In particular, we can test the adjacency of two vertices in $O(\lg k \lg\lg k)$ time, while previous work uses $O(k)$ time [70, 71, 35].

We expect that our approach can be extended to support some of the other types of graphs, which is an open research topic. Another open problem is to represent vertex labeled k-page graphs succinctly.

Our final comment is that because Theorem 6.2, Theorem 6.3, Theorem 6.5 and Theorem 6.6 provide succinct indexes for vertex/edge labeled planar triangulations and planar graphs, we can in fact store the labels in compressed form as we have done in Theorem 3.4 to compress strings, while still providing the same support for operations. This also applies to Theorem 6.9, where we apply succinct indexes for binary relations to encode the labels.

Chapter 7

Conclusion

In this thesis, we define succinct indexes for the design of data structures. We show their advantages by presenting succinct indexes for strings, binary relations, multi-labeled trees and multi-labeled graphs, and by applying them to various applications.

Using our techniques, we design a succinct encoding that represents a string of length n over an alphabet of size σ using $nH_k + \lg \sigma \cdot o(n) + n \cdot o(\lg \sigma)$ bits to support access/rank/select operations in $O((\lg \lg \sigma)^{1+\epsilon})$ time, for any fixed constant $\epsilon > 0$. We also design a succinct text index using $n(H_k + o(\lg \sigma))$ bits that supports pattern matching queries in $O(m \lg \lg \sigma + \mathtt{occ} \lg^{1+\epsilon} n \lg \lg \sigma)$ time, for a given pattern of length m. Previous results on these two problems either have a $\lg \sigma$ factor instead of $\lg \lg \sigma$ in terms of running time [44], or are not easily compressible [41]. We also design a succinct encoding that represents a binary relation formed by t pairs between n objects and σ labels using $\lg \binom{n\sigma}{t} + t \cdot o(\lg \sigma)$ bits to support various types of rank/select operations efficiently. This space cost is close to the information-theoretic minimum. Our succinct representation of multi-labeled trees supports label-based ancestor, child and descendant queries at the same time, while previous results do not [36, 37, 29, 5]. We also design the first succinct representations of multi-labeled graphs.

The concept of succinct indexes is of both theoretical and practical importance to the design of data structures. In theory, the separation of the ADT and the index enables researchers to design an encoding of the given data to achieve desired results or tradeoffs more easily, as the encoding only need support the ADT. In addition, to support new operations, researchers merely need design additional succinct indexes without redesigning the whole structure. In practice, this concept allows developers to engineer the implementation of ADTs and succinct indexes separately. The fact that multiple succinct indexes for the same ADT can be easily combined to provide one succinct index makes it possible to further divide the implementation of succinct indexes into several (possibly concurrent) steps. This is good software engineering practice, to allow separate testing and concurrent development, and to facilitate the design of expandable software libraries. Furthermore, succinct indexes provide a way to support efficient operations on implicit data, which is common in both theory and practice. We thus expect that the concept of succinct indexes will influence the design of succinct data structures.

Other contributions of the thesis include various preliminary results that we obtain in order to design succinct indexes. We present a theorem that characterizes a permutation as a suffix array, based on which we design succinct text indexes. We design a succinct representation of ordinal trees that supports all the navigational operations supported by various succinct tree representations while requiring only $2n + o(n)$ bits. In addition, this representation also supports two other encodings schemes, the balanced parenthesis sequence [70, 71] and the `DFUDS` sequence [10, 9], of ordinal trees as abstract data types. To design succinct indexes for multi-labeled graphs, we design a succinct representation of unlabeled planar triangulations and straight-line embedded planar graphs to support the rank/select of edges in ccw (counter clockwise) order in addition to the other operations supported in previous work [18, 19, 17, 15, 16]. We also design a succinct representation for a k-page graph when k is large to support various navigational operations more efficiently. In particular, we can test the adjacency of two vertices in $O(\lg k \lg \lg k)$ time, while previous work uses $O(k)$ time [70, 71, 35].

In addition to the specific open problems mentioned in the discussion part of each chapter, there are several general directions for future work. The work in this thesis concentrates on static data structures; we aim at designing succinct representations of static data structures that allow data to be retrieved efficiently. However, various large applications require data to be updated frequently. For example, general-purpose XML databases usually need provide support for update operations. Previous results on succinct dynamic data structures primarily focus on designing succinct integrated encodings [73, 76, 63, 47]. How to design succinct indexes for dynamic data structures is thus an open research field.

The results in this thesis are for data structures in internal memory. Another approach of handling large data sets is to design IO-efficient algorithms and data structures. It remains open to extend our work to handle data in external memory. Finally, proving the lower bounds of the sizes of various succinct indexes is another open research field. The issue of lower bounds is addressed in [40], though several related problems still remain open.

Appendix A

Glossary of Definitions

α-predecessor/α-successor (binary relations) Consider a binary relation formed by t pairs from an object set $[n]$ and a label set $[\sigma]$, a literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ and an object $x \in [n]$. The *α-predecessor* of object x, denoted by $\texttt{label\_pred}(\alpha, x)$, is the last object matching α before (and not including) object x, if it exists. Similarly, the *α-successor* of object x, denoted by $\texttt{label\_succ}(\alpha, x)$, is the first object matching α after (and not including) object x, if it exists.

α-predecessor/α-successor (strings) Consider a string $S \in [\sigma]^n$, a literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ and a position $x \in [n]$ in S. The *α-predecessor* of position x, denoted by $\texttt{string\_pred}(\alpha, x)$, is the last position matching α before (and not including) position x, if it exists. Similarly, the *α-successor* of position x, denoted by $\texttt{string\_succ}(\alpha, x)$, is the first position matching α after (and not including) position x, if it exists.

τ^*-name Given a node x whose τ-name is $\tau(x) =< \tau_1(x), \tau_2(x), \tau_3(x) >$, its *$\tau^*$-name* is $\tau^*(x) =< \tau_1(x), \tau_2(x), \tau_3^*(x) >$, if x is the $\tau_3^*(x)^{\text{th}}$ node of its micro-tree in $\texttt{DFUDS}$ order.

ascending run Given a segment $[i, j]$ $(1 < i \leq j \leq n)$ of a permutation $M[1..n]$, we call it an *ascending run* iff for any k, l where $1 \leq k, l < n$, if $i \leq M^{-1}[k] < M^{-1}[l] \leq j$, then $M^{-1}[k+1] < M^{-1}[l+1]$.

ascending-to-max Given a permutation $M[1..n]$ of $[n]$, we call it *ascending-to-max* iff for any integer i where $1 \leq i \leq n-2$, we have:

(i) if $M^{-1}[i] < M^{-1}[n]$ and $M^{-1}[i+1] < M^{-1}[n]$, then $M^{-1}[i] < M^{-1}[i+1]$, and

(ii) if $M^{-1}[i] > M^{-1}[n]$ and $M^{-1}[i+1] > M^{-1}[n]$, then $M^{-1}[i] > M^{-1}[i+1]$.

`DFUDS` changer List the nodes in `DFUDS` order, numbered $1, 2, ..., n$. The i^{th} node in `DFUDS` order is a *tier-1* (or *tier-2*) `DFUDS` *changer* if $i = 1$, or if the i^{th} and $(i-1)^{\text{th}}$ nodes in `DFUDS` order are in different mini-trees (or micro-trees).

k^{th} order empirical entropy Consider a string S of length n over alphabet $[\sigma]$. Given another string $w \in [\sigma]^k$, we define the string w_S to be a concatenation of all the single characters immediately following one of the occurrences of w in S. Then the *k^{th} order empirical entropy* of S is

$$H_k(T) = \frac{1}{|S|} \sum_{w \in [\sigma]^k} |w_S| H_0(w_S).$$

level successor graph The *tier-1* (or *tier-2*) *level successor graph* $G = \{V, E\}$ is a undirected graph in which the i^{th} vertex, v_i, corresponds to the i^{th} tier-1 (or tier-2) preorder segment, and the edge $(v_i, v_j) \in E$ iff there exist nodes x and y in the i^{th} and j^{th} tier-1 (or tier-2) preorder segments, respectively, such that either x is y's level successor, or y is x's level successor.

literal (binary relations) Consider a binary relation formed by t pairs from an object set $[n]$ and a label set $[\sigma]$. An object $x \in [n]$ *matches literal* $\alpha \in [\sigma]$ if x is associated with α. An object $x \in [n]$ *matches literal* $\bar{\alpha}$ if x is not associated with α. For simplicity, we define $[\bar{\sigma}]$ to be the set $\{\bar{1}, \ldots, \bar{\sigma}\}$.

literal (strings) Consider a string $S[1 \ldots n]$ over the alphabet $[\sigma]$. A position $x \in [n]$ *matches literal* $\alpha \in [\sigma]$ if $S[x] = \alpha$. A position $x \in [n]$ *matches literal* $\bar{\alpha}$ if $S[x] \neq \alpha$. For simplicity, we define $[\bar{\sigma}]$ to be the set $\{\bar{1}, \ldots, \bar{\sigma}\}$.

maximal ascending run Given an ascending run $[i, j]$ of a permutation $M[1..n]$, it is a *maximal ascending run* iff for any segment $[s, t]$ of M, if $[s, t] \supset [i, j]$, then $[s, t]$ is not an ascending run.

non-nesting Given a permutation $M[1..n]$ of $[n]$, we call it *non-nesting* iff for any two integers i, j, where $1 \leq i, j \leq n-1$ and $M^{-1}[i] < M^{-1}[j]$, we have:

(i) if $M^{-1}[i] < M^{-1}[i+1]$ and $M^{-1}[j] < M^{-1}[j+1]$, then $M^{-1}[i+1] < M^{-1}[j+1]$, and

(ii) if $M^{-1}[i] > M^{-1}[i+1]$ and $M^{-1}[j] > M^{-1}[j+1]$, then $M^{-1}[i+1] < M^{-1}[j+1]$.

permuted binary relation Given a permutation π on $[n]$ and a binary relation $R \subset [n] \times [\sigma]$, the *permuted binary relation* $\pi(R)$ is the relation such that $(x, \alpha) \in \pi(R)$ if and only if $(\pi^{-1}(x), \alpha) \in R$.

preorder changer Node x is a *tier-1* (or *tier-2*) *preorder changer* if $x = 1$, or if nodes x and $(x-1)$ are in different mini-trees (or micro-trees).

preorder segment A *tier-1* (or *tier-2*) *preorder segment* is a sequence of nodes $x, (x+1), \cdots, (x+i)$ that satisfies:

- Node x is a tier-1 (or tier-2) preorder changer;
- Node $(x+i+1)$ is a tier-1 (or tier-2) preorder changer if $x+i+1 \leq n$;
- None of the nodes $(x+1), (x+2), \cdots, (x+i)$ is a tier-1 (or tier-2) preorder changer.

pseudo leaf Each leaf of a mini-tree (or micro-tree) is a *pseudo leaf* of the original tree T. A pseudo leaf that is also a leaf of T is a *real leaf*. Given a mini-tree (or micro-tree), we mark the leftmost real leaf of the mini-tree (or micro-tree), and the first real leaf in preorder after each subtree of T rooted at a node that is not in the mini-tree (or micro-tree), but is a child of a node in it. These nodes are called *tier-1* (or *tier-2*) *marked leaves*.

realizer A *realizer* of a planar triangulation $\mathcal{T}$ is a partition of the set of the internal edges into three sets T_0, T_1 and T_2 of directed edges, such that for each internal vertex v the following conditions hold:

- v has exactly one outgoing edge in each of the three sets T_0, T_1 and T_2;

- *local condition:* the edges incident to v in counterclockwise (ccw) order are: one outgoing edge in T_0, zero or more incoming edges in T_2, one outgoing edge in T_1, zero or more incoming edges in T_0, one outgoing edge in T_2, and finally zero or more incoming edges in T_1.

recursivity The *recursivity ρ_α of a label* α in a multi-labeled tree is the maximum number of occurrences of α on any rooted path of the tree. The *average recursivity ρ of a multi-labeled tree* is the average recursivity of the labels weighted by the number of nodes associated with each label α (denoted by t_α): $\rho = \frac{1}{t}\sum_{\alpha\in[\sigma]}(t_\alpha\rho_\alpha)$.

reverse pair A *reverse pair* on two given permutations π_1 and π_2 is a pair of integers (i, j), where $1 \le i, j \le n$, such that $\pi_1^{-1}[i] < \pi_1^{-1}[j]$ but $\pi_2^{-1}[i] > \pi_2^{-1}[j]$, i.e. the relative positions of i and j in π_1 and π_2 are different.

three traversal orders on a planar triangulation The *zeroth order*, π_0, is defined on all the vertices of $\mathcal{T}$ and is simply given by the preorder traversal of $\overline{T_0}$ starting at v_0 in *counter clockwise order* (ccw order).

The *first order*, π_1, is defined on the vertices of $\mathcal{T}\setminus v_0$ and corresponds to a traversal of the edges of T_1 as follows. Perform a preorder traversal of the contour of $\overline{T_0}$ in a ccw manner. During this traversal, when visiting a vertex v, we enumerate consecutively its incident edges $(v, u_1), \ldots, (v, u_i)$ in T_1, where v appears before u_i in π_0. The traversal of the edges of T_1 naturally induces an order on the nodes of T_1: each node (different from v_1) is uniquely associated with its parent edge in T_1.

The *second order*, π_2, is defined on the vertices of $\mathcal{T} \setminus \{v_0, v_1\}$ and can be computed in a similar manner by performing a preorder traversal of T_0 in *clockwise order* (cw order). When visiting in cw order the contour of $\overline{T_0}$, the edges in T_2 incident to a node v are listed consecutively to induce an order on the vertices of T_2.

weak visibility representation A *weak visibility representation* of a graph G is a mapping of its vertices into non-overlapping horizontal segments called *vertex segments* and of its edges into vertical segments called *edge segments*. Under this mapping, the edge between any two given vertices x and y is mapped to an edge segment whose end points are on the vertex segments of x and y, and this edge segment does not cross any other vertex segment.

zeroth order empirical entropy The *zeroth order empirical entropy* of a string S of length n over alphabet $[\sigma]$ is

$$H_0(S) = \sum_{\alpha=1}^{\sigma}(p_\alpha \log_2 \frac{1}{p_\alpha}) = -\sum_{\alpha=1}^{\sigma}(p_\alpha \log_2 p_\alpha),$$

where p_α is the frequency of the occurrence of character α, and $0 \log_2 0$ is interpreted as 0.

Bibliography

[1] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[2] Hideo Bannai, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Inferring strings from graphs and arrays. In *Proceedings of the 28th International Symposium on Mathematical Foundations of Computer Science*, pages 208–217. Springer-Verlag LNCS 2747, 2003.

[3] Jérémy Barbay. Adaptive search algorithm for patterns, in succinctly encoded XML. Technical Report CS-2006-11, University of Waterloo, Ontario, Canada, 2006.

[4] Jérémy Barbay, Luca Castelli Aleardi, Meng He, and J. Ian Munro. Succinct representation of labeled graphs. In *Proceedings of the 18th International Symposium on Algorithms and Computation*, pages 316–328. Springer-Verlag LNCS 4835, 2007.

[5] Jérémy Barbay, Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching*, pages 24–35. Springer-Verlag LNCS 4009, 2006.

[6] Jérémy Barbay, Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 680–689, 2007.

[7] Jérémy Barbay and Srinivasa Rao. Succinct encoding for XPath location steps. Technical Report CS-2006-10, University of Waterloo, Ontario, Canada, 2006.

[8] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Theoretical Informatics Symposium*, pages 88–94, 2000.

[9] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[10] David Benoit, Erik D. Demaine, J. Ian Munro, and Venkatesh Raman. Representing trees of higher degree. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, pages 169–180. Springer-Verlag LNCS 1663, 1999.

[11] Frank Bernhart and Paul C. Kainen. The book thickness of a graph. *Journal of Combinatorial Theory, Series B*, 27(3):320–331, 1979.

[12] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 679–688, 2003.

[13] Stefan Burkhardt and Juha Kärkkäinen. Fast light suffix array construction and checking. In *Proceedings of the 14th Symposium on Combinatorial Pattern Matching*, pages 55–69. Springer-Verlag LNCS 2676, 2003.

[14] Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[15] Luca Castelli Aleardi, Olivier Devillers, and Gilles Schaeffer. Succinct representation of triangulations with a boundary. In *Proceedings of the 9th Workshop on Algorithms and Data Structures*, volume 3608 of *LNCS*, pages 134–145. Springer, 2005.

[16] Luca Castelli Aleardi, Olivier Devillers, and Gilles Schaeffer. Optimal succinct representations of planar maps. In *Proceedings of the 22nd Annual ACM Symposium on Computational Geometry*, pages 309–318, 2006.

[17] Yi-Ting Chiang, Ching-Chi Lin, and Hsueh-I Lu. Orderly spanning trees with applications to graph encoding and graph drawing. In *Proceedings of the 12th Annual ACM-SIAM symposium on Discrete algorithms*, pages 506–515, 2001.

[18] Richie Chih-Nan Chuang, Ashim Garg, Xin He, Ming-Yang Kao, and Hsueh-I Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 118–129, 1998.

[19] Richie Chih-Nan Chuang, Ashim Garg, Xin He, Ming-Yang Kao, and Hsueh-I Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. *The Computing Research Repository*, cs.DS/0102005, 2001.

[20] Fan R. K. Chung, Frank Thomson Leighton, and Arnold L. Rosenberg. Embedding graphs in books: a layout problem with applications to VLSI design. *SIAM Journal on Algebraic Discrete Methods*, 8(1):33–58, 1987.

[21] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

[22] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *Proceedings of the 4th Annual ACM symposium on Theory of computing*, pages 73–80, 1972.

[23] O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Engineering the LOUDS succinct tree representation. In *Proceedings of the 5th International Workshop on Experimental Algorithms*, pages 134–145, 2006.

[24] Erik D. Demaine and Alejandro Lopez-Ortiz. A linear lower bound on index size for text retrieval. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 289–294, 2001.

[25] Pierre Duchet, Yahya Ould Hamidoune, Michel Las Vergnas, and Henry Meyniel. Representing a planar graph by vertical lines joining different levels. *Discrete Mathematics*, 46(3):319–321, 1983.

[26] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[27] István Fáry. On straight line representation of planar graphs. *Acta Scientiarum Mathematicarum (Szeged)*, 11:229–233, 1948.

[28] Peter Fenwick. Block sorting text compression – final report. Technical Report 130, University of Auckland, 1996.

[29] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th IEEE Symposium on Foundations of Computer Science*, pages 184–196, 2005.

[30] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[31] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 269–278, 2001.

[32] Paolo Ferragina and Giovanni Manzini. On compressing and indexing data. Technical Report TR-02-01, University of Pisa, 2002.

[33] Paolo Ferragina and Giovanni Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In *Proceedings of the 15th Annual ACM-SIAM symposium on Discrete algorithms*, pages 655–663, 2004.

[34] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly FM-index. In *Proceedings of the 11th Symposium on String Processing and Information Retrieval*, pages 150–160. Springer-Verlag LNCS 3246, 2004.

[35] Cyril Gavoille and Nicolas Hanusse. On compact encoding of pagenumber k graphs. *Discrete Mathematics & Theoretical Computer Science*, 2007. To appear.

[36] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–10, 2004.

[37] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.

[38] Raffaele Giancarlo and Marinella Sciortino. Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, pages 129–143, 2003.

[39] Alexander Golynski. Optimal lower bounds for rank and select indexes. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming*, pages 370–381, 2006.

[40] Alexander Golynski. *Upper and Lower Bounds for Text Indexing Data Structures*. PhD thesis, University of Waterloo, December 2007.

[41] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 368–373, 2006.

[42] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In William B. Frakes and Ricardo A. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 5, pages 66–82. Prentice-Hall, 1992.

[43] Rodrigo González and Gonzalo Navarro. Statistical encoding of succinct data structures. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching*, pages 294–305, 2006.

[44] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.

[45] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 397–406, 2000.

[46] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[47] Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. A framework for dynamizing succinct data structures. In *Proceedings of the 34st International Colloquium on Automata, Languages and Programming*, pages 521–532, 2007.

[48] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA, 1997.

[49] Torben Hagerup. Sorting and searching on the word ram. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, pages 366–398, London, UK, 1998. Springer-Verlag.

[50] Torben Hagerup. Improved shortest paths on the word RAM. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 61–72, 2000.

[51] Donna Harman, Edward A. Fox, Ricardo A. Baeza-Yates, and Whay C. Lee. Inverted files. In William B. Frakes and Ricardo A. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 3, pages 28–43. Prentice-Hall, 1992.

[52] Meng He, J. Ian Munro, and S. Srinivasa Rao. A categorization theorem on suffix arrays with applications to space efficient text indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 23–32, 2005.

[53] Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct ordinal trees based on tree covering. In *Proceedings of the 34st International Colloquium on Automata, Languages and Programming*, pages 509–520, 2007.

[54] Martin Isenburg and Jack Snoeyink. Face fixer: compressing polygon meshes with properties. In *Proceedings of the 27th Annual Conference on Computer Graphics*, pages 263–270, 2000.

[55] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[56] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[57] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proceedings of the 14th Symposium on Combinatorial Pattern Matching*, pages 186–199. Springer-Verlag LNCS 2676, 2003.

[58] David G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.

[59] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2nd edition, 1998.

[60] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Proceedings of the 14th Symposium on Combinatorial Pattern Matching*, pages 200–210. Springer-Verlag LNCS 2676, 2003.

[61] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[62] Hsueh-I Lu and Chia-Chi Yeh. Balanced parentheses strike back. Accepted to *ACM Transactions on Algorithms*, 2007.

[63] Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching*, pages 306–317, 2006.

[64] Veli Mäkinen and Gonzalo Navarro. Position-restricted substring searching. In *Proceedings of the 7th Latin American Theoretical Informatics Symposium*, pages 703–714, 2006.

[65] Udi Manber and Eugene W. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[66] Peter Bro Miltersen. Lower bounds on the size of selection and rank indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 11–12, 2005.

[67] Christian Worm Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 618–627, 2003.

[68] Christian Worm Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM Journal on Computing*, 35(6):1494–1525, 2006.

[69] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, pages 345–356, 2003.

[70] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, pages 118–126, 1997.

[71] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[72] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[73] J. Ian Munro, Venkatesh Raman, and Adam J. Storm. Representing dynamic binary trees succinctly. In *Proceedings of the 12th Annual ACM-SIAM symposium on Discrete algorithms*, pages 529–536, 2001.

[74] J. Ian Munro and S. Srinivasa Rao. Succinct representations of functions. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming*, pages 1006–1015, 2004.

[75] 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*, pages 233–242, 2002.

[76] Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, pages 357–368, 2003.

[77] Arnold L. Rosenberg. The DIOGENES design methodology: toward automatic physical layout. In *Proceedings of the international workshop on Parallel algorithms & architectures*, pages 335–348, 1986.

[78] Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix arrays. In *Proceedings of the 11th International Symposium on Algorithms and Computation*, pages 410–421. Springer-Verlag LNCS 1969, 2000.

[79] Kunihiko Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 225–232, 2002.

[80] Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the 17th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1230–1239, 2006.

[81] Walter Schnyder. Embedding planar graphs on the grid. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 138–148, 1990.

[82] Roberto Tamassia and Ioannis G. Tollis. A unified approach to visibility representations of planar graphs. *Discrete & Computational Geometry*, 1:321–341, 1986.

[83] Robert Endre Tarjan. Sorting using networks of queues and stacks. *Journal of the ACM*, 19(2):341–346, 1972.

[84] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[85] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

[86] Mihalis Yannakakis. Four pages are necessary and sufficient for planar graphs. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Theory of Computing*, pages 104–108, 1986.

[87] Ning Zhang, M. Tamer Özsu, Ashraf Aboulnaga, and Ihab F. Ilyas. XSEED: Accurate and Fast Cardinality Estimation for XPath Queries. In *Proceedings of the 22nd International Conference on Data Engineering*, pages 61–72, 2006.

Index