Succinct Representation of Trees and Graphs

by

Arash Farzan

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

Waterloo, Ontario, Canada, 2009

© Arash Farzan 2009

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

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

Abstract

In this thesis, we study succinct representations of trees and graphs. A succinct representation of a combinatorial object is a space efficient representation that supports a reasonable set of operations and queries on the object in constant or near constant time on the RAM with logarithmic word size. The storage requirement of a succinct representation is intended to be optimal to within lower order terms.

We first propose a uniform approach for succinct representation of various families of trees. The method is based on two recursive decompositions of trees into subtrees. The approach simplifies the existing representation of ordinal trees while allowing the full set of navigational operations and queries. The approach applied to cardinal (*i.e.,* k-ary) trees yields a space-optimal succinct representation allowing cardinal-type operations (*e.g.,* determining child labeled i) as well as the full set of ordinal-type operations (*e.g.,* reporting the number of siblings to the left of a node). Previous space-optimal succinct representations had not been able to support both types of operations efficiently [8, 58]. We demonstrate how the approach can be applied to obtain a space-optimal succinct representation for the family of free trees where the order of children is insignificant. Furthermore, we show that our approach can be used to obtain entropy-based succinct representations. The approach adapts to match the degree-distribution entropy suggested by Jansson *et al.* [41]. We discuss that our approach can be made adaptive to various other entropy measures.

Next, we focus on ordinal trees, and present a novel universal succinct representation. Our new representation is able to simultaneously *emulate* previous ordinal tree representations of the balanced parenthesis (BP), depth first unary degree sequence (DFUDS) and partitioned representations using a single instance of the data structure. They not only support the union of all the ordinal tree operations supported by these representations, but will also automatically inherit any new operations supported by these representations in the future; hence the *universality* title we attributed to the representation.

We then move to more general graphs rather than trees, and consider the problem of encoding a graph with n vertices and m edges compactly supporting adjacency, neighborhood and degree queries in constant time. The adjacency query asks whether there is an edge between two vertices, the neighborhood query reports the neighbors of a given vertex in

constant time per neighbor, and the degree query reports the number of edges incident to a given vertex. The representation is to achieve the optimal space requirement as a function of n and m to within lower order terms. We prove a lower bound in the cell probe model that it is impossible to achieve the information theoretic lower bound to within lower order terms unless the graph is too sparse (namely $m = o\left(n^{\delta}\right)$ for any constant $\delta > 0$) or too dense (namely $m = \omega\left(n^{2-\delta}\right)$ for any constant $\delta > 0$).

We also present a succinct encoding for graphs for all values of n, m supporting queries in constant time. The space requirement of the representation is always within a multiplicative $1 + \epsilon$ factor of the information-theory lower bound for any constant $\epsilon > 0$. This is the best achievable space bound according to our lower bound where it applies. The space requirement of the representation achieves the information-theory lower bound tightly to within lower order terms when the graph is sparse ($m = o\left(n^{\delta}\right)$ for any constant $\delta > 0$), or very dense ($m = \omega\left(\frac{n^2}{\sqrt{\log n}}\right)$).

Acknowledgements

During my work on the thesis, I had the distinct honor of being supervised by J. Ian Munro whose vast knowledge and insight in the field helped me throughout. I am thankful to Rajeev Raman and S. Srinivasa Rao for their collaboration and great discussions on some results in this thesis. I am also grateful to my thesis committee members Timothy M. Chan, Alejandro López-Ortiz, Robert Sedgewick, and Chaitanya Swamy for their constructive comments.

I would like to also thank Francisco Claude F., Ehsan Chiniforooshan, Reza Dorrigiv, Meng He, Robert Fraser, Mahdi Mirzazadeh, Patrick Nicholson, Alejandro Salinger, and Matthew Skala for great discussions.

Contents

List of tables viii

List of figures ix

1 Introduction 1

2 Preliminaries 4

3 Uniform representation of trees 11

3.1 Introduction . . . 11

3.2 Contribution . . . 13

3.3 Tree Decomposition . . . 14

3.4 Ordinal trees . . . 20

3.4.1 Operations . . . 22

3.5 Cardinal trees . . . 27

3.6 Free trees . . . 29

3.7 Entropy-based succinct encodings . . . 31

3.7.1 Succinct encoding based on degree-distribution entropy . . . 31

3.7.2 Other entropy measures . . . 33

3.8 Summary and discussion . . . 34

4 Universal representations of ordinal trees 35

4.1 Introduction and motivation . . . 35

4.2 Succinct ordinal trees representations . . . 36

4.2.1 Level order unary degree sequence (LOUDS) . . . 36

4.2.2 Balanced parenthesis (BP) . . . 37
4.2.3 Depth first unary degree sequence (DFUDS) . . . 38
4.2.4 Tree covering (TC) . . . 39
4.3 The unified tree representation . . . 39
4.3.1 Reduction to within mini-trees . . . 40
4.4 Supporting representations within a mini-tree . . . 44
4.4.1 Support for the BP and DFUDS encodings . . . 44
4.4.2 Support for the tree covering representation . . . 55
4.5 Summary and discussion . . . 57

5 Succinct representations of graphs **58**
5.1 Introduction . . . 58
5.1.1 Related work . . . 60
5.2 Preliminaries . . . 61
5.3 Space lower bounds . . . 64
5.3.1 Lower bound for the moderate case . . . 64
5.3.2 Lower bound for the sparse case . . . 68
5.4 Upper bound: the representation . . . 71
5.4.1 The almost-full case . . . 72
5.4.2 The extremely-dense case . . . 74
5.4.3 The dense case . . . 75
5.4.4 The moderate case . . . 77
5.4.5 The sparse case . . . 79
5.5 Undirected graphs . . . 81
5.5.1 Space lower bounds for undirected graphs . . . 81
5.5.2 Space upper bounds for undirected graphs . . . 82
5.6 Conclusion and final remarks . . . 84

6 Conclusion **85**

Bibliography **86**

Index **94**

List of Tables

3.1 Space lower bounds up to lower order terms in bits to represent families of trees with n nodes and references to succinct representations. 12
3.2 Comprehensive list of operations on an ordinal tree suggested by [38]. 22

4.1 Main approaches in succinct representations of ordinal trees and the queries they support in constant time: balanced parentheses (BP), depth first unary degree sequence (DFUDS), and tree covering (TC). 37

5.1 Space lower and upper bounds for representing a directed graph with n vertices and m edges which supports the queries in constant time. All the upper bounds are up to lower order terms. 65
5.2 Space lower and upper bounds for representing a directed graph with n vertices and m edges which supports the queries in constant time. All upper bounds are up to lower order terms. 84

List of Figures

3.1 A tree decomposed into component subtrees for value $L = 5$. 15

3.2 Temporary and permanent subtrees on recursive call of node v. Temporary subtrees are indicated by dotted lines and permanent subtrees by solid lines. The leftmost child of v is a branching node and therefore its component is declared permanent; the other two children of v are declared temporary. . . . 16

3.3 A micro-tree root v with its children in different micro-trees. $u_{i_1}, u_{i_2}, \ldots, u_{i_p}$ are indexed in a fully indexable dictionary (FID). 21

3.4 Implementation of operation subtree size. To obtain the subtree size for node v the subtree size of v in microtree μ_1 is read from the look-up table. The subtree size of r within mini-tree m_1 is added to the value and finally the subtree size of r' within the entire value is also added. 25

4.1 An ordinal tree and the corresponding LOUDS, BP, and DFUDS sequences. 38

4.2 Different components of a skinny tree in the new unified representation. (a) A skinny tree (b) The skeleton of the tree along with their immediate children. (c) P_D: unary representation of skeleton degrees to the left. (d) P_U: unary representation of skeleton degrees to the right. (e) T_D: Concatenation of the BP sequences of subtrees to the left. (f) T_U: concatenation of the BP sequences of subtrees to the right. 46

4.3 Four components of a skinny tree representation (P_D, P_U, T_D, and T_U) are given for a skinny tree. 47

4.4 The skeleton of a tree is decomposed into left-leaning and right-leaning paths to partition the tree into skinny trees. 52

4.5 The transition diagram between left-leaning and right-leaning skinny trees going up or down corresponding to a $\frac{\lg n}{16}$-bit subsequence of the BP/DFUDS sequence. 54

5.1 Summary bits are stored for each tiny row and column. 76
5.2 Structures used to represent a small matrix 78

Chapter 1

Introduction

The volume of data we deal with in computer science is increasingly growing. With the ever increasing size of the data sets, an important aspect in handling information is storage requirement. Compression schemes tackle this issue by reducing the space requirement of stored data. The issue, however, with such schemes is slow access and inefficient usage of the data. With most compression schemes, the data must be decompressed almost entirely to answer a query that is only relevant to a small fraction of the data. Auxiliary structural information in the form of pointers to represent trees, graphs, and other combinatorial objects typically claim a significant part of storage requirement. Succinct data structures [47] are an attempt to overcome this shortcoming by accompanying compression with fast access to data and support for operations. Succinct structures are used to encode the auxiliary structural information in small amount of space so it can be used with compression, especially methods that permit partial decompression.

In analyzing the storage requirement of data structures, we often conclude with a formula in "big oh" notation, disregarding the true space cost by hiding the constant factors. These constant factors have an enormous impact in real applications where there is a massive data set to be stored. Succinct data structures tackle this issue by giving an exact highest order term for the space requirement (constant factors are explicitly given). However, little attention is paid to the lower order terms as long as they stay lower order terms as their impact becomes insignificant in practice as the size of data grows. In such applications where the lower order terms become a major player due to the large constant factors they are involved with one extends the definition to the first several high order terms as is studied

in a branch of theoretical literature on succinctness [26, 29, 57].

More precisely, a succinct representation of a combinatorial object is an encoding which supports a reasonable set of operations on the object in constant (or nearly constant) time and has a storage requirement matching the information theory lower bound, to within lower order terms. Succinct data structures perform under the uniform-time word RAM-model with $\Theta(\lg n)$[1] word size where n is the size of the object to be represented; we make the $\Theta(\lg n)$-word RAM model assumption throughout the thesis. Two fundamental data structures which are used ubiquitously in computer science are trees and graphs.

Trees are a key data structure in computer science and as a result a great deal of research has been done on their succinct representation [40, 39, 14, 49, 8, 25, 38, 41, 58, 15]. Succinct representation of two major families of trees have been well studied: ordinal and cardinal trees. Ordinal trees are rooted trees and the order of children is significant and thus must be preserved in any representation. Cardinal or k-ary trees are rooted trees where each node has k slots for children; these slots can be occupied or not occupied independently by a child. We propose a uniform approach for representing trees succinctly that encompasses the two existing families of trees as well as free trees, whose succinct representation had not previously been studied explicitly. "Free" usually means unrooted and unordered. However, the root does not matter much as it requires $O(\lg n)$ bits to distinguish a node as the root. Hence, we treat free trees as rooted trees where the order of children of a node is insignificant. The proposed approach out-performs the numerous previously-existing approaches by being simpler and more intuitive and at the same time allowing fast support for a wider range of operations on the tree (such as parent, child, depth, and height). Furthermore, the representation of a tree in the new approach can simulate two other representations which are widely used for succinct encoding of trees: namely, balanced parenthesis (BP) representation [49] and depth first unary degree sequence (DFUDS) representation [8]. . This simulation power indicates that the new representation can replicate any functionality of the BP and DFUDS representations support.

General graphs—directed or undirected—are another fundamental structure in computer science, as they capture the relations and connectivity between a number of entities. There is a large volume of research on succinct and generally space-efficient representation of graphs [42, 10, 63, 43, 49, 13, 3]. However, most of the research is performed on graphs

[1] $\lg n$ denotes $\log_2 n$.

with certain combinatorial properties such as planar graph, separable graphs, and limited arboricity graphs, . We focus on graphs with no given combinatorial property. We study the problem of representing a graph with n nodes and m edges succinctly to allow the most natural operations in constant time. The operations supported are (1) given a node, output its degree, (2) given two nodes, determine whether there is a direct edge between them, and (3) given a node, iterate through its neighbors and output them in constant time per neighbor. We show lower bounds which are stronger and better than the obvious bounds we obtain via information theory. Moreover, we match these bounds by presenting a succinct encoding of graphs.

The rest of the thesis is organized as follows. Some preliminary data structures which are heavily used in the thesis are presented in chapter 2. Chapter 3 describes the uniform approach we suggest to represent trees succinctly. A preliminary version of this work appeared in [18]. Chapter 4 shows how the proposed approach can simulate two other widely used representations of BP and DFUDS. This result was announced in [20]. Chapter 5 focuses on succinct encoding of graphs. A preliminary version of this work appeared in [17]. Chapter 6 summarizes and concludes the thesis.

Chapter 2

Preliminaries

We use a variety of succinct data structures in designing our representations. In this chapter, we introduce the previous work and results on these data structures:

Bit vectors. We want to store a bit vector $A[1 \ldots n]$ ($A[i] \in \{0, 1\}$) succinctly and allow the prefix sum queries of `rank(r)` and `select(s)`. Query `rank(r)` returns the number of ones in positions with an index at most r (*i.e.*, $\sum_{i=1}^{r} A[i]$) and query `select(s)` returns the index of the s-th one in the array (*i.e.*, *index* (k) s.t. $\sum_{i=1}^{k} = s$).

Jacobson [40] and Clark and Munro [14] were the first to give a representation with space $n + o(n)$ bits and constant query time. Although space $n + o(n)$ is the best one can obtain for a dense bit vector (in which about half of the number of bits are set), better bounds are expected in sparse bit vectors. Bit vectors with more one than zero bits, are clearly symmetrical to the those with more zeros than ones by switching ones and zeros. These bounds are indeed sensitive to the number of ones in the bit vector. We overview these results in "succinct dictionary" structures.

Dictionaries. The standard definition of the dictionary problem is to store a set S of n elements from a universe U of u elements, and answer membership queries ($x \overset{?}{\in} S$) efficiently. In our context, an easy solution is to store a bit vector of length u containing n ones, though clearly, the optimal space is $\mathcal{B} = \left\lceil \lg \binom{u}{n} \right\rceil$ bits which can be much less than u bits when $n \ll u$ (or symmetrically when $n > u/2$ and is very close to u). The FKS dictionary [24] uses $O(\mathcal{B})$ bits and supports membership queries in constant time. Brodnik and Munro [11] first studied

the dictionary problem from the perspective of succinct structures. Their dictionary uses $\mathcal{B}+O\left(\mathcal{B}/\log\log\log u\right)$ bits and answers membership queries in constant time on a $\lg u$-bit word. Pagh [56] improved the space redundancy to obtain space $\mathcal{B}+O\left(n\frac{(\log\log n)^2}{\log n}+\log\log u\right)$ while still answering membership queries in constant time.

Other than membership queries, a dictionary is sometimes desired to support other operations, for instance `rank` and `select`. These operations "index" elements present in the subset. Given an element x, query `rank`(x) returns the number of elements i in S such that $i \le x$, and hence the "rank" of x among elements of S. The reverse operation of `Select`(j), returns the j-th smallest element in S (in other words, the element whose rank is j). `rank` and `select` operations are equivalent to `rank` and `select` queries we defined over bit vector for the bit vector with length u that realizes elements of S. We note that constant-time support for `rank` implies constant-time support for membership queries. Depending on whether we want to answer `rank` and `select` queries on members only or on both members and non-members of S, we distinguish two dictionary problems of *fully indexable* dictionary or only *indexable* dictionary.[1]

More formally, a fully indexable dictionary supports `rank` and `select` operations on both elements of S and non-elements of $U \setminus S$ (and hence membership queries). In other words, given any element x of the universe U, we support `rank(`x`)` in constant time whether $x \in S$ or not. We note that a fully indexable dictionary is a powerful structure as, for example, it can answer predecessor queries in constant time by computing `select(rank(`i`))`. However, we know that predecessor queries cannot be answered in constant time on the RAM model with $n^{O(1)}$ words of space [7] (for "awkward" ranges of n as a function of u). Therefore, any fully indexable structure is bound to be relatively space-inefficient. Raman *et al.* [58] gave a representation of a fully indexable dictionary:

Lemma 2.1. *[58] Given a subset S of size n from a universe U of size u, there is a* fully indexable dictionary **(FID)** *structure which requires* $\lg\binom{u}{n}+O\left(u\log\log u/\log u\right)$ *bits and supports rank and select queries both on members and non-members of S in constant time in a* $\lg u$*-bit word-RAM model.*

A series of improvements [29, 31] on the redundancy of the space in the lemma culminated in overall space of $\lg\binom{u}{n}+\frac{u}{\text{polylog}\, u}+\tilde{O}\left(u^{3/4}\right)$ [57]. Nevertheless, we use the result as stated in

[1] Adopting the terminology of Raman *et al.* [58].

the lemma for simplicity and since we do not focus on redundancy reduction in lower order terms in this thesis.

An indexable dictionary supports membership, `rank`, and `select` operations, however the latter two are supported only on members of S. In other words, given an element $x \in U$, we support `rank(`x`)` only if $x \in S$. If $x \notin S$ then, the query only has to report that $x \notin S$. This feature significantly reduces the power of this type of dictionaries as, for example, we can no longer support predecessor queries. Therefore, this type of dictionaries are more space efficient. Raman *et al.* [58] also gave a representation of an indexable dictionary:

Lemma 2.2. *[58] Given a subset S of size n from a universe U of size u, there is an* indexable dictionary **(ID)** *structure which requires* $\lg \binom{u}{n} + o(n) + O(\log \log u)$ *bits and supports rank and select queries only on members of S in constant time in a* $\lg u$*-bit word-RAM model.*

We heavily use the FID and ID structures of lemmas 2.1 and 2.2 in designing succinct structures for trees and graphs.

Trees. Trees are one of the most fundamental data structures in computer science and as a result, they have been explored extensively from the perspective of succint structures. Various classes of trees arise in practice and each class has been considered individually for the purpose of succinct representation. We briefly survey these classes here. For a detailed discussion of each of these classes, we refer the reader to chapter 3.

Binary trees, where nodes can have a left and/or a right child, were among the first that were considered. Jacobson [39] gave the first succinct representation; each (internal) node is marked with a 1 bit and the external nodes are added to the tree and are marked with a 0 bit. A bit vector is formed by traversing the tree in a left-right level-order fashion and reading off the marked bits associated with the nodes. The length of the bit vector for a binary tree with n nodes is $2n+1$ bits. By storing this bit vector as a dictionary structure that supports `rank` and `select`, one can support operations `parent`, `left-child`, and `right-child` in constant time.

Ordinal trees (also known as ordered trees) are rooted trees where nodes can have high degrees and children of a node are ordered. There is a one-to-one correspondence between ordinal trees with n nodes and binary trees with n nodes. Therefore, $2n$ bits suffice to encode an ordinal tree with n nodes. There are various approaches towards succinct representations

of ordinal trees: level order unary degree sequence (LOUDS) approach [39], balanced parenthesis (BP) approach [50], depth first unary degree sequence (DFUDS) approach [8], and tree covering (TC) approach [25]. There has been many improvements, particularly on operation support capability of these representations. We refer the reader to chapter 4 for a detailed discussion.

Cardinal trees or k-ary trees are trees where nodes have k slots for edges to children and these slots can be independently occupied or not. Binary trees are special cardinal trees where $k = 2$. First succinct representation [8] of cardinal tree consisted of two components. The first component was the representation of the underlying ordinal tree (using the DFUDS approach) and the second component was essentially a succinct dictionary structure to represent which children of nodes are indeed present. The storage requirement of this approach became very close to the minimum required space, however there was a gap. Later work used the indexable dictionary structures (in lemma 2.2) to improve the space to the minimum to within lower order terms [58]. We refer the reader to chapter 3 for a more detailed discussion.

In chapter 3, we consider free trees which are unordered rooted trees and also discuss entropy-based encodings of trees. Representing dynamic trees succinctly is another related line of work [52, 59, 35, 19].

Permutations. Given a permutation $\pi : [n] \rightarrow [n]$, we consider the problem of succinctly representing π such that arbitrary powers of π are computed in constant time. That is, given an integer k, the representation must support computing $\pi^k(\cdot)$ in constant time. We note that k can be a negative integer and in fact, having a minimum-space representation that computes both $\pi(\cdot)$ and $\pi^{-1}(\cdot)$ in constant time is already a hard problem.

The naïve representation of a permutation is by listing $\pi(1), \pi(2), \ldots, \pi(n)$ in order. Although, this method achieves the optimal space of $n \lceil \lg n \rceil$ bits, it supports only $\pi(.)$ operation in constant time and for example, $\pi^{-1}(.)$ has no efficient implementation.

Munro *et al.* [48] give a representation of permutations which supports both $\pi(\cdot)$ and $\pi^{-1}(\cdot)$ in constant time, however the storage requirement is $(1 + \epsilon)n \lg n$ bits for any fixed epsilon $\epsilon > 0$. The representation is based on the cycle representation of permutations and storing explicit back edges at every $O(1/\epsilon)$ step of the cycle. Moreover, they show that once there is constant-time support for $\pi(.)$ and $\pi^{-1}(\cdot)$ operations, arbitrary powers of π ($\pi^k(.)$ for any integer k) can be supported in constant time.

Golynski *et al.* [27, 28] prove that the constant $1+\epsilon$ multiplicative factor for the storage requirement is indispensable as it is not possible to have an arbitrary permutation in $n \lg n + o(n \log n)$ bits and support both $\pi(.)$ and $\pi^{-1}(.)$ in constant time. The proof is in the cell probe model with cells of length $O(\log n)$ bits.

Functions. Given a function $f : [n] \rightarrow [m]$, we consider the problem of succinctly representing f such that arbitrary positive powers of f, (*i.e.,* $f^k(x),\ k > 0$), can be computed in constant time and arbitrary negative powers of f, (*i.e.,* $f^{-k}(x),\ k > 0$), can be computed in $O\left(1 + \left|f^{-k}(x)\right|\right)$ time (we note that f^{-k} is a set when f is not one-to-one). Succinct representation of functions is an extension to that of permutations.

Munro *et al.* [53] first consider representing functions with the same domain and range size (*i.e.,* $n = m$) and give a $(1+\epsilon)n \lg n$-bit representation for any constant $\epsilon > 0$. Where $n \geq m$, they give a representation with $n \lg m + \epsilon m \lg m$ bits, and where $n < m$, their representation has $n \lg m + (1+\epsilon)n \lg n$ bits.

We present a new succinct representation of functions in section 5.2 with some added capabilities.

Strings. Given a string of length n over alphabet $\Sigma = [\sigma]$, we consider the problem of encoding the string in minimum space such that the following operations are supported efficiently:

- `string-access`(i): returns the character at position i.
- `string-rank`(c, i): returns the number of occurrences of character c up to position i.
- `string-select`(c, i): returns the position of the i-th occurrence of character c in the string.

The wavelet tree [33] structure can encode a string in $n \lg \sigma + o(n) \lg \sigma$ bits (more precisely in $nH_0 + o(n) \lg \sigma)$ bits where H_0 is the zeroth order entropy, and do all operation of `string-access`, `string-rank`, and `string-select` in $O(\log \sigma)$ time. For larger alphabets, there is a representation [30] that encodes a string in $n \lg \sigma + o(n) \lg \sigma)$ bits (and is not adaptive to the zeroth order entropy), however performs all the operations in $O(\log \log \sigma)$ time.

Binary relations. Binary relations associate r objects to s labels such that an object can be associated to multiple labels and a label can be assigned to multiple objects. We denote the number of object-pair associations by t. Therefore, one can abstract a binary relation as a 0-1 $r \times s$ matrix where position (i, j) is one if object i is associated with label j and the position is zero otherwise. Hence, the matrix contains precisely t ones.

The problem we are considering is, given a $r \times s$ binary relation with t object-label pairs, to encode it in the minimum space such that the following operations are performed fast:

- `object-access`(o, l): Determines whether object o is associated with label l. This operation corresponds to retrieving position (o, l) of the corresponding 0-1 matrix.

- `object-rank`(o, l): Reports the number of labels up to l that are associated with object o. This operation corresponds to operation `rank`(l) in row number o of the corresponding 0-1 matrix.

- `object-select`(o, i): Reports i-th label in order that is associated with object o. This operation corresponds to operation `select`(l) in row number o of the corresponding 0-1 matrix..

- `label-access`(l, o): Determines whether label l is associated with object o. This operation corresponds to retrieving position (o, l) of the corresponding 0-1 matrix.

- `label-rank`(l, o): Reports the number of object up to o that are associated with label l. This operation corresponds to operation `rank`(o) in column number o of the corresponding 0-1 matrix.

- `label-select`(l, i): Reports i-th object in order that is associated with label l. This operation corresponds to operation `select`(l) in column number l of the corresponding 0-1 matrix.

Strings are binary relations where an object is associated with exactly one label. The problem of succinct representation of binary relations is a generalization of that of strings. Barbay *et al.* [5] give a representation that encodes such a binary relation in $t \lg \sigma + o(\lg \sigma)$ bits and supports all these operations in worst case $O(\log\log \sigma \log\log\log \sigma)$ time.

Graph representations. Given a graph as an abstract data type, one requires a representation that occupies the minimum space (to within lower order terms) and supports expected navigational queries in ideally constant time.

There is a large body of literature on space-efficient representations of graphs most of which deal with graphs with particular properties such as: graphs with limited arboricity, c-decomposable graphs [42], separable graphs [10], planar graphs [63, 43, 49, 13], triconnected and/or triangulated planar graphs [3]. Mainly planar graphs and their subclasses such as triconnected planar graphs and triangulated planar graphs have been considered from the perspective of succinct representations [49, 13, 3].

In contrast to this body of work, in chapter 5, we deal with graphs with no particular combinatorial properties, and of which we only know the number of vertices and edges.

Text indexes. A text index is a structure to represent a text T such that given a pattern P we can perform string matching queries (finding occurrences of P in T) fast.

Suffix trees and suffix arrays are two common text indexes which are widely used in practice. Therefore, there is work on space-efficient and succinct representation of suffix trees [51, 34] and suffix arrays [34, 60]. Ferragina and Manzini [21] introduce an *opportunistic* data structure for indexing a text with a storage requirement which is adaptive to the k-th order entropy of the text. This high order entropy adaptiveness is continued in a line of work [62, 33, 22].

Chapter 3

Uniform representation of trees

In this chapter, we describe a new approach for succinct representation of trees. This approach is a simple one with which many classes of trees can be represented succinctly within their particular space bounds. The power of this approach is that it consolidates representations of various families of trees for which there are different approaches for succinct representations. We consider the class of "free trees" for the first time for the purpose of succinct representation. We show our scheme can be easily adapted for succinct encodings of this family. Furthermore, for different families of trees, the new approach improves on the existing approaches by adding some extra functionalities or easing their implementation which were non-existent in previous ad-hoc approaches.

3.1 Introduction

Succinct representations of two major families of trees have been well studied: *ordinal trees* and *cardinal trees*. In ordinal trees, the order of children of nodes is significant and preserved. However, in cardinal trees (also known as k-ary trees), each node has k slots for edges to children which can be independently occupied or not. Binary trees are a subclass of cardinal trees for value $k = 2$. In this chapter, we also study the succinct encoding of another family of trees: *free trees* (not previously studied in the content of succinct data structures) in which the order of children of a node is not significant.

Certain subfamilies of these major families of trees have been studied in the context of succinct representations. Binary trees ($k = 2$) [49] and DNA trees ($k = 4$) [8] form two of

Tree family	Space lower bound (Highest order term)	Succinct representation
Ordinal trees	$2n$ [44]	[40, 14, 49, 8, 25, 38]
Ordinal trees with a given degree distribution	$\sum_i n_i \lg \frac{n}{n_i}$ [61]	[41]
Cardinal trees	$(k \lg k - (k-1)\lg(k-1))\, n$ [32]	[8, 58]
Binary trees	$2n$ [32]	[40, 39, 15, 14, 49]
Free trees	$1.56 \cdots n$ [55]	chapter 3
Free binary trees	$1.31 \cdots n$ [16, 64]	chapter 3

Table 3.1: Space lower bounds up to lower order terms in bits to represent families of trees with n nodes and references to succinct representations.

the best known subfamilies of cardinal trees. Ordinal trees with a given degree distribution where a list of numbers n_i $(i \geq 0)$ is given and the tree is guaranteed to have exactly n_i nodes with i children form a subfamily of ordinal trees studied recently by Jansson *et al.* [41]. We also investigate *free binary trees* which are free trees, with maximum two children per node.

Space lower bounds on the required number of bits to represent each class of trees is obtained via information theory by counting the number of trees in the class. For instance, the number of binary trees with n nodes is known [32] to be the n-th Catalan number $C_n = \frac{1}{n+1}\binom{2n}{n}$. Hence, an encoding of binary trees with n nodes requires at least $\lg C_n = 2n - \lg n + O(1)$ number of bits. As another example, the number of ordinal trees with n nodes is also C_n since there is a one to one correspondence between ordinal trees and binary trees with the same number of nodes; therefore, $\lg C_n = 2n - \lg n + O(1)$ is the minimum storage requirement for ordinal trees. The number of cardinal (k-ary) trees with n nodes is the generalized Catalan number $C(n,k) = \frac{1}{kn+1}\binom{kn+1}{n}$, and therefore the minimum number of bits required to represent such a tree is $\lg C(n,k) = (k \lg k - (k-1)\lg(k-1))\, n - O(\log kn)$. Table 3.1 illustrates the space lower bounds for these classes along with existing references to succinct representations which achieve the optimal space to within lower order terms and support a variety of operations.

3.2 Contribution

We propose a uniform approach for representing trees succinctly that encompasses the families of trees in table 3.1. The method is based on a two-level decomposition of a tree into subtrees. The recursive decomposition method is a common technique in succinct representations of various data structures [14, 58, 6] and has been used to represent trees [52, 25, 38].

In the case of ordinal trees, the most comprehensive list of operations supported is achieved by He *et al.* [38]. The new approach supports the entire range of operations proposed by He *et al.* [38] and simplifies implementation of the supported operations. In the case of cardinal trees, there had been no known succinct representation that supports a wide range of navigational operations. Raman *et al.* [58] state that their succinct representation for cardinal trees cannot support subtree size. Our succinct representation of cardinal trees can support all ordinal-tree-type operations listed by He *et al.* [38] (such as subtree size) as well as cardinal-tree-type operations suggested by Raman *et al.* [58] (such as following the edge labeled i from a node where $1 \leq i \leq k$).

To show the power of our method, we consider free trees and show that we can have a succinct representation taking the optimal $(1.56\ldots)n$ bits (ignoring lower order terms) supporting all navigational operations. Similarly, free binary trees, which are free trees with maximum two children per node, can be represented in the optimal $(1.31\ldots)n$ number of bits.

Existing succinct encodings of trees assume a uniform distribution over the family of trees and therefore give worst case space guarantees. In practice however, there might be many reasons to have trees with certain property that are more likely than others, and therefore an entropy-based succinct representation is necessary. Jansson *et al.* [41] considered this case where the distribution is based on degrees of nodes and gave a representation that matches the degree-distribution entropy. Our succinct tree representation can not only match the degree-distribution entropy, but also can be made adaptive to a variety of other entropy measures, *e.g.,* trees with a particular probability distribution of number of children (a node has i children with probability p_i).

3.3 Tree Decomposition

At the heart of our method is the tree decomposition technique. The aim is to decompose the tree into subtrees of roughly equal sizes. Geary *et al.* [25] and He *et al.* [38] use the same decomposition algorithm to decompose an ordinal tree into smaller trees. Their decomposition algorithm, when run on a binary tree, matches the decomposition algorithm of Munro *et al.* [52] which is given for binary trees. Given a target subtree size L, the algorithm of Geary *et al.* decomposes a tree into subtrees with size between L and $3L$ (with a possible exception of the root subtree). Furthermore, these subtrees are disjoint other than their roots; many subtrees can share a common root node.

The drawback with this approach is that numerous edges can leave a node in a subtree to a node in another subtree, and these edges can stem from any node in a subtree. Our decomposition algorithm has the feature that all edges leaving a subtree stem from the root of the subtree, except perhaps for one single edge which can be incident to a non-root node. We guarantee this by allowing (a small number of) undersized subtrees.

Fredrickson [23] introduces a topological partitioning of the vertex set of trees which is suitable for our application. This partitioning, unfortunately, works for (unrooted) trees of degree at most three.

Theorem 3.1. *A tree with n nodes can be decomposed into $\Theta(n/L)$ subtrees of size at most $2L$ which are pairwise disjoint aside from the subtree roots. Furthermore, aside from edges stemming from the component root nodes, there is at most one edge leaving a node of a component to its child in another component.*

Proof. Figure 3.1 depicts the result of the decomposition algorithm run on a tree. We start the proof by considering the nodes that have many descendants:

Definition 3.1. *For a fixed parameter L, a node is* **heavy** *if it has at least L descendants (including the node itself). Ancestors of a heavy node are heavy by definition. Therefore, heavy nodes form a subtree on the original tree. We call this tree the* **heavy-subtree***. A* **branching node** *is a node which has at least two heavy children.* **Branching edges** *are edges between a branching node and its heavy children.*

For instance, in the tree of figure 3.1, heavy nodes are a, b, d, k, o, q, r. Branching nodes are a, o and branching edges are ab, ad, oq, or.

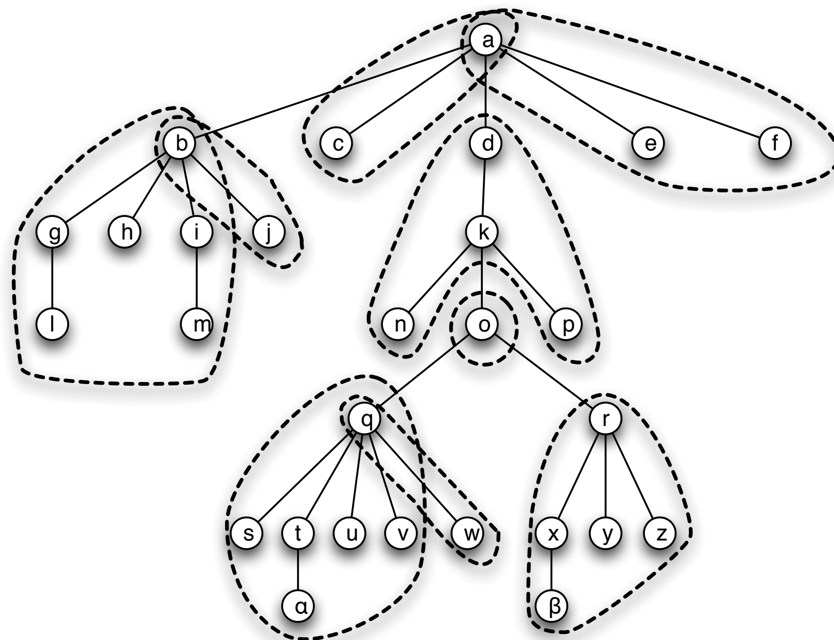


Figure 3.1: A tree decomposed into component subtrees for value $L = 5$.

A crucial observation is that the number of branching edges is bounded. In the heavy-subtree of tree T, the number of branching edges is exactly twice the the number of leaves of the heavy-subtree minus two. The number of leaves is at most n/L as each heavy-tree leaf is associated with at least L distinct nodes of the original tree. Clearly, the number of branching nodes is less than the number of branching edges. Thus:

Lemma 3.2. *The number of branching nodes and edges in a tree T with n nodes and parameter L is $O(n/L)$.* □

As with previous decomposition methods [14, 52, 25], we use a recursive bottom-up approach. Each recursive call decomposes a tree rooted at a node and returns the component subtrees. To decompose a tree rooted at a node v, we first recursively decompose the trees rooted at its children $u_1, \ldots, u_k$. Component subtrees that do not contain any of the children $u_1, \ldots, u_k$ of v are *permanent* and remain invariant through recursive calls on other nodes of the tree. The root components that contain one of $u_1, \ldots, u_k$ can initially be declared as *temporary*. Temporary components and the parent v can potentially merge. Figure 3.2 depicts existing temporary and permanent subtrees on the recursive call of a node.

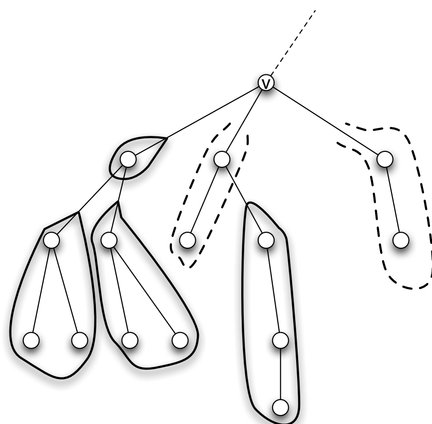


Figure 3.2: Temporary and permanent subtrees on recursive call of node v. Temporary subtrees are indicated by dotted lines and permanent subtrees by solid lines. The leftmost child of v is a branching node and therefore its component is declared permanent; the other two children of v are declared temporary.

The merging procedure of the temporary components at a recursive call on node v, depends on the number of heavy children of the node. The procedure is depicted in algorithm 3.2 and is as follows:

1. If v has no heavy children (*e.g.,* node b, q in figure 3.1), entire children subtrees containing $u_1, \ldots, u_k$ are temporary components to be merged together. The procedure is a greedy approach depicted in algorithm 3.1. We create a new component initially containing only v. We scan the list of children $u_1, \ldots, u_k$ from left to right adding the entire tree rooted at these nodes to the component. If the component size exceeds L, we finalize that component by declaring it as permanent and create a new component containing v only and continue in this manner. Since none of the children is heavy, the size of components does not exceed $2L$. The last such component can have size less than L. If we had created at least one other component aside from the last component, we declare the last component as permanent, even though its size is small, and we charge its small size to its neighbor component which must have a proper size. Otherwise, if there is only one component, we have put all descendants of v together in a component which we declare as temporary and send up to the parent of v.

2. If v has only one heavy child u_i (*e.g.,* nodes d, k in figure 3.1 as n is heavy), we put

children of v into components analogously to the previous case. The only difference occurs where the component containing u_i has been declared permanent as opposed to temporary. In this case, we simply ignore u_i, skipping from u_{i-1} to u_{i+1} during the scan. Analogous to the previous case, if more than one component are formed, all are declared as permanent. If only one component is formed, if its size is less than L, it is declared as temporary, otherwise as permanent.

3. If v is a branching node—*i.e.,* with two or more heavy children (*e.g.,* nodes a, o in figure 3.1)—then among children $u_1, \ldots, u_k$, there are $h \geq 2$ heavy nodes $u_{i_1}, \ldots, u_{i_h}$. We first declare permanent the components containing these heavy nodes. If the component containing u_{i_j} for some j is undersized, we charge it to the branching edge vu_{i_j}.

   If there is no non-heavy children left, v by itself is a permanent single-node component (we charge this undersized component to branching node v itself). Otherwise, the remaining children of v are broken by the heavy nodes into intervals of consecutive non-heavy nodes. We consider the intervals separately, treating each interval as the first case (no-heavy-children case). The only difference is that all components formed are finalized and declared as permanent without exception. We charge the possible undersized component at the end of each such interval l to one of the interval's end edges $vu_{i_{l-1}}$ or vu_{i_l} (note that $vu_{i_{l-1}}, vu_{i_l}$ are both branching edges).

The decomposition algorithm creates components with a set of properties which we include in the following lemma:

We now prove that the decomposition of algorithm 3.2 satisfies the desired properties of theorem 3.1.

Firstly, it is easy to verify that no component contains more than $2L$ nodes. Components of non-constant size are only constructed by "GreedilyPack" algorithm (algorithm 3.1). This procedure is called only on temporary components and these components have size less than L. Since these components are combined one by one so the aggregate size exceeds L, the maximum size can be $2L$.

Secondly, it is also easy to argue that two different components can only share a node that is a root of both components. The argument is recursive; if node v is placed into more than one component in algorithm 3.2, these components are declared as permanent and therefore

Algorithm 3.1 Greedily packs components of children $u_1, \ldots, u_k$ of node v together.

Procedure GreedilyPack($v, u_1, \ldots, u_k$):
/ $c_1, \ldots, c_k$ are the components to which $u_1, \ldots, u_k$ respectively belong. */*
$\mathcal{C} \leftarrow \{c_1, \ldots, c_k\}$.
while $\mathcal{C} \neq \emptyset$ **do**
  $s \leftarrow \min \text{index}_i \; c_i \in \mathcal{C}$.
  $m \leftarrow \min \text{index}_j \; \text{size}\,(c_{s+1} \cup \ldots \cup c_j) \geq L$.
  $\mathcal{C} \leftarrow \mathcal{C} - (c_s, c_{s+1}, \ldots, c_m)$.
  Create a new component $c = \{v\} \cup (\{c_s\} \cup \{c_{s+1}\} \ldots \cup \{c_m\})$.
  / c is declared permanent unless it is the only component created and it is undersized: */*
  if size(c)$< L$ and c contains all of $u_1, \ldots, u_k$ **then**
    Declare c as "temporary".
  else
    Declare c as "permanent".
  end if
end while

v is root in all these components.

Thirdly, we argue on the pattern of inter-component edges, and show that other than the edges that stem out of the root of a component, there is at most a single edge leaving a node to a child in another component. We argue about this property recursively. Given that all children components of node v have this property, we demonstrate that the manner in which we combine these components in algorithm 3.2 maintains the property. If v has no heavy children then entire children subtrees are packed together which means no node other than roots have edges leaving a component. In case v has only one heavy child u, components involving other children of v contain an entire subtree and therefore have no outgoing edges. If u belongs to a temporary component (before the call to GreedilyPack), it must belong to only one component c which can have one outgoing edge from a node other than u. Therefore, in procedure GreedilyPack when combined with other children components of v can result in a component which has at most one other outgoing edge. If, however, u is in a permanent component, in algorithm 3.2 it is not combined with other children components and therefore one of the components including v has a single outgoing edge (v, u).

Finally, to bound the number of components, one only has to account for the undersized components. During the decomposition, we charged undersized components to components with the proper size (size larger than L) in cases of nodes with at most one heavy child or to

Algorithm 3.2 Recursive decomposition of the subtree rooted at node v with children $u_1, \ldots, u_k$ into component subtrees.

```
Procedure Decompose(v):
/* Recursive calls:*/
if v is a leaf then
    return {v} as a temporary component;
else
    run Decompose(u_i) for all children u_1, ..., u_k of v.
end if

/* Combining components of children u_1, ..., u_k of v */
if v has no heavy children then
    GreedilyPack(v, u_1, ..., u_k)
else if v has only one heavy child u_i then
    if u_i belongs to a temporary component then
        GreedilyPack(v, u_1, ..., u_k)
    else
        GreedilyPack(v, u_1, ..., u_{i-1}, u_{i+1}, ..., u_k)
    end if
else if v has two or more heavy children u_{i_1}, ..., u_{i_h} then
    /* set u_{i_0} ← 0, u_{i_{h+1}} ← k */
    for j = 0, ..., h do
        GreedilyPack(v, u_{i_j+1}, ..., u_{i_{j+1}-1})
        Declare all returned component subtrees as "permanent".
        Declare component of u_{i_j} as "permanent".
    end for
    If all children of v are heavy, declare {v} by itself as a "permanent" component.
end if
```

branching edges or branching nodes in cases of nodes with at least two heavy nodes. Each properly-sized component is charged at most once and each branching edge can be charged at most three times (from left or right adjacent intervals and from the component below), and each branching node is charged at most once. The number of properly-sized components is $O(n/L)$ and the number of branching edges/nodes is $O(n/L)$ by lemma 3.2. Therefore, the number of undersized components is $O(n/L)$ and thus the total number of components is $\Theta(n/L)$. □

3.4 Ordinal trees

In this section, we outline our succinct representation for ordinal trees. The representation is analogous to that of Munro *et al.* [52] and of Geary *et al.* [25] in that it is a two-level recursive decomposition of a given tree. In the first level of recursion, the tree with n nodes is first decomposed into subtrees using value $L = \lceil \lg^2 n \rceil$, and subsequently these subtrees are, in turn, decomposed into yet smaller subtrees using value $L = \lceil \frac{\lg n}{4} \rceil$ to obtain the subtrees on the second level of recursion. Using the terminology of Geary *et al.* [25], we refer to as the subtrees on the first level by *mini-trees* and the second level by *micro-trees*.

Micro-trees which have size less than $\lceil \frac{\lg n}{2} \rceil$ are small enough to be represented by a look-up table. The representation of a micro-tree with $k < \lceil \frac{\lg n}{2} \rceil$ nodes consists of two fields: the first field simply is the size of the micro-tree ($O(\log k) = O(\lg \lg n)$ bits) and the second field is an index to the look-up table ($2k$ bits). These indices sum up to $2n$ bits over all micro trees and are the dominant term in our representation; other auxiliary data amounts to $o(n)$ bits.

The table stores encodings of all trees with sizes up to $\lceil \frac{\lg n}{2} \rceil$ along with answers to variety of types of queries for each of those trees. One can use any space-efficient encoding of trees for this purpose (enumeration code, balanced parenthesis, *etc.*). The auxiliary data we keep together with each tree helps us answer queries and will be described when we show how to perform queries. The size of the auxiliary data for each tree will be poly-logarithmic in n and thus the size of the entire table is $o(n)$.

Mini-trees consist of micro-trees and links between them. Links between different micro-trees can be either in form of a common root node or an edge from a non-root node from a micro-tree to the root of another micro-tree. We represent these edges by introducing a *dummy node* on them; if there is an edge from a non-root node v of a micro-tree M_1 to the root node r of another micro-tree M_2, we introduce dummy node d and subdivide the edge vr into edges vd and dr. Edge vd becomes part of the micro-tree M_1 and therefore is accounted for by the micro-tree representation. We refer to edges with a dummy parent such as dr as *dummy edges*. We keep an explicit pointer to represent edge dr. Due to the manner in which the tree is decomposed, there is at most one such edge leaving a non-root node in a micro-tree (theorem 3.1), therefore the number of these edges is $O(\log n)$ in any mini-tree. Hence, the

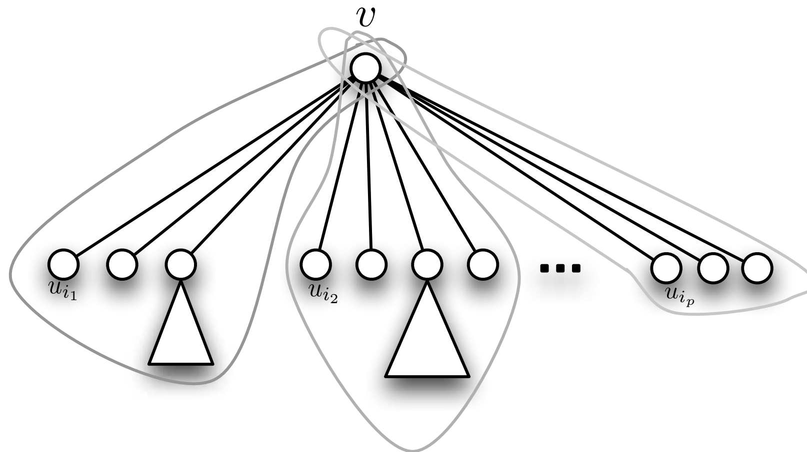


Figure 3.3: A micro-tree root v with its children in different micro-trees. $u_{i_1}, u_{i_2}, \ldots, u_{i_p}$ are indexed in a fully indexable dictionary (FID).

dummy nodes contribute $O\left(\log^2 n / \log n\right) = O(\log n)$ nodes to a mini-tree which amounts to a lower order term. Moreover, we can afford to keep explicit pointers (of size $O(\log \log n)$) to represent dummy edges as they contribute $O\left(\log \log n \log^2 n / \log n\right) = O(\log n \log \log n)$ bits per mini-tree.

To represent the common roots among micro-trees, we use the fully indexable dictionary (FID) of lemma 2.1 which essentially indexes and represents a subset of a universe. Given a root node v with children $u_1, \ldots, u_k$ in p different micro-trees, if $i_1, \ldots, i_p$ are the indices of children that belong to a different micro-tree than their immediate left siblings (as depicted in figure 3.3), we form set $I = \{i_1, i_2, \ldots, i_p\}$ over the universe of $[k]$. We represent this set as a FID and thus we can navigate on children of v. The required space for this FID is $\lg \binom{k}{p} + O(k \log \log k / \log k)$ bits by lemma 2.1. We argue this only adds up to a lower order term. To account for the first additive term, it suffices to observe $\lg \binom{k}{p} < p \lg k = O(p \log \log n)$. Therefore, we can charge $\log \log n$ to every micro-tree involved for the space. Considering the summation of the second additive terms over all root nodes, one can verify that since there are $O(\log n)$ micro-trees and therefore root nodes and by concavity of function $f(x) = x \lg \lg x / \lg x$, the sum is $O(n \log \log \log n / \log \log n) = o(n)$.

The tree consists of mini-trees and links between them. The tree over mini-trees is represented analogously to the manner a mini-tree is represented over micro-trees: *i.e.,* explicit pointers for edges coming out of mini-trees from non-root nodes and a FID to represent

Operations	Definition
child(v,i), child_rank(v)	i^{th} child of node v, Number of left siblings of node v
degree(v), subtree_size(v)	Number of children of v, Number of descendants of v
depth(v), height(v)	The depth/height of node v
leftmost(rightmost)_leaf(v), leaf_size(v)	v's leftmost/rightmost descendant leaf , number of descendant leaves
leaf_rank(v), leaf_select(v)	number of leaves before v in preorder, i^{th} leaf of the tree in preorder
node_rank$_{\text{pre}}(i)$, node_select$_{\text{pre}}(v)$	position of v in preorder, i^{th} node in pre order
node_rank$_{\text{post}}(i)$, node_select$_{\text{post}}(v)$	position of v in post order, i^{th} node in post order
level_anc(v, i), LCA(x, y), distance(x, y)	ancestor of v at level i, lowest common ancestor and distance of x, y
level_left/rightmost(i), level_succ/pred(v)	left/right most node at level i, successor or predecessor of v on its level

Table 3.2: Comprehensive list of operations on an ordinal tree suggested by [38].

edges out of a common mini-tree root. The space analysis is the same and shows that the auxiliary data amounts to $o(n)$.

3.4.1 Operations

In this section we show that various operations on ordinal trees can be implemented in a straightforward and, more importantly, a uniform manner using our representation. Table 3.2 defines a comprehensive list of operations suggested by He *et al.* [38] for ordinal trees.

Most operations become much easier to support compared to implementations of Geary *et al.* [25], and He *et al.* [38]. Generally, the technique is as follows: We store some relevant data in the look-up table. Within a mini-tree, we keep the necessary information at the roots of the micro-trees to answer the query in the mini-tree. Finally, we keep the answers to the queries at the roots of the mini-trees. Using these three categories of data and using the fact that there is at most one edge going out of a non-root node in a mini/micro-tree, we can answer various types of queries easily. We explain exactly how the mentioned operations can be supported in constant time with our representation.

Dummy ancestor within mini-tree: This operation is to check whether an arbitrary given node v in a mini-tree is an ancestor of the dummy node d belonging to the same mini-tree. This operation is a special case of a general ancestor checking operation where we determine, given two arbitrary nodes i, j, if i is an ancestor of j. Ancestor operation is itself

a special case of lowest common ancestor operation. Hence, dummy-ancestor is not listed as an operation in table 3.2. However, since other operations rely heavily on this operation and there is a more straightforward way to support this operation, we give an implementation of this operation in constant time.

Given a node v within a mini-tree, we must answer whether v is an ancestor of the dummy node d of the mini-tree. If v and d belong to the same micro-tree then one can use the look-up table to retrieve the answer in constant time. Otherwise, we retrieve the dummy node d_1 of the micro-tree of v (if exists), and check whether d_1 is an ancestor of d. In order to perform the last step, we explicitly store whether each micro-tree dummy node is an ancestor of the dummy node of the encapsulating mini-tree. The stored information requires only $O(\log n)$ bits per mini-tree and therefore contributes to $o(n)$ to the storage requirement of the entire representation.

Child and child rank: To determine the ith child of v, we narrow down to a particular mini-tree and then a micro-tree. In order to narrow down to a mini-tree, if v is a mini-tree root, we use the FID to find the mini-tree the ith child of v belongs to, otherwise if v is not a mini-tree root, the child is in the same mini-tree as v (it could be a dummy node). If the child is a dummy node, we follow the stored explicit pointer to the root of the following micro-tree. Analogously, once we know the mini-tree, we narrow down to a micro-tree. In a micro-tree we use the look-up table to find the child and report it.

To compute child rank, which is the number of left siblings of a node v, we find the parent p of v first. If p is a mini-tree and/or micro-tree root, then we use the corresponding FIDs to compute the child rank of v, otherwise we use the look-up table to do this.

Degree and subtree size: If the given node v is not a micro-tree root then we use the look-up table to find the degree. Otherwise, if it is a micro-tree and/or mini-tree root we use the relevant FIDs to compute the degree.

To compute the subtree size for node v, we explicitly store the subtree size at mini-tree roots and we store the subtree size at micro-tree roots within mini-trees, and the look-up table contains the subtree size within a micro-tree for each node of the micro-tree.

If v is a mini-tree root then the value is explicitly stored. Otherwise, if v is a micro-tree root, we have the subtree size stored within the mini-tree; we add to this value if v is an

ancestor of other mini-trees. We determine if v is an ancestor of the dummy node of the mini-tree (using the support for the dummy ancestor operation) and if so we add the subtree size value of the mini-tree root that the dummy node goes to. If v is not a micro-tree root then we determine the subtree size within the micro-tree from the look-up table to get the first value. We then determine if v is an ancestor of the micro-tree's dummy node. Now we go to the root of the descendant micro-tree and compute the subtree size as mentioned previously to which we add the former value.

This procedure is depicted in figure 3.4. In this figure to compute the subtree size of node v, we first look-up the subtree size of v in the containing micro-tree μ_1 (the subtree size is 6). We next determine if v is an ancestor of the dummy d of micro-tree μ_1 via the look-up table. Node v is an ancestor of d in this example. Therefore, we read the subtree size of the micro-tree root r with respect to mini-tree m_1 (this value is 16). We use the dummy ancestor within mini-tree operation (discussed previously) to determine whether v is also an ancestor of dummy node of their mini-trees d'. In the example, v is indeed an ancestor of d', therefore we go to the descendant mini-tree m_2 and to its root r'. We have explicitly stored the subtree size of r' with respect to the entire tree (value 15 in this example as there are no mini-tree descendent of m_2). The subtree size of v is the sum of the values $6, 16, 15$ which is 37.

Depth and height We first show we can determine the depth and height of a node with respect to the mini-tree it belongs to. Given a node v, if it is a micro-tree root, then its depth and height within the encapsulating mini-tree are explicitly stored. If v is not a micro-tree root, the depth and height within the micro-tree are determined by querying the look-up table. The height and depth of the root of the micro-tree and the dummy node (corresponding to the exit edge from the micro-tree, if it exists) are explicitly stored. We can adjust the height and depth of v within the micro-tree to height and depth of v within the encapsulating mini-tree using these values. In the process of adjusting the depth, we need to determine whether v is an ancestor of the dummy node which is performed as a dummy ancestor operation as discussed previously.

Given the depth and height with respect to the encapsulating mini-tree, and the depth and height of the root of the mini-tree and its dummy node with respect to the entire tree, we adjust the depth and height of node v in a similar manner to obtain depth and height

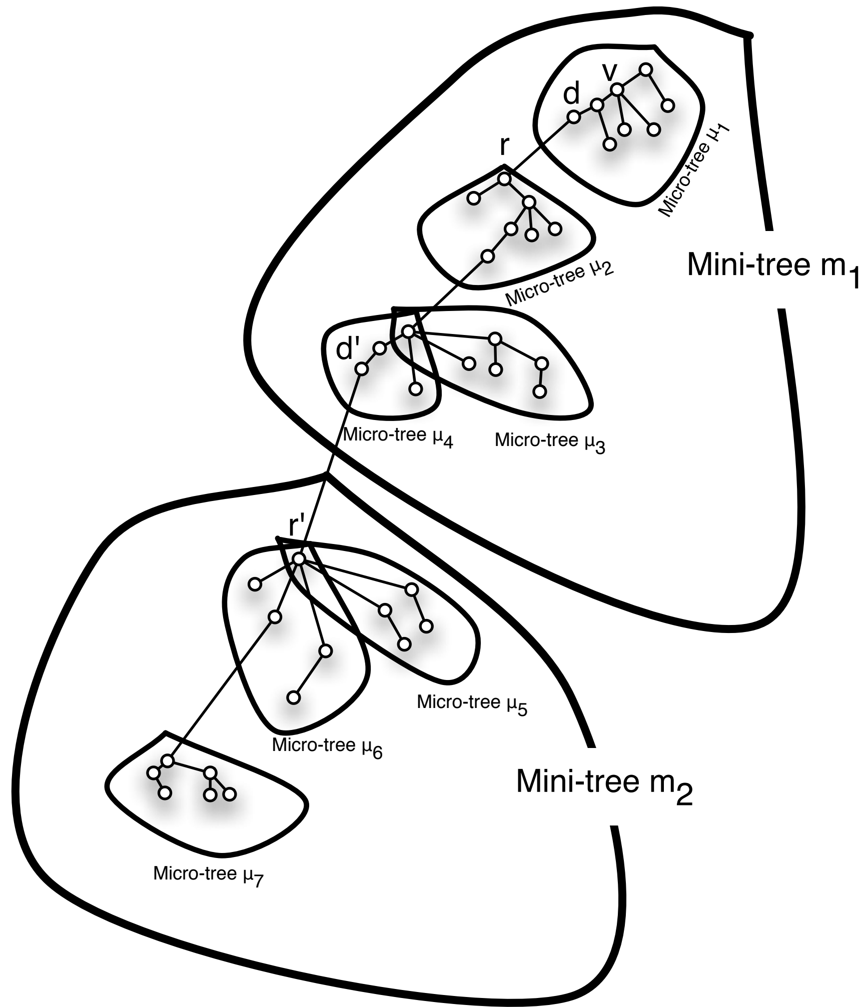


Figure 3.4: Implementation of operation subtree size. To obtain the subtree size for node v the subtree size of v in microtree μ_1 is read from the look-up table. The subtree size of r within mini-tree m_1 is added to the value and finally the subtree size of r' within the entire value is also added.

with respect to the entire tree.

Leftmost/rightmost leaf, leaf size Determining leftmost/right leaves is done by storing pointers in mini-trees and micro-trees that point to the mini-tree and micro-tree respectively which contain the leftmost/rightmost leaves. To compute leaf size, we store the number of leaves in the look-up table and with the root of each micro-tree, we store number of descendant leaves in the mini-tree and with the root of the mini-tree we store the total number of descendant leaves in the entire tree. Computation of leaf size is analogous to computation of subtree size described previously.

Leaf rank/select To compute the number of leaves that appear before a particular node v in preorder, we store the number of leaves before the roots of mini-trees in the mini-tree and similarly, within a mini-tree, we keep the number of leaves before a micro-tree roots in the micro-trees and keep leaf ranks for micro-trees in the look-up table. The computation of leaf rank is straightforward with this data. Computation of leaf select is performed as in He *et al.* [38] which is by way of a two level FID over the universe of leafs for mini-trees and micro-trees.

Node rank/select in pre/post order We store preorder and postorder rank and select data in the look-up table. Within each mini-tree we store a FID which stores the pre/post order rankings of the roots of the micro-tree within the mini-tree. Similarly, we store a FID on the entire tree which stores the rankings of the roots of the mini-trees. Using these two FIDs and the look-up table, and the fact that each micro/mini-tree has at most one edge from a non-root node going to another micro/mini-tree, it is easy to see we can perform node rank and select in pre or post order.

Least common ancestor, level ancestor, and distance There are no further simplifications for these operations in our approach and these operations can be supported as in Geary *et al.* [25], and He *et al.* [38].

Level left/right most, level successor/predecessor The support for these operations are independent of the decomposition method and can be performed as in He *et al.* [38].

Hence, we have proved the following:

Theorem 3.3. *The uniform approach can be used to encode an* **ordinal tree** *with n nodes in $2n+o(n)$ bits such that all operations listed in table 3.2 are supported in constant time.* □

3.5 Cardinal trees

In this section, we show the uniform approach can be applied to represent cardinal (k-ary) trees. This representation is the first succinct structure that supports in constant time, cardinal-type queries such as "find the child labeled j" as well as all ordinal-type queries such as subtree size, degree, or the i th child.

The number of k-ary trees with n nodes is $C(n,k) = \frac{1}{kn+1}\binom{kn+1}{n}$ [32] which suggests that a space-optimal representation requires $\lg C(n,k) = (k \lg k - (k-1)\lg(k-1))\, n - O(\lg(kn))$ bits. We assume a word-RAM model with word size $w = \max\{\lg n, \lg k\}$.

The representation is a two-level recursive decomposition of the tree analogous to the representation for ordinal trees in section 3.4. We decompose the tree with value $L = \lg^2 w$ into mini-trees and then recursively decompose each mini-tree into micro-trees with value $L = \max\left\{\frac{\lg w}{4 \lg k}, 1\right\}$. Without loss of generality, we assume $n \geq k$ and thus $w = \lg n$. All the arguments go through analogously where $k > n$ which causes $w = \lg k$ and mini-trees with $L = \lg^2 k$ and micro-trees with $L = 1$. Hence, we assume $L = \lg^2 n$ for mini-trees and $L = \max\left\{\frac{\lg n}{4 \lg k}, 1\right\}$ for micro-trees.

The representation only differs from that of ordinal trees in how we form the look-up table and represent the roots of mini/micro-trees. We distinguish nodes that are roots of a micro or a mini tree and represent them separately. The representation we use is the indexable dictionary (ID) of lemma 2.2. We use the ID to represent the set of present children at a node over the universe of k-slots ($U = \{1, 2, \ldots, k\}$). In contrast to ordinal trees, in a root of a micro-tree, we do not confine ourselves to the framework of the containing mini-tree and use the ID on all edges of the root. We note that all ordinal-tree structures are included in our representation such as the FID on roots of micro-trees built over the universe of present edges (confined to the containing mini-tree). The ID and FID structures together will help us do the cardinal-type queries as well as ordinal-type queries on a root node.

The look-up table contains all possible micro-trees. Since we keep root nodes' information separately, the trees in the look-up table does not contain the k-ary information in regards to the root nodes. In other words, the trees in the look-up table are such that all nodes are

k-ary except for the root node whose children are only ordered. We refer to such trees as *root-relaxed* cardinal trees. We enumerate all root-relaxed tree of size less than $\frac{\lg n}{4 \lg k}$ and list them in the lookup table. There is clearly less root-relaxed cardinal trees with n nodes that cardinal trees with n nodes, and hence the table has space $o(n)$.

The rest of the representation is the same as the ordinal representation; for instance, dummy nodes and edges are introduced and represented in the same manner. Now we argue that the representation is space optimal to within lower order terms.

Space optimality: All auxiliary data pertinent to the ordinal tree prove to sum to $o(n \lg k)$ bits analogously to the space analysis in section 3.4. Thus, we only have to account for the new structures we have introduced: IDs on the root nodes and the sum of indices to the look-up table. An ID on a root node v with d_v children requires $\lg \binom{k}{d_v} + o(d_v) + O(\log \log k)$ by lemma 2.2. It is easy to verify that the second and third terms add up to $o(n \lg k)$ over the entire tree and thus contribute only to lower order terms. Hence, the contribution of IDs to the space over the entire tree is $\sum_{v:\, root} \lg \binom{k}{d_v}$.

The space required to represent a micro-tree is the size of the index to the look-up table. Consider a root-relaxed tree T with root r and root children $r_1, \ldots, r_d$. We define T_i as the subtree rooted at child r_i and refer to its size as $n_i = |T_i|$. We use enumeration to encode root-relaxed trees and thus we obtain the shortest code. One (possibly inefficient) way to represent T is to encode the vector $(d, n_1, T_1, n_2, T_2, \ldots, n_d, T_d)$. Numbers n_i can be encoded in $\lg n_i + O(\log \log n_i)$ bits and T_i can be encoded by enumeration in $\lceil \lg C(n_i, k) \rceil$ bits. We show even by using this coding we achieve the optimal space to within lower order terms. Since $\sum_{i=1}^{d} n_i = |T|$ and $f(x) = \log \log x$ is concave, the sum of $O(\log \log n_i)$ is $o(|T| \log k)$ and therefore is $o(n \log k)$ when summed over the entire tree. The remaining terms sum as follows:

$$\sum_{i=1}^{d} (\lg n_i + \lg C(n_i, k)) = \lg \prod_{i=1}^{d} n_i C(n_i, k) = \lg \prod_{i=1}^{d} \binom{k n_i}{n_i - 1} \leq \lg \binom{k(|T| - 1)}{|T| - 1 - d_v}$$

Over all micro-trees these terms together with space for IDs, which is $\lg \binom{k}{d_v}$ for each root

v, sum up to:

$$\begin{aligned}\sum_{T_i}\left(\lg\binom{k(|T_i|-1)}{|T_i|-1-d_{root}}+\lg\binom{k}{d_{root}}\right) &= \lg\left(\prod_{T_i}\binom{k(|T_i|-1)}{|T_i|-1-d_{root}}\binom{k}{d_{root}}\right)\\ &\leq \lg\binom{kn}{n}=\lg\left(nC(n,k)\right).\end{aligned}$$

Thus, the space requirement of our representation matches the lower bound within lower order terms: $\lg C(n,k)+o\left(n\log k\right)$.

Operations in constant time: We can support all ordinal-type operations listed in table 3.2 in the same fashion as for ordinal trees in section 3.4. This is since we carry forward all auxiliary structures from ordinal trees to answer such queries.

It only remains to explain how to answer cardinal-type queries. At a non-root node, we use the look-up table to answer them and at root nodes we use the ID together with the FID(s) to perform the query. We explain how to determine the child labeled i of a node v. If v is not a micro-tree root, then the answer to the query is looked-up from the table. If v is a root node, then we use its ID to see if there is a child at that label. If it exists, we perform `rank(i)` on the ID to know how many siblings to the left there are. Then we can use `select` on the FID to actually determine the mini-tree and then the micro-tree and finally the child labeled i.

Theorem 3.4. *The uniform approach can be used to encode a cardinal (k-ary) tree with n nodes in $\lg C(n,k)+o\left(n\log k\right)$ bits such that all ordinal-type operations listed in table 3.2 as well as cardinal-type operation of determining the child of a node with a given label are supported in constant time.* □

3.6 Free trees

Free trees are unrooted and unordered trees. However, since distinguishing a node as the root requires only $O\left(\log n\right)$ bits, we consider free trees as rooted but unordered (*i.e.,* there is no particular order among children of nodes) and give succinct encoding for them. However,

all results apply to unrooted free trees as only $O(\log n)$ bits suffice to indicate a node as the root.

We are interested in succinct encodings of such trees allowing navigation in the tree in constant time. A *free binary tree* is a free tree such that nodes have at most two children, or alternatively a binary tree in which ignore the distinction between left and right branches. To show the power of the uniform approach we explain how these families of trees can be encoded succinctly.

Lower bounds. The lower bounds for binary and general free trees come directly from counting by information theory. Define $FB(n)$ and $F(n)$ as the number of free binary trees, and general free trees with n nodes, respectively.

There is no known explicit closed form formula for $FB(n)$ or $F(n)$. Nevertheless, asymptotic behavior of either series is well-studied [37, 55, 16, 64].

The sequence $(FB(n)), n = 1, 2, \ldots$ is known as Etherington-Wedderburn sequence [16, 64] and is known to have the following asymptotic behavior: $\lim_{n\to\infty} FB(n) = \alpha\beta^n n^{-3/2}$, where $\alpha = 0.79160\cdots$ and $\beta = 2.48325\cdots$. This implies that $\lg FB(n) = (1.3122\cdots)n + o(n)$. Otter [55] proved the following asymptotic formula for $F(n)$: $\lim_{n\to\infty} F(n) = \gamma\delta^n n^{-3/2}$, where $\gamma = 0.43985\cdots$ and $\delta = 2.95666\cdots$. This implies that $\lg F(n) = (1.5639\cdots)n + o(n)$.

Theorem 3.5. *The information-theoretic lower bound on the number of bits required to represent free binary trees and free general trees with n nodes is $(1.3122\cdots)n + o(n)$ and $(1.5639\cdots)n + o(n)$ respectively.* □

This implies that free trees can be represented more space-efficiently than ordinal trees which require $2n + o(n)$ bits.

Upper bounds. The representation differs from that of ordinal trees in section 3.4 in the look-up table. In the case of free binary trees, all free binary trees of size up to $\frac{1}{4}\lg n$ are enumerated modulo isomorphisms and listed in the look-up table in increasing order of their sizes. To represent a micro-tree we use a pair (k, i) index to the table. k is the size of the micro-tree and i is the index to the look-up table which is an offset from the start location of trees with size k. All auxiliary data are carried forward from ordinal trees as they only take $o(n)$ space. As in section 3.4, one can easily argue that the total bits required by the

first fields of pairs (k) is also $o(n)$. Therefore, the dominant field is the sum of the bits of the second fields of pairs (i). Theorem 3.5 suggest that the size of this field for a tree of size t is $(1.3122\ldots)t + o(t)$ bits. The second term $o(t)$ term adds up to $o(n)$ over the entire tree. The first term is the dominant term which adds up to $(1.3122\ldots)n + o(n)$ over the entire tree when n is the number of nodes.

The $(1.5639\cdots)n + o(n)$ bit representation for free general trees is analogous; the look-up table lists free general trees as opposed to free binary trees:

Theorem 3.6. *The succinct representation for free binary trees and free general trees with n nodes requires $(1.3122\cdots)n + o(n)$ and $(1.5639\cdots)n + o(n)$ bits respectively and supports all navigational operations listed in table 3.2 in constant time.* □

3.7 Entropy-based succinct encodings

Thus far in this chapter, we assumed a uniform distribution among trees belonging to a certain family of trees. However, there might be many applications so that some trees are biased against other trees within the tree family, and therefore the distribution is non-uniform. Thus, entropy-based succinct encodings are necessary. Jansson *et al.* [41] were the first to give entropy-based succinct encodings for the degree-distribution entropy. In this section, we show how our method can be used to match the degree-distribution entropy as well as a variety of other entropy measures.

3.7.1 Succinct encoding based on degree-distribution entropy

The degree-distribution of an ordinal tree with n nodes is a series of numbers $(n_0, n_1, \ldots)$ such that the tree has n_i nodes that have exactly i children ($\sum_i n_i = n$ and $\sum_i i\, n_i = n - 1$). Rote [61] showed that the number of trees with a given degree-distribution is $\frac{1}{n}\binom{n}{n_0, \ldots, n_{n-1}}$, the logarithm based two of which is $L(T) = \sum_i n_i \lg \frac{n}{n_i}$ to within lower order terms. $L(T)$ is therefore a lower bound on the required number of bits to represent such trees succinctly.

Jansson *et al.* [41] gave a representation that requires $L(T) + O(n(\log\log n)^2/\log n)$ number of bits and supports a variety of operations in constant time. Using our approach we obtain another space-optimal succinct representation with $L(T) + O(n \log\log\log n/\log\log n)$ number of bits supporting all operations in table 3.2 in constant time.

Jansson *et al.* [41] did not assume that the degree-distribution is explicitly given. We first show that we can make explicit the assumption that the degree-distribution is given as it takes negligible space to encode the sequence. Thus we can augment the succinct representation with it.

Lemma 3.7. *The degree-distribution of a tree with n nodes can be encoded in $O(\sqrt{n}\log n)$ bits.*

Proof. There can be at most $\Theta(\sqrt{n})$ distinct values of n_i larger than zero, since otherwise, the sum $\sum_i in_i$ would exceed n. We form a bit vector of size n which has raised bits at position i where $n_i > 0$. We encode this bit vector using the indexable dictionary representation of lemma 2.2. This requires $O(\sqrt{n}\lg n)$ bits as there are at most $\sqrt{n}$ non-zero bits. Now, we use $\lceil \lg n \rceil$ bits to represent non-zero values of the sequence which together use $O(\sqrt{n}\lg n)$ bits. □

Our succinct representation sensitive to degree-distribution entropy is the same as that of the ordinal trees with the difference in the look-up table. The look-up table contains all trees with less than $\frac{1}{4}\lg n$ as in ordinal trees; However, the trees are ordered based on their degree-distribution sequence in the lexicographical order and listed in the table accordingly. In order to encode a micro-tree T, we use an index to the table. The index to the table is a pair $(\mathcal{N}, k)$ where $\mathcal{N}$ is the degree-distribution encoding of the tree and k is an offset in the table from where the trees with degree-distribution $\mathcal{N}$ start to the actual position of tree T we reference to.

By lemma 3.7, the total number of bits required by the first fields of indices (*i.e.*, $\mathcal{N}$) is $O\left(n \log\log n/\sqrt{\lg n}\right)$ and thus negligible. The second field k is the dominant term. The size of this field for a micro-tree T_t, by a counting argument, is at most

$$\lceil L(T_t) \rceil = \left\lceil \lg\left(\frac{1}{|T_t|}\binom{|T_t|}{n_{t,0}\ldots n_{t,|T_t|}}\right)\right\rceil,$$

where $n_{t,i}$ is the number of nodes with i children in tree T_t. Hence, the sum of the length of the second fields over all micro-trees $T_1, \ldots, T_m$ is the dominant term in the space requirement of our method and can be calculated as follows:

$$\begin{aligned}
\sum_{t=1}^{m} L(T_i) &= \sum_{t=1}^{m} \sum_{i \geq 0} \lg \left(\frac{1}{|T_t|} \binom{|T_t|}{n_{t,0} \ldots n_{t,|T_t|}} \right) \\
&= \lg \prod_{t,i} \frac{1}{|T_t|} \binom{|T_t|}{n_{t,0} \ldots n_{t,|T_t|}} \\
&\leq \lg \prod_{t,i} \binom{|T_t|}{n_{t,0} \ldots n_{t,|T_t|}} \\
&\leq \lg \prod_{t,i} |T_t| \binom{|T_t| - 1}{\tilde{n}_{t,0} \ldots \tilde{n}_{t,|T_t|}},
\end{aligned}$$

where $\tilde{n}_{t,i}$ is the number of non-root nodes with i children in T_t. Analogously, we define $\tilde{n}_i$ be the number of non-root nodes in the entire tree with i children. Using a simple combinatorial counting argument, we infer:

$$\begin{aligned}
\sum_{t=1}^{m} L(T_i) &\leq \sum_{t} \lg |T_t| + \lg \binom{n-m}{\tilde{n}_0, \ldots, \tilde{n}_{n-1}} \\
&\leq \sum_{t} O(\log \log n) + \lg \binom{n}{n_0, \ldots, n_{n-1}} \\
&= L(T) + O\left(\frac{n \log \log n}{\log n} \right).
\end{aligned}$$

Hence, the representation has the optimal space to within lower order terms and clearly we can perform all operations listed in table 3.2 in constant time as in an ordinal tree.

3.7.2 Other entropy measures

Similar to the manner in which we represented trees adaptive to their degree-distribution entropy, we can use the approach to obtain succinct representations adaptive to various other combinatorial properties and entropy measures. For instance, consider the family of ordinal trees such that internal nodes have at least two children. The number of such trees with n nodes is known as Riordan number [9]. The logarithm based two of Riordan numbers is asymptotically $\lg(3)n + o(n) \approx 1.58n + o(n)$. One can use our approach to encode this family

of trees. Similarly, The family of ordinal trees with a fixed constant upper bound d on the number children of a node can be represented in the same manner. More generally, where there is a probability distribution for the number of children of a node, our representation can match the entropy bound. In all these representations, the only change necessary is the look-up table which for these trees contains an enumerated list of such trees.

Another interesting family of trees is AVL trees which consists of binary trees such that the height of left and right subtrees differ by at most one. Odlyzko [54] showed that if a_n is the number of AVL trees with n nodes, $\lg a_n \approx (0.9381 \cdots)n + o(n)$, our representation matches this entropy bound and thus can represent AVL trees in optimal number of bits within lower order terms. One can argue that, since the tree is an AVL tree, most nodes belong to a leaf micro-tree. We call a micro-tree as a leaf micro-tree if no non-root node has a child outside of the micro-tree. One can easily see than the number of nodes not belonging to a leaf micro-tree is $O(n/\log n)$. All leaf micro-trees are AVL trees and therefore the table look-up contains only AVL trees. Therefore, their representation size matches their entropy. Micro-trees which are not leaves are not necessarily AVL trees but as we argued, their collective size is $O(n/\log n)$ and therefore, we can afford to encode them using merely an ordinal tree representation (and not entropy-based).

3.8 Summary and discussion

In this section, we proposed a uniform approach towards succinct representation of trees. We showed that all families of trees with an existing succinct representation can be represented using our framework. Our representation improves on the existing ones on cardinal trees as we are able to answer ordinal-type queries such as subtree size as well as cardinal-type queries. We consider a new family of trees for succinct encoding: *free trees*. We demonstrated how easily our approach can represent these trees succinctly

We argued that our approach can represent trees succinctly adaptive to the degree-distribution entropy. We discussed that a variety of other entropy measures can be dealt with similarly.

Chapter 4

Universal representations of ordinal trees

4.1 Introduction and motivation

In chapter 3, we presented a new approach towards succinct encodings of trees which encompasses a wide range of families of trees. In this chapter, we focus on the family of ordinal trees. The uniform approach of chapter 3 gives a succinct representation for this family of trees that supports a comprehensive list of queries and operations on the trees in constant time. The list of supported operations includes almost all operations supported by all other existing succinct encodings (see table 3.2).

There is no guarantee, however, that any operation defined and desired in the future will be supported by the representation in constant time. The list of supported operations in each representation is dynamic and expands; for instance, the list of supported operations on the balanced parenthesis representation was recently expanded substantially [45]. There are operations which can be implemented in constant time in one of the other representations but not implementable (as yet) in the uniform approach of chapter 3 (*e.g.,* DFUDS node rank/select which is rank and select operations on nodes in the order which DFUDS traverses the tree). The set of such operations can increase as new operations different from the existing operations arise. Therefore, it is desirable to have an umbrella succinct representation of ordinal trees which unifies the power of all existing succinct representations.

The existing representation approaches are the balanced parenthesis (BP), the depth first unary degree sequence (DFUDS), the tree covering (TC) representations, and the level order unary degree sequence (LOUDS). The LOUDS representation is different in nature from all the other representations and, furthermore, has very limited power in supporting operations. Therefore, in this chapter we focus on the other three representations. By the tree covering representation, we mean the uniform approach of chapter 3 as the original tree covering representations leave a degree of freedom in decomposition of trees into mini and micro trees and the uniform approach of chapter 3 adopts a more careful approach in decomposing the tree into mini-tree and micro-tree components. As a result, any operation supported in constant time in a general tree covering representation can naturally be supported in constant time in the uniform representation. Hence, in the rest of this chapter, we refer to the uniform approach of chapter 3 by the tree covering (TC) approach.

In this chapter, we give a succinct representation for ordinal trees that encapsulates all other representations. Namely, we give a representation for an ordinal tree which can be used as a black box to simulate any of the BP, DFUDS, or the TC representations in constant time. This automatically means that the new representation supports, in constant time, the union of all the operations the other representations support in constant time.

4.2 Succinct ordinal trees representations

As noted, there are four approaches that have been used for succinct representation of an ordinal tree: the level order unary degree sequence (LOUDS), the balanced parenthesis (BP), the depth first unary degree sequence (DFUDS), and the tree covering (TC) representations. In this section, we give an overview of these representations. Figure 4.1 illustrates an ordinal tree together with the LOUDS, BP, and the DFUDS sequences. Table 4.1 compares the query support power of these approaches (LOUDS representations are very limited in supporting queries and support none of the queries listed in the table in constant time.)

4.2.1 Level order unary degree sequence (LOUDS)

Jacobson's succinct tree representation [40] was based on the *level order unary degree sequence* of a tree, which lists the nodes in a level-order traversal. The ordering puts the root

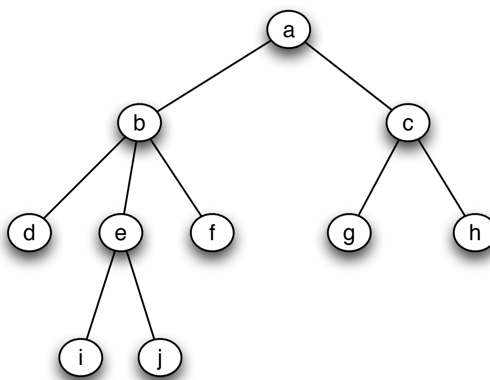
Query	BP [49, 50, 12, 53, 45]	DFUDS [8, 41]	TC [25, 38],Chap. 3
pre-order node rank/select	✓	✓	✓
post-order node rank/select	✓		✓
DFUDS node rank/select		✓	
degree	✓	✓	✓
child rank/select	✓	✓	✓
height	✓		✓
depth	✓	✓	✓
subtree size	✓	✓	✓
lowest common ancestor	✓	✓	✓
distance	✓	✓	✓
level ancestor	✓	✓	✓
leaf rank/select and size	✓	✓	✓
leaf leftmost and rightmost	✓	✓	✓
level pred. and succ.	✓		✓
level leftmost and rightmost			✓

Table 4.1: Main approaches in succinct representations of ordinal trees and the queries they support in constant time: balanced parentheses (BP), depth first unary degree sequence (DFUDS), and tree covering (TC).

first, then all of its children, from left to right, followed by all the nodes at each subsequent level (depth) of the tree and encodes their degrees in unary (*i.e.,* a sequence of 1's terminated by a 0). With this, Jacobson [40] 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 [14] 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 other ways to represent ordinal trees using $2n + o(n)$ bits.

4.2.2 Balanced parenthesis (BP)

Munro and Raman [49, 50] proposed another succinct representation of trees based on the balanced parenthesis sequence of an ordinal tree. 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



LOUDS sequence: 1 1 0 1 1 1 0 1 1 0 1 0 1 1 0 0 0 0 0 0.

	a	b	c	d	e	f	g	h
LOUDS:	1 1 0	1 1 1 0	1 1 0	1 0	1 1 0	0	0	0

BP sequence: ((( ) (( ) ( )) ( )) (( ) ( ))).

	a	b	d	d	e	i	i	j	j	e	f	f	b	c	g	g	h	h	c	a
BP	(	(	(	)	(	(	)	(	)	)	(	)	)	(	(	)	(	)	)	)

DFUDS sequence: (( ) ((( )) (( )))) (( ))).

	a	b	d	e	i	j	f	c	g	h
DFUDS:	(( )	((( )	)	(( )	)	)	)	(( )	)	)

Figure 4.1: An ordinal tree and the corresponding LOUDS, BP, and DFUDS sequences.

are visited (see figure 4.1). Opening and closing parentheses are replaced with 0's and 1's respectively to obtain the BP sequence bits. They presented a succinct representation of an ordinal tree of n nodes in $2n + o(n)$ bits based on the BP sequence, which supports a limited number of navigational operations and queries in constant time. Chiang *et al.* [12], Munro and Rao [53] , and Lu and Yeh [45] improved the BP representations by incrementally expanding the set of queries supported in constant time.

4.2.3 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) (see figure 4.1). Opening and closing parentheses are

replaced with 0's and 1's respectively to obtain the DFUDS sequence bits. Benoit*et al.* [8] presented the first succinct tree representation based on DFUDS: the representation requires $2n + o(n)$ bits and supports a limited number of operations in constant time in constant time. Jansson [41] extended this representation using $o(n)$ additional bits by providing constant-time support for some additional queries.

4.2.4 Tree covering (TC)

Another approach to represent static ordinal trees is based on a *tree covering* algorithm. Geary *et al.* [25] proposed an algorithm to decompose an ordinal tree to a set of mini-trees, each of which is further decomposed into a set of micro-trees. Their representation occupies $2n+o(n)$ bits, and supports a limited set of queries in constant time. He *et al.* [38] extended the set of queries supported in constant time. Both of these schemes leave a large degree of freedom with the decomposition algorithm to decompose the tree into mini-trees and further to micro-trees. In chapter 3, we proposed a tree decomposition algorithm and described how this decomposition algorithm results in a simple and straightforward representation for ordinal trees. We can replace any decomposition algorithm in representations of the TA approach with our decomposition scheme of section 3.3. Hence, our uniform representation of chapter 3 can be viewed as one realization of the existing tree covering approaches. Therefore, in the rest of this chapter, we refer to our representation as the TC representation.

4.3 The unified tree representation

In this section, we present the new representation that unifies the three approaches of BP, DFUDS and TC. All these representations consist roughly of two parts: one part which claims the dominant term of the space encodes the tree (*i.e.,* $2n$ bits for a tree with n nodes) and the other part is of size a lower order terms of the first part (*i.e.,* $o(n)$ bits) and stores all additional information to perform navigation and various operations.

In BP and DFUDS representations, the first part is the bit vector corresponding to the sequence of open and closed parentheses, and the second part is all other auxiliary information which is mostly maintained to perform operations in constant time (particularly `rank` and `select`). In tree covering representations the first part consists of encodings of

all micro-trees, and the second part is all pointers and auxiliary information we maintain to perform queries in constant time.

In the new unified representation, we can afford to replicate the second part of each approach as they contribute only to lower order terms. The challenge is to replicate the first parts without increasing the highest order term ($2n$ bits for a tree with n nodes). We will show this by giving a representation and demonstrating how each word of the BP and DFUDS representations and also each micro-tree encoding in the TC representation can be produced in constant time. By doing this, we have proved that the new representation can be used as a blackbox to emulate BP, DFUDS, and TC representations at once.

Hence, we define the following three operations and show that the new unified representation can perform them in constant time:

Definition 4.1. *Operation* `BP-word(`k`)` *returns the* k*-th* $\lg n$ *bits of the BP representation. Operation* `DFUDS-word(`k`)` *returns the* k*-th* $\lg n$ *bits of the DFUDS representation. Operation* `TC-microtree(`m`,`μ`)` *returns the encoding of the* μ*-th micro-tree in the* m*-th mini-tree.*

The representation is based on a one-level decomposition approach. We use the decomposition algorithm in section 3.3 for value $L = \lceil \lg^2 n \rceil$ to decompose the tree into mini-trees. However, unlike section 3.4, there is no further decomposition into micro-trees. The representation of a mini-tree is more complicated and is described in section 4.4. We first show in section 4.3.1 that we can reduce the problem to within individual mini-trees.

4.3.1 Reduction to within mini-trees

In this section, we demonstrate that the problem of supporting previous representations (BP, DFUDS, and TC) can be confined to mini-trees. In other words, if any BP or DFUDS word and any micro-tree encoding corresponding to a mini-tree can be generated in constant time, then any BP or DFUDS word and any micro-tree encoding corresponding to the entire tree can be generated in constant time. The claim is obvious for the micro-tree encodings of the TC representation as micro-tree encodings within a mini-tree are the same as for the entire tree. We prove the claim for the BP and DFUDS representations in this section.

In both the BP and DFUDS representations of a given tree, each node corresponds to two bits (one open and one closing parenthesis); in the BP representation, the first bit corresponding to a node (an opening parenthesis) is placed where the node is first discovered

in the pre-order traversal and the second bit corresponding to the node (a closing parenthesis) is placed where the subtree of the node is fully discovered in the pre-order traversal. In the DFUDS representation, nodes are traversed in the depth-first order and their degrees represented in unary format: opening parentheses followed by a single closing parenthesis. The first bit corresponding to a node is the corresponding open parenthesis in the unary representation of the degree of the parent of the node. The second bit corresponding to a node is the trailing closing parenthesis in the unary representation of its degree. The root of the entire tree is an exception in that it misses the first bit; the exception is handled by adding an extra opening parenthesis at the beginning of the representation.

Thus given any subset of the nodes, one can talk about the set of bits corresponding to the set of nodes. We note that as mini-trees may share their roots, their corresponding set of bits may share bits which represent their roots. We first argue that each mini-tree of any tree corresponds to a set of $O(1)$ consecutive subsequences from the BP/DFUDS sequence of the tree.

Lemma 4.1. *In the BP (or the DFUDS) sequence of a tree, the bits corresponding to a mini-tree form a set of constant number of consecutive subsequences. Furthermore, these subsequences concatenated with each other in order, form the BP (or the DFUDS) sequence of the mini-tree.*

Proof. The decomposition algorithm in section 3.3 guarantees by lemma 3.1 that each mini-tree has the property that other than the mini-tree root, at most one node has a child outside that mini-tree. Moreover, it is easy to observe that a mini-tree contains a consecutive subset of children of its root.

The BP and DFUDS sequences are based on ordered depth first traversal of the tree. In the BP sequence, bits corresponding to a mini-tree are spread over at most four consecutive subsequences. The bits corresponding to the mini-tree root form two singleton blocks by themselves and the bits corresponding to the non-root nodes of the mini-tree occur somewhere in between these two bits. There is at most one (exit) edge going out of the entire non-root nodes of the mini-tree. As a result, the bits corresponding to non-root nodes form at most two consecutive subsequences. Thus, the possible four blocks are as follows:

1. opening parenthesis corresponding to the mini-tree root,
2. closing parenthesis corresponding to the mini-tree root,

3. the bits corresponding to nodes of the mini-tree that are visited on the first entrance by the depth-first traversal until the traversal leaves the mini-tree on the singleton exit edge, and

4. the bits corresponding to nodes of the mini-tree that are visited on the second entrance of the traversal until it leaves the mini-tree through its root.

In the DFUDS sequence, bits corresponding to nodes of a mini-tree are spread over at most six consecutive subsequences. Two bits corresponding to the mini-tree root potentially form two such subsequences. Opening parentheses corresponding to the children of the mini-tree root form the third consecutive subsequence. There is at most one edge (u, v) leaving a non-root node u of the mini-tree to a node v outside the mini-tree; the DFUDS sequence exits the mini-tree at most twice: once to represent the degree of u and the second time on the actual representation of the subtree rooted at node v. These two points of exit define three consecutive subsequences each of which is entirely within the mini-tree. Thus, the possible six blocks are as follows:

1. opening parenthesis corresponding to the mini-tree root,

2. closing parenthesis corresponding to the mini-tree root,

3. opening parentheses corresponding to the children of the mini-tree root,

4. the bits corresponding to nodes of the mini-tree that are visited to the point where the degree of u is encoded.

5. the bits corresponding to nodes of the mini-tree that are visited on the second visit of the traversal in the mini-tree to the point where the traversal exits to encode the subtree rooted at v.

6. the bits corresponding to nodes of the mini-tree that are visited on the third visit of the traversal in the mini-tree to the point where the traversal exits the root of the mini-tree forever.

Thus, the BP and the DFUDS sequences corresponding to a mini-tree are spread over four and six consecutive subsequences respectively. As these consecutive subsequences occur in

order, the BP and DFUDS sequences are formed by combining their respective subsequences in order. □

We are now ready to present the main result of this section, which shows that support for `BP-word(k)` and `DFUDS-word(k)` operations in constant time within mini-trees implies support for the same operations in constant time over the entire tree:

Theorem 4.2. *If there is a structure which represents a mini-tree with m nodes ($m = O((\log n)^2)$) in $2m + o(m)$ bits and supports operations* `BP-word()` *and* `DFUDS-word()` *in constant time, then the entire tree with n nodes can be represented in $2n + o(n)$ bits and operations* `BP-word()` *and* `DFUDS-word()` *supported in constant time.*

Proof. Lemma 4.1 guarantees that bits corresponding to a mini-tree are broken into $O(1)$ consecutive chunks. Since there are $O(n/(\log n)^2)$ mini-trees, the entire BP/DFUDS sequence of length $2n$ is split into $O(n/(\log n)^2)$ chunks. We store a fully indexable dictionary (FID) as described in lemma 2.1 which stores the starting positions of chunks over the universe of all positions in the BP/DFUDS sequence (*i.e.,* $\{1, \ldots, 2n\}$). As mentioned previously chunks can overlap, and in the event that two chunks share a starting position, one is chosen arbitrary. Along with each chunk, we store a reference to the mini-tree it belongs to, the length of the chunk, and finally the offset of the starting position of the chunk within the bits associated with the mini-tree. Furthermore, for each chunk, we store the proceeding $\lg n$ bits of the BP and DFUDS sequence to the chunk. The space requirement of the FID is $O\left(\frac{n \log n}{\log^2 n} + \frac{n \log\log n}{\log n}\right)$, and the storage requirement of other auxiliary data stored is $O\left(\frac{n \log n}{\log^2(n)}\right)$. Hence, all the additional data kept contributes to $o(n)$ bits overall.

In order to support `BP/DFUDS-word(k)`, we need to output the bit sequence from position $k \lg n$ to $(k+1)\lg n$ in constant time. We use the FID to locate the chunk which contains position $k \lg n$. We use the reference stored to determine the mini-tree the chunk belongs to. Given the offset of the starting position of the chunk in the mini-tree bit sequence, using the `BP/DFUDS-word` operation within the mini-tree we can produce the next $\lg n$ bits in constant time.

In the event that position $(k+1)\lg n$ falls outside the chunk, we use the portion which occurs inside the chunk and concatenate it with the explicitly stored $\lg n$ bits proceeding the chunk, to produce the desired bit sequence. □

Theorem 4.2 shows that the problem of representing the entire tree succinctly to support both `BP-word` and `DFUDS-word` in constant time reduces to the same problem confined to within mini-trees. In the next section, we give a succinct representation for mini-trees which can support these operations in constant time.

4.4 Supporting representations within a mini-tree

In this section, we confine our attention to within mini-trees. We give a new representation scheme for mini-trees and argue how the scheme supports the previous representations in a mini-tree.

We start by showing support for the BP and DFUDS representations. We present the new representation for a mini-tree in section 4.4.1 and show how any word (of length $\Theta(\log n)$) of the BP and DFUDS encodings can be produced in constant time. Section 4.4.2 demonstrates how the new representation can support the tree covering representation by showing how micro-trees can be produced in constant time.

4.4.1 Support for the BP and DFUDS encodings

Given a mini tree consisting of $k = O(\log^2 n)$ nodes, we now describe how to represent it using $2k + o(k)$ bits to produce any $O(\log n)$ bit subsequence of its BP and DFUDS sequences in constant time.

Definition 4.2. *We define a node in the mini tree as* **significant** *if its subtree size is larger than* $\frac{\lg n}{16}$*. Note that all the significant nodes form a connected subtree as all the ancestors of a significant node are also significant. We call this subtree the* **skeleton** *of the mini tree. Since each leaf in the skeleton has at least* $\frac{\lg n}{16}$ *nodes in its subtree (which are not part of the skeleton), the number of leaves in the skeleton is* $O(n/\log n)$.

We start with the easy case where the skeleton is only a path and then focus on the general case where the skeleton can be an arbitrary tree.

Skinny trees

We first consider the case when the skeleton is a *path*, and call such a tree *skinny* (*e.g.,* see figures 4.2(a),(b)). Let u be the leaf of the skeleton (which is a path) and let v be the last

(rightmost) child of u in the tree. We denote the set of all immediate children of the nodes of the skeleton whose preorder numbers are at most the preorder number v (including v) by S_D, and the set of all immediate children of the nodes of the skeleton whose preorder numbers are more than the preorder number of v by S_U. The new representation consists of the following four components:

- *Path down,* P_D: This consists of unary representations of the number of children of each node of the skeleton in the set S_D, in order from the root to leaf u (*e.g.,* see figure 4.2(c)).
- *Path up,* P_U: This consists of the unary representations of the number of children of each node of the skeleton in the set S_U, from leaf u to root (*e.g.,* see figure 4.2(d)).
- *Trees on path down,* T_D: Let $D_1, D_2, D_3, \ldots$ be all the subtrees rooted at the nodes of S_D, ordered by the preorder numbers of their roots. The bit sequence T_D is obtained by concatenating the BP representations of each of the trees T_i with the first bit (opening parenthesis) removed (*e.g.,* see figure 4.2(e)).
- *Trees on path up,* T_U: Let $U_1, U_2, U_3, \ldots$ be all the subtrees rooted at the nodes of S_U, ordered by the preorder numbers of their roots. The bit sequence T_U is obtained by concatenating the BP representations of each of the trees T_i with the first bit (opening parenthesis) removed (*e.g.,* see figure 4.2(f)).

These four components are shown for an example tree in figure 4.3.

We first show how to reconstruct the original tree form these four components. This can be done by first reconstructing the skeleton and all its immediate children using P_D and P_U. Then we attach the subtrees to the immediate children of the skeleton using T_D and T_U. The important thing to note is that the representations of subtrees in both T_D and T_U are self-delimiting, as these are the BP representations of a tree with the first bit (open parenthesis) removed.

The sum of the sizes of the four components is exactly twice the number of nodes in the mini tree. This follows form the following observations. Each node of the skeleton is represented using one bit in P_D and one bit in P_U (these are the last bits in the unary representations of the number of children of the node in S_D and S_U). Each of the immediate

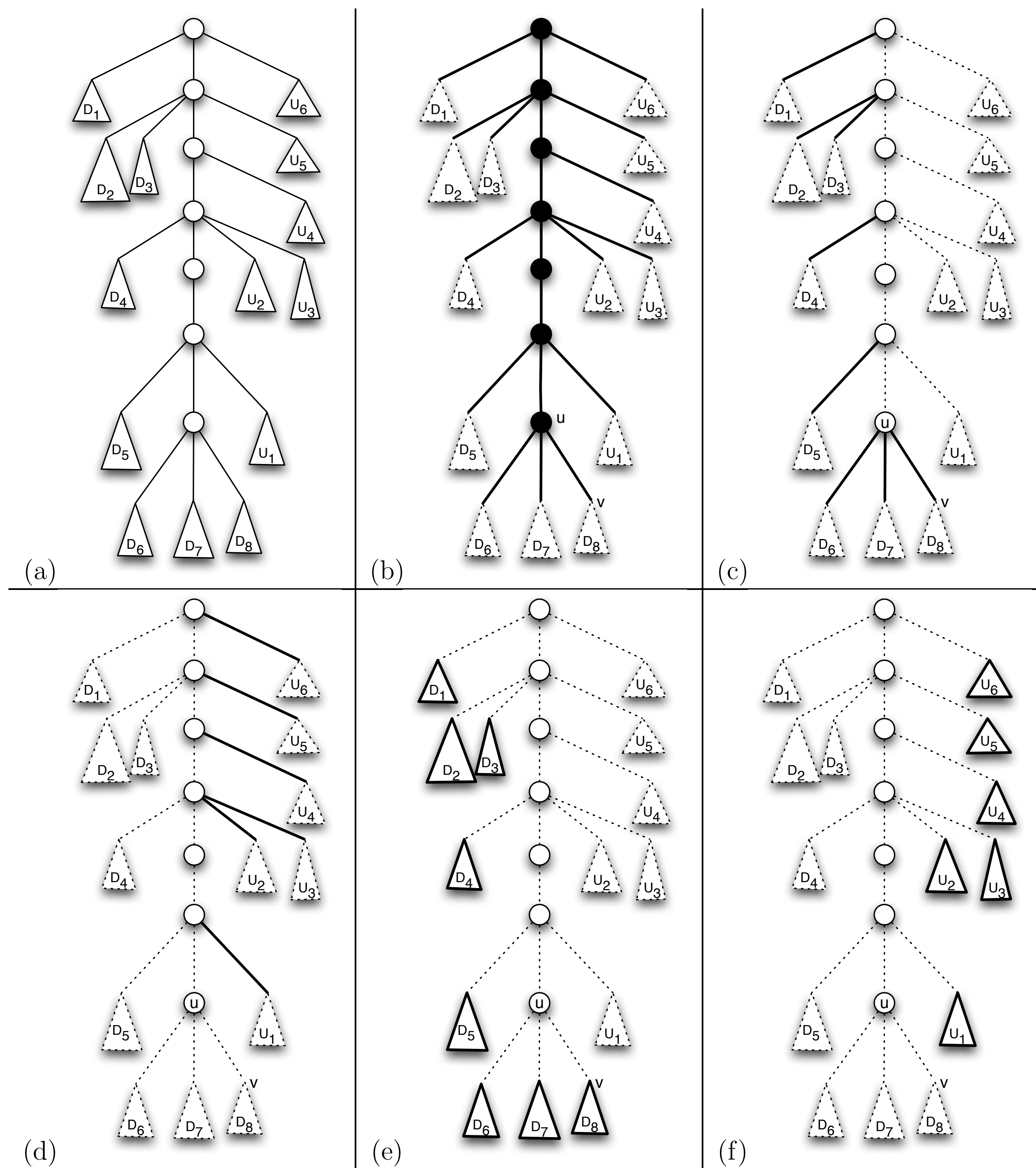
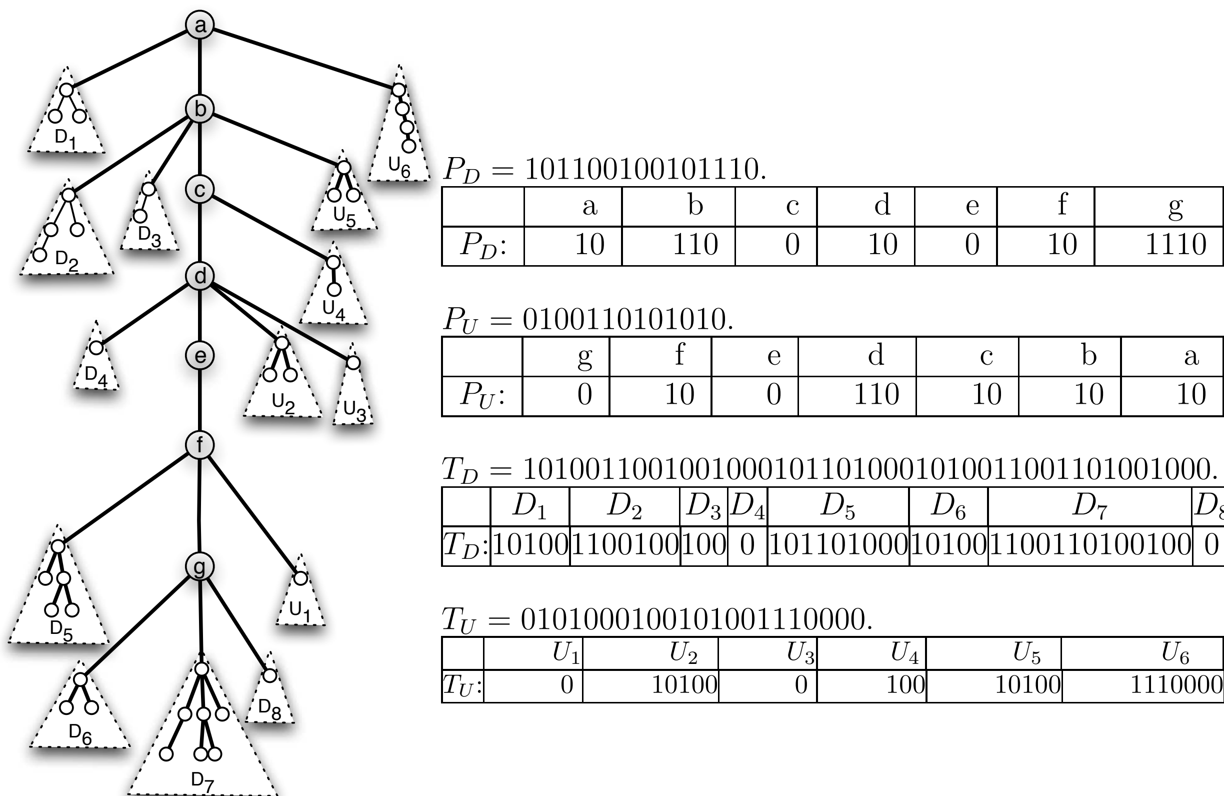


Figure 4.2: Different components of a skinny tree in the new unified representation. (a) A skinny tree (b) The skeleton of the tree along with their immediate children. (c) P_D: unary representation of skeleton degrees to the left. (d) P_U: unary representation of skeleton degrees to the right. (e) T_D: Concatenation of the BP sequences of subtrees to the left. (f) T_U: concatenation of the BP sequences of subtrees to the right.



$P_D = 101100100101110.$

	a	b	c	d	e	f	g
P_D:	10	110	0	10	0	10	1110

$P_U = 0100110101010.$

	g	f	e	d	c	b	a
P_U:	0	10	0	110	10	10	10

$T_D = 10100110010010001011010001010011001101001000.$

	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8
T_D:	10100	1100100	100	0	101101000	10100	1100110100100	0

$T_U = 0101000100101001110000.$

	U_1	U_2	U_3	U_4	U_5	U_6
T_U:	0	10100	0	100	10100	1110000

Figure 4.3: Four components of a skinny tree representation (P_D, P_U, T_D, and T_U) are given for a skinny tree.

children of the skeleton are represented using one bit in either P_D or P_U (when we write the unary representation of the number of children of its parent in either S_D or S_U), and one bit in either T_D or T_U (the last bit in the BP representation of the subtree rooted at that node). Each of the other nodes is represented by two bits in T_D if it is a descendant of a node in S_D or by two bits in T_U if it is a descendant of a node in S_U.

Supporting the BP representations for a skinny tree. We now show how to produce a word of the BP sequence from the four-component representation of the skinny tree. We divide the BP sequence into blocks of length $\lceil \frac{\lg n}{8} \rceil$, starting at every multiple of $\lceil \frac{\lg n}{8} \rceil$. We prove in lemma 4.3 that we can produce each such block in constant time.

Lemma 4.3. *Each block of length $\lceil \frac{\lg n}{8} \rceil$ from the BP sequence of a skinny mini tree can be*

reported in constant time using $o\left((\log n)^2\right)$ *extra space.*

Proof. Corresponding to the starting position of every block, we store an "index" of size $\Theta(\log\log n)$ to our representation of the skinny tree. Let t be the node corresponding to the starting bit of the block in the BP sequence. The index has an `up/down` field, which indicates we are traversing down or up the skeleton (*i.e.,* preorder number of t is less than that of v which is the rightmost child of the skeleton leaf). Let t' be the first ancestor of t which is in S_D or S_U (depending on the value of `up/down`). The index includes two pointers to locations in components P_D, T_D (or P_U, T_U) which correspond to node t'. In addition, the offset of the bit corresponding to t in the BP sequence of the subtree rooted at t' in T_D or T_U is stored as a separate field in the index.

Since the size of each index is $\Theta(\log\log n)$ bits, the total size of such indices is $\Theta(\log n \log\log n)$, which is $o\left((\log n)^2\right)$. It only remains to show a block can be produced in constant time. To produce a block, we use the index associated with the starting position of the block. The key claim is that starting from node t', we can generate $\frac{\lg n}{4}$ bits of the BP sequence.

Without loss of generality, assume the `up/down` field is set to `down` and t' the BP sequence is traversing down the skeleton. The BP bits following the bit corresponding to t' are stored in order in T_D and P_D; these two sequences only need to be merged together. Since the BP sequence of a subtree in T_D is self-delimiting, we can perform the merge by a look-up table. We form a table for all possible values of (t_d, p_d) where t_d, p_d are two bit vectors of size $\lceil \frac{\lg n}{4} \rceil$ each. There is only one copy of this table stored and shared among all mini-trees and has size $o(n)$. Using this look-up table, we can look-up on constant time, the next $\frac{\lg n}{4}$ bits of the BP sequence. In the event, we reach the rightmost leaf of the skeleton leaf with these bits and change directions from downward traversal on the skeleton to an upward traversal, we break the bits to two segments and report each segment independently.

By reporting $\frac{\lg n}{4}$ bits starting from the bit corresponding to t', we are guaranteed to have covered the $\frac{\lg n}{8}$ bits which start from the bit of t, since the size of each subtree is at most $\frac{\lg n}{16}$ by definition. Hence, using the stored offset of t, we remove the redundant leading bits and report the block starting at the bit corresponding to t. □

By lemma 4.3, each block of length $\lceil \frac{\lg n}{8} \rceil$ bits can be produced in constant time. This implies that any word of length $\lceil \lg n \rceil$ bits can be generated in constant time:

Theorem 4.4. *The new unified representation supports operation* `BP-word()` *in constant time in a skinny mini-tree using* $o\left((\log n)^2\right)$ *extra space.*

Supporting the DFUDS representations for a skinny tree. We now show how to produce a word of the DFUDS sequence from the four-component representation of the skinny tree. Analogous to producing BP bits, We divide the DFUDS sequence into blocks of length $\lceil\frac{\lg n}{24}\rceil$, starting at every multiple of $\lceil\frac{\lg n}{24}\rceil$. We prove in lemma 4.5 that we can produce any block in constant time.

Lemma 4.5. *Each block of length* $\lceil\frac{\lg n}{24}\rceil$ *from the DFUDS sequence of a skinny mini tree can be reported in constant time using* $o\left((\log n)^2\right)$ *extra space.*

Proof. In a similar manner to the proof of lemma 4.3, corresponding to the starting position of every block, we store an "index" of size $\Theta(\log\log n)$ to our representation of the skinny tree. Let t be the node corresponding to the starting bit of the block in the DFUDS sequence. We distinguish two cases whether or not t belongs to S_D or S_U (an immediate children of the skeleton nodes) and the bit is an opening parenthesis. If so, then the starting bit of the block occurs somewhere in the unary encoding of the degree of a skeleton node; and otherwise, the bit belongs to the encoding of the degree of a non-skeleton node. In the former case, to generate the DFUDS bits, we must finish the bits corresponding to the degree encoding of the skeleton node and continue producing bits from the next available subtree. In the latter case, we perform analogously to generating the BP bits and start producing bits from the first ancestor t' of the subtree t which belongs to $S_D \cup S_U$. We focus on the latter case in the rest of the proof as the former case is a slight variation of the latter case and needs a straightforward phase in the beginning to finish reporting the degree as previously mentioned.

The index has an `up/down` field, which indicates we are traversing down or up the skeleton (*i.e.,* preorder number of t is less than that of v which is the rightmost child of the skeleton leaf). Let t' be the first ancestor of t which is in S_D or S_U (depending on the value of `up/down`). The index includes three pointers to locations in components P_D, P_U, and T_D (or T_U if traversing up) which correspond to node t'. In addition, the offset of the bit corresponding to t in the DFUDS encodings of the subtree rooted at t' in T_D or T_U is stored as a separate field in the index.

Since the size of each index is $\Theta(\log\log n)$ bits, the total size of such indices is $\Theta(\log n \log\log n)$, which is $o\left((\log n)^2\right)$. It only remains to show a block can be produced in constant time. To produce a block, we use the index associated with the starting position of the block. The key claim is that starting from node t', we can generate $\frac{\lg n}{6}$ bits of the DFUDS sequence.

Without loss of generality, assume the `up/down` field is set to `down` and t' the DFUDS sequence is traversing down the skeleton. The DFUDS bits following the bit corresponding to t' are stored in the three sequences: T_D,P_D, and P_U; these three sequences need to be somehow combined together. The main difference with the proof of lemma 4.3 is that both P_D and P_U bits are required as they together constitute the degree of skeleton nodes. Furthermore, the encoding of subtrees rooted at nodes of $S_D \cup S_U$ in T_D (and T_U) are the BP encodings, nevertheless we need the DFUDS encodings when combining the three sequences together. However, since we merge sequences by table look-ups, the conversion from the BP encoding to the DFUDS encoding is implicit and is precomputed and present in the table.

We perform the combination of the three sequences by a three-way table look-up. We form a table for all possible values of (t_d, p_d, p_u) where t_d, p_d, p_u are three bit vectors of size $\left\lceil \frac{\lg n}{6} \right\rceil$ each. There is only one copy of this table stored and shared among all mini-trees and has size $o(n)$. Using this look-up table, we can look-up on constant time, the next $\frac{\lg n}{6}$ bits of the DFUDS sequence. In the event, we reach the rightmost leaf of the skeleton leaf with these bits and change directions from downward traversal on the skeleton to an upward traversal, we break the bits to two segments and report each segment independently.

By reporting $\frac{\lg n}{6}$ bits starting from the bit corresponding to t', we are guaranteed to have covered the $\frac{\lg n}{24}$ bits which start from the bit of t, since the bits between t and t' is at most twice the size of a subtree which is at most $\frac{2\log n}{16} = \frac{\lg n}{8}$, and $\frac{\lg n}{6} - \frac{\lg n}{8} = \frac{\lg n}{24}$. □

By lemma 4.5, each block of length $\left\lceil \frac{\lg n}{24} \right\rceil$ bits can be produced in constant time. This implies that any word of length $\lceil \lg n \rceil$ bits can be generated in constant time:

Theorem 4.6. *The new unified representation supports operation* `DFUDS-word()` *in constant time in a skinny mini-tree using* $o\left((\log n)^2\right)$ *extra space.* □

General trees

We now consider the general case where the skeleton is an arbitrary tree and not just a path. In this case, we decompose the skeleton into $O(\lg n)$ paths using the following recursive procedure.

If the given subtree of the skeleton is a path, then return it as the only path of that subtree. Otherwise, find the maximal leftmost path of the skeleton subtree from the root to the leftmost skeleton leaf, and remove it. The remaining nodes of the subtree form a set of disjoint subtrees. Among these subtrees, we first identify all the "rightmost" subtrees, i.e., subtrees whose roots are the rightmost children of their parents, and remove the rightmost paths of each. For each of the disjoint subtrees thus obtained, we apply the decomposition algorithm recursively. Figure 4.4 shows the partitioning of a tree into these left-leaning and right-leaning paths.

This recursive decomposition produces $O(\lg n)$ paths since each leaf in the skeleton is associated with exactly one path, and the number of leaves in the skeleton is $O(\log n)$; as each skeleton leaf, by definition, has $\frac{\lg n}{16}$ descendants in the original tree which are disjoint from descendants of other skeleton leaves.

We associate each of the nodes of the mini-tree that is not part of the skeleton with its lowest ancestor that is in the skeleton. Thus the above procedure decomposes the mini-tree into $O(\log n)$ skinny trees (a tree whose skeleton is a path). We use the representation described in section 4.4.1 for skinny trees for each of the skinny trees.

We now show how each word of $\lg n$ bits long from the BP/DFUDS sequence can be produced in constant time in a general tree. As the tree is partitioned into skinny trees, the BP/DFUDS sequences are split into parts each of which is obtained from a skinny tree.

Definition 4.3. *We split the BP/DFUDS subsequences into maximal subsequences such that bits of each subsequence can be extracted from the representation of the same skinny-tree. We refer to these maximal subsequences by* skinny chunks*.*

We next argue that any $\frac{\lg n}{16}$-bit subsequence of the BP/DFUDS sequences consists of at most four skinny chunks:

Lemma 4.7. *Each* $\lg n$ *bit subsequence of the BP or the DFUDS sequences spans over* $O(1)$ *skinny chunks (definition 4.3).*

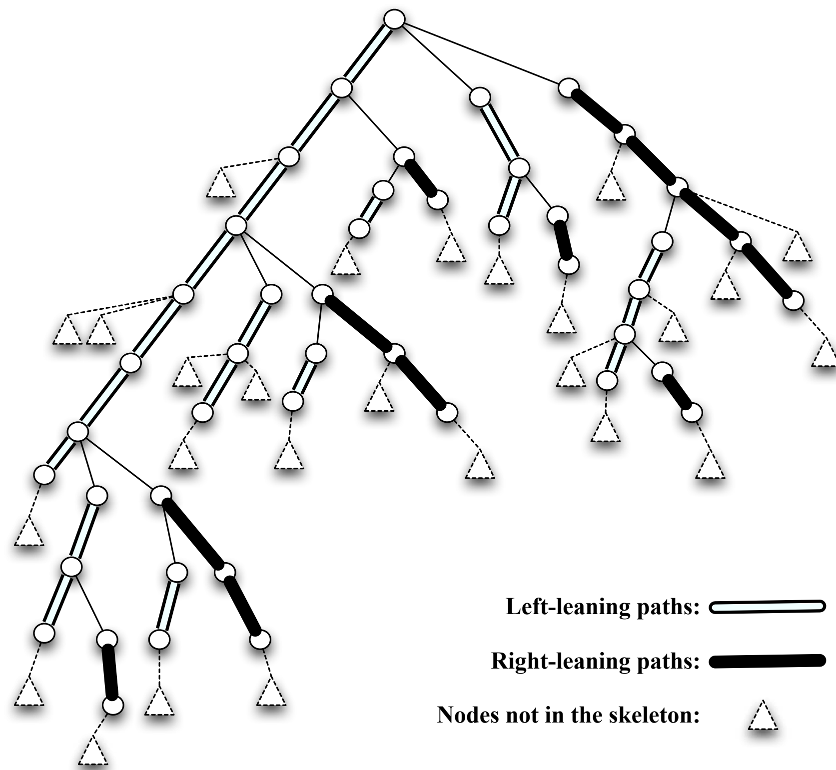


Figure 4.4: The skeleton of a tree is decomposed into left-leaning and right-leaning paths to partition the tree into skinny trees.

Proof. We prove that each $\frac{\lg n}{16}$ consecutive bits of the BP or the DFUDS sequences span over at most four skinny chunks. Having proved this, one derives the lemma by 16 applications of this claim.

We partitioned the skeleton of the tree into two types of paths: leaf-leaning and right-leaning paths. We define a skinny tree as left-leaning (or right-leaning) if its core path is left-leaning (or right-leaning respectively).

Both the BP and the DFUDS encodings are based on a depth first traversals, and $\frac{\lg n}{16}$ consecutive bits of each of these sequences start from a node and traverse up or down on a skinny tree. Depending on whether the skinny tree is a left-leaning or a right-leaning one and traversal is up or down the tree, we distinguish four cases:

On a left-leaning skinny tree, going down: The BP bits corresponding to all the nodes up to the leaf can be produced using the representations of P_D and T_D. The DFUDS bits of these nodes can be produced using the representations of P_D, T_D and P_U. Since the subtree size of the leaf is at least $\frac{\lg n}{16}$, at least the proceeding $\frac{\lg n}{16}$ bits of either BP or DFUDS sequence belong to the same skinny chunk.

On a right-leaning skinny tree, going down: As in the previous case, we can produce the BP/DFUDS bits using the representations of P_D, T_D and P_U till we either reach the leaf or the root of a left-leaning path. If we reach a skeleton leaf then the next $\frac{\lg n}{16}$ bits of the BP/DFUDS bits can be retrieved from the representation of the skeleton leaf subtree which is in T_D. If we reach a left-leaning path, we need to go down the path, and hence the case reduces to the previous case.

On a left-leaning skinny tree, going up: We can produce the BP bits using P_U and T_U (and DFUDS bits using only T_U) until we reach a child (left-leaning or right-leaning) path or the parent path or a sibling path. In all the cases, we need to explore a (child/parent/sibling) path going down, which is handled in the previous two cases.

On a right-leaning skinny tree, going up: As before, we can produce the BP/DFUDS bits using P_U and T_U till we reach the parent left-leaning path (or the root of the mini-tree, in which case, we can stop). We need to explore this parent left-leaning path going up which is handled in the previous case.

The transitions between skinny trees in the four cases above is depicted in figure 4.5.

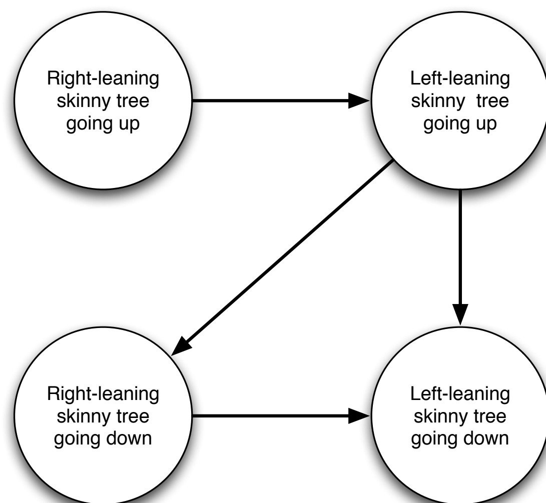


Figure 4.5: The transition diagram between left-leaning and right-leaning skinny trees going up or down corresponding to a $\frac{\lg n}{16}$-bit subsequence of the BP/DFUDS sequence.

Hence, we demonstrated that we switch at most three times between different skinny trees and therefore, there can be at most four skinny chunks which constitute any consecutive $\frac{\lg n}{16}$ bits of the BP or the DFUDS sequences. □

As a corollary to lemma 4.7, we can limit the number of skinny chunks in the entire BP/DFUDS sequences for the mini-tree:

Lemma 4.8. *The number of skinny chunks (definition 4.3) in the BP/DFUDS subsequences of a mini-tree is* $O(\log n)$*.* □

We are now ready to present the main result of this section that any $\lg n$ bit subsequence of the BP/DFUDS sequences can be reported in constant time:

Theorem 4.9. *Any subsequence of length* $\lg n$ *bits from the BP/DFUDS subsequences of a mini-tree can be reported in constant time using* $o\left((\log n)^2\right)$ *bits of extra storage.*

Proof. We build a FID structure (lemma 2.1) over the universe of $2n$ bits of the BP and the DFUDS sequences which indicates the starting points of the skinny chunks (definition 4.3). For each skinny chunk, we store a pointer to the representation of the corresponding skinny tree and the offset within the BP/DFUDS sequences of the skinny tree where the chunk

starts. The storage requirement of this dictionary is $o\left((\log n)^2\right)$ bits by lemma 2.1 and lemma 4.8.

Using the FID, for any skinny chunk c, we can produce the bits of the intersection of the BP or the DFUDS word and the chunk within the skinny tree in constant time by theorem 4.4 and theorem 4.6 respectively.

Lemma 4.7 states that at most a constant number of skinny chunks can intersect a word of length $\lg n$ bits, therefore by repeating the procedure for each chunk that interests the word, we discover the entire word in constant time. $\square$

4.4.2 Support for the tree covering representation

In previous sections, we showed how the new representation supports the BP and DFUDS representations as it can produce any word of the BP and DFUDS representation in constant time. This section will show how the new representation can also support the tree-covering representation on ordinal trees. The tree cover representation decomposes the tree into mini-trees as in section 3.3 using value $L = \lceil \lg^2 n \rceil$ and furthermore decomposes mini-trees using value $L = \lceil \frac{\lg n}{4} \rceil$ into micro-trees. Micro-trees are stored using references to a look-up table which lists all trees with size less than $\lceil \frac{\lg n}{4} \rceil$. In our new unified representation approach we decompose the entire tree to mini-trees identically, and therefore, the mini-trees in the unified representation are identical to the mini-trees in the tree-covering representation. The difference is that in the unified representation we do not decompose mini-trees further into micro-trees. Nevertheless, this section shows that we can generate micro-trees in the unified representations.

The main result of this section is that the new representation can produce the BP/DFUDS bit sequence of any micro-tree in constant time:

Lemma 4.10. *Within a mini-tree, the BP/DFUDS bit sequence of any micro-tree in the tree-covering representation can be produced in constant time using the new unified representation with an additional space of $o((\log n)^2)$ bits.*

Proof. An argument analogous to that in proof of lemma 4.1 proves that in the BP bit sequence corresponding to the nodes of a particular mini-tree are split into $O(1)$ consecutive subsequences whose concatenation yields the actual BP bit sequence of the micro-tree. For each micro-tree, we store $O(1)$ pointers of size $\Theta(\log\log n)$ that indicates the start of each

of the subsequences within the context of the mini-tree. We also store the length of the subsequence along with the pointers. These structures require $O(\log n \log\log n) = o\left((\log n)^2\right)$ bits.

To retrieve the BP/DFUDS bits of a micro-tree, starting points and the length of each subsequence is determined. The length of the BP/DFUDS sequence of a micro-tree is at most $\lg(n)/2$ bits and therefore, each subsequence is at most $\lg(n)/2$ bits long. Each subsequence is determined in constant time using theorem 4.9. These subsequences combined give the BP/DFUDS representation of the micro-tree. □

Using lemma 4.10, one can easily derive the mini/micro-tree representation:

Theorem 4.11. *The encoding of any micro-tree in the tree-covering representation can be determined in the new unified representation with an additional space of $o\left((\log n)^2\right)$ bits.*

Proof. The mini/micro-tree representation decomposes the tree into mini-trees which are further decomposed into micro-trees. Micro-trees are represented by indices to a look-up table which lists all trees of size less than $\left\lceil \frac{\lg n}{4} \right\rceil$.

In the new representation, the subtree decomposition into mini-trees is identical. Lemma 4.10 shows that, by using an additional $o\left((\log n)^2\right)$ bits, one can produce the BP/DFUDS bits corresponding any given micro-tree. We build a translation table which for all trees of size less than $\lg(n)/4$ maps the BP sequence (or analogously the DFUDS sequence) to the corresponding index in the look-up table which is used by the tree-covering representaion. This table requires $o(n)$ bits which is shared among all mini-trees in the entire tree. To determine a micro-tree, we generate its BP bits (or the DFUDS bits) using lemma 4.10 and use the translation table to determine the corresponding index in the look-up table. All other auxiliary data used in the tree-covering representation aside from representations of micro-trees require $o\left((\log n)^2\right)$ space and thus are replicated in the new representation. □

Theorems 4.9 and 4.11 imply directly the following grand result of this chapter:

Corollary 4.12. *Given an ordinal tree with n nodes, the unified representation uses $2n + o(n)$ bits and supports the BP, DFUDS, and TC representations by supporting operations* `BP-word(`k`)`, `DFUDS-word(`k`)`, *and* `TC-microtree(`m`,`μ`)` *in constant time.* □

4.5 Summary and discussion

In this chapter, we presented a universal succinct representation of ordinal trees. We introduced level order unary degree sequence (LOUDS), balanced parentheses (BP), depth first unary degree sequence (DFUDS), and tree covering (TC) methods as the main approaches in succinct representation of ordinal trees. We argued that the power of LOUDS approach is limited and it is intrinsically different from other approaches.

We gave a new universal succinct representation of ordinal trees which is able to simultaneously emulate BP, DFUDS, and TC representations. We argued that this representation is important and useful as it has the power of all the three representations combined; the queries supported in constant time in the universal representation is the union of queries supported in constant time in any of these representations.

The ability to simulate the LOUDS representation as well as the other three representations remains open. The intrinsic difference between LOUDS and other representations may make it impossible to have such a universal representation and in this event, a proof of impossibility of such universal representation is of interest.

Chapter 5

Succinct representations of graphs

In previous chapters, we studied succinct encoding of various families of trees. In this chapter, we study graphs as the target combinatorial object for the purpose of succinct representation. We consider representing arbitrary graphs to a given number of vertices and edges in a succinct manner to support (1) asking whether there is an edge between two given nodes and (2) scanning the edges incident with a given node. In comparing with the standard graph representations (1) adjacency matrix and (2) adjacency list, we combine the strength of each and avoiding their shortcomings. The adjacency matrix representation is near space-optimal for graphs with almost half of the "potential edges" present and it supports an edge existence query by examining a single bit. On the other hand, its $\Theta(n^2)$ space is unsuitable for sparse graphs and scanning the edges incident with a node takes $\Theta(n)$ bit operations independent of the number of edges. The adjacency list on the other hand is reasonably space efficient for sparse graphs and supports scanning the edges incident with a node ideally. On the other hand, it has a $\Theta(\lg n)$ factor space overhead for dense graphs and may require scanning through all edges incident with a given node to test for a single edge. We give a representation that is succinct and provides fast support for all these types of the queries.

5.1 Introduction

In this chapter, we study the problem of representing a given graph with n vertices and m edges with vertex labels from $[n] = \{1, \ldots, n\}$ to support adjacency, degree, and neigh-

borhood queries in constant time. We focus primarily on directed graphs, in section 5.5 however, we extend our results to undirected graphs. We assume that there are no multiple edges in the graph, that is there is at most one edge from vertex i to vertex j for each order pair i, j (edges (i, j) and (j, i) can be simultaneously present as can loop edges (i, i)).

Supporting these queries gives the functionality of both an adjacency matrix and an adjacency list to the encoding. An `adjacency` query on a pair of vertices (i, j) determines whether (i, j) is an existing edge. A `neighborhood` query has the functionality to iterate through vertices incident to a given vertex i ideally in constant time per neighbor. A crucial requirement is that the representation must be able to iterate through both the *in-neighbors* and *out-neighbors* of a given vertex (though we do not insist on any particular order on neighbors for either of these query types). Finally, the `degree` query is to output the in-degree and out-degree of a vertex ideally in constant time. Adjacency query is supported easily in constant time by an adjacency matrix but not an adjacency list. Conversely, neighborhood and degree queries are supported easily by an adjacency list but not an adjacency matrix. Our representation is succinct and supports `adjacency`, `neighborhood`, and `degree` query in constant time.

The storage requirement of such a representation is a key issue. A counting argument suggests that the information-theory lower bound for the space requirement of any representation which encodes graphs with n vertices and m edges is $\left\lceil \lg \binom{n^2}{m} \right\rceil$ bits. However, we prove a lower bound in section 5.3 that where $n^\delta < m < n^{2-\delta}$ for a constant $\delta > 0$, it is impossible to achieve this space to within lower order terms and support all the desired queries in constant time. Therefore, the best storage requirement has to be a multiplicative factor away from the information-theory bound. We match this lower bound by presenting an encoding in section 5.4 that requires $(1 + \epsilon) \lg \binom{n^2}{m}$ bits for an arbitrary small constant $\epsilon > 0$. In the case where $m < n^\delta$ for any constant $\delta > 0$, the space of the representation matches the information-theory lower bound tightly to within lower order terms. For extremely large values of m, where $m = \omega\left(\frac{n^2}{\sqrt{\log n}}\right)$, the space requirement of the representation again matches the information theory lower bound (this includes the entire range where $m > n^2/2$).

5.1.1 Related work

There is a large body of literature on space-efficient representations of graphs most of which deal with graphs with particular properties such as: graphs with limited arboricity, c-decomposable graphs [42], separable graphs [10], planar graphs [63, 43, 49, 13], triconnected and/or triangulated planar graphs [3]. In contrast to this body of work, we deal with graphs with no particular combinatorial properties, and of which we only know the number of vertices and edges.

Raman *et al.* [58] study the problem of indexable dictionary and as they observe a given graph can be represented by having a dictionary structure for each vertex containing the endpoints of all its incident edges. Using this approach to be able to report both in-neighbors and out-neighbors of a vertex in constant time per neighbor, we need to include in the dictionary of a vertex both its in and out neighbors. This potentially doubles the storage requirement and is space inefficient.

A problem closely related to representation of graphs is binary relations whose succinct encodings has been studied [6, 5]. Binary relations associate r objects to s labels such that objects can be associated to multiple labels and a particular label can be assigned to multiple objects. According to this definition a graph is essentially a binary relation from objects in $[n]$ to labels in $[n]$. The supported operations on relatives are rank and select on objects and labels (essentially row-wise and column-wise rank and select operations in a 0-1 matrix). It is not hard to verify that we can implement adjacency, neighborhood, and degree queries using these operations. Their representation requires $r \lg s + o(r \log s)$ bits which is space optimal only in the sparse case and even in such cases the running time of operations is dependent on the value of r or s and therefore the operations do not perform in constant time. Golynski [27] proves that any representation with $r \lg s + o(r \log s)$ bits cannot perform these operations in constant time. Nevertheless, the operations we require on graphs are weaker than the full rank and select operations and therefore, the infeasibility results do not hold for our purposes.

5.2 Preliminaries

We view the problem of encoding a graph with n vertices and m edges compactly as encoding its adjacency matrix. Given 0-1 matrix A of size $n \times n$ containing m ones, we wish to represent the matrix succinctly so it supports the `access` and `successor` queries. The adjacency query in the graph corresponds to `access` query in the matrix and the neighborhood query corresponds to `successor` query. The degree query in the graph corresponds to reporting the number of ones in a given row or column.

Definition 5.1. *Given a 0-1 matrix $A[n,n]$, the query* `access(i,j)` *simply reports the value of $A[i,j]$.* `r-successor(i,0)` *gives the position of a one in row i (if there is one).* `r-successor(i,j)` *(where $A[i,j]=1$) gives the entry one in row i that "follows" the one in position (i,j). By "follows" we permit the encoding to iterate through ones in row i in any order.* `c-successor(.,.)` *is defined identically on columns.* `successor(r/c,.,.)` *query encapsulates* `r-successor` *and* `c-successor` *queries as its first parameter is a switch with values r or c:* `successor(r, i, j) = r-successor(i, j)`*, and* `successor(c, i, j) = c-successor(i, j)`*.*

At the heart of our representation sits a succinct encoding of functions allowing access, select, and partial rank in constant time.

Definition 5.2. *Given a function $f : [d] \rightarrow [r]$,* `access` *operation is, given $i \in [d]$ to compute $f(i)$. We refer to this as the forward navigation capability of the function. The* `select(i,j)` *operation applied to f determines the i-th smallest element k such that $f(k) = j$. We can use this operation to report the set $f^{-1}(j)$ in constant time per element for any given j; we refer to this as the reverse navigation capability of the function. The* `partial_rank(i)` *operation is to determine the rank of i among all elements k such that $f(k) = f(i)$.*

The previously existing succinct representation of functions [53] does not support `select` and `partial_rank` in constant time. The representation supports reporting $f^{-1}(i)$ in time $O(1 + |f^{-1}(i)|)$, however, this operation is much weaker than supporting `select` and `partial_rank` in the corresponding set $f^{-1}(i)$.

The problem of representing a function $f : [d] \rightarrow [r]$ supporting the described set of operations is equivalent to representing a string S of length d over an alphabet of size r. The equivalent operations are string access (to report S[i]), string select (to determine the

location of the i-th occurrence of a symbol of the alphabet), and partial rank (given a location i, to determine the number of symbols of type $S[i]$ preceding location i). Golynski *et al.* [30] studies a stronger version of the problem where instead of `partial-rank`, we require support for `full_rank` (given any position i, to determine the number of occurrences of a symbol of any given type before i). Their space matches the information theory bound of $d \lg r$ to within lower order terms, however the operations do not perform in constant time. It is not possible to support these operations in constant time even if we relax the space bound to $(1+\epsilon)d \lg r$ for any constant $\epsilon > 0$ as these operations clearly provide support in constant time for the predecessor search problem which is proven impossible with this much space [7]. However, we obtain constant time support for weaker operations of `partial-rank`, `select`, and `access` with $(1+\epsilon)d \lg r$ storage requirement for any constant $\epsilon > 0$ by modifying the representation of [30].

Theorem 5.1. *Given a function $f : [d] \to [r]$, there is a succinct encoding of f which requires $(1+\epsilon)d \lg r + O(d)$ bits of space for any constant $\epsilon > 0$ and supports* `access`*,* `select`*, and* `partial-rank` *(as defined in definition 5.2) in constant time $O(1/\epsilon)$.*

Proof. We first focus on the main case where $d \geq r$ (the $d < r$ case will be discussed subsequently). The main idea in the proof, as in Golynski *et al.* [30], is to split the domain $[d]$ into blocks of size r. They build structures with total size $O(d)$ bits to support, in constant time, rank and select at the boundaries of the blocks; these operations are referred to as `rank-b` and `select-b` [30]. The second step is to support rank and select locally within blocks (`rank-l` and `select-l` in [30]). Constant time support for rank and select across blocks and constant time support for partial-rank and select and access within individual blocks results in constant time support for partial-rank, select, and access over the entire function.

We now build structures to support `access`, `select`, and `partial-rank` within blocks in constant time. Within a block, we form a list L which includes for each i in increasing order, elements j (in increasing order) such that $f(j) = i$. This yields a permutation of the block. A bit vector $X = 1^{l_1}01^{l_2}0 \ldots 1^{l_r}0$ is formed where $l_i = |f^{-1}(i)|$ where f^{-1} is confined to within the block. This structure is stored in $O(d)$ bits using the fully indexable dictionary of lemma 2.1. Thus far, the structures are identical to those in [30]. The new representation deviates from [30] in how the permutation is represented. As we can afford

to have space within $1+\epsilon$ factor away from the information theory lower bound, we use the $(1+\epsilon)k \lg k$-bit representation of a permutation $\pi : [k] \rightarrow [k]$ by Munro *et al.* [48] which supports operations $\pi(), \pi^{-1}()$ in constant time depending on ϵ,*i.e.,* $(O(1/\epsilon))$. Operations `access` and `select` in the block can be supported using a constant number of applications of operations π, π^{-1} in the permutation and rank and select on zeros and ones in X as shown by Golynski *et al.* [30]. Operations π, π^{-1} are supported in constant time in our representation and rank and select operations on zeros and ones in the fully indexable dictionary of X is performed in constant time by lemma 2.1. Hence, `access` and `select` in a block are supported in constant time. Golynski *et al.* [30] support the (full) rank operation and therefore build some auxiliary structures. Since we need only support for `partial-rank`, no auxiliary structure is formed or stored. `Partial-rank` can be supported using a constant number of applications of operations π, π^{-1} in the permutation and rank and select on zeros and ones in X as follows. Given i, j such that $f(i) = j$ to determine the rank of i among all those k's such that $f(k) = j$, we use the π^{-1} operation to determine the rank of i in list L. We use rank and select on zeros in X to determine the starting position of elements of $f^{-1}(j)$ in the list and use rank on ones to compute the offset of i from that position. The offset computed is the partial-rank. Hence, `partial-rank` is performed in constant time as well in a block.

The required operations of `access`, `select`, and `partial-rank` are supported in constant time and the storage requirement for the structures maintained is $(1+\epsilon)d \lg r + O(d)$ bits.

The case where $d < r$ can be handled as follows. Since the entire range $[r]$ of f cannot be covered, we build an indexable dictionary structure (lemma 2.2) to represent the set $R' \subset [r]$ which consists of elements $j \in [r]$ such that there exists an i such that $f(i) = j$. We define $r' = |R'|$ and reduce the range from $[r]$ to $[r']$ remembering the correspondence between elements of these two sets (using the ID) and define function f' as the equivalent of function f on the new range ($f'(i) = j$ if and only if $f(i) = j$ and $j \in [r]$ corresponds to $j' \in [r']$.

Function $f' : [d] \rightarrow [r']$ has the property that $d \geq r'$ and therefore can be represented as described in the previous case in $(1+\epsilon)d \log r' + O(d)$. The indexable dictionary uses $\lg \binom{r}{r'} + o(r')$ bits by lemma 2.2. Since $r' \leq d < r$, the total number of bits required is at most $(1+\epsilon)d \lg r + O(d)$. All operations on function f can be supported using the representation of function f' with an extra level of indirection through the indexable dictionary structure. $\square$

5.3 Space lower bounds

For the purpose of proving lower bounds, we assume the cell probe model in which the size of a word is $w = O(\log n)$ bits. The cell probe model is a model of computation in which the computation time is measured by the number of probes to a random access machine with words of size w [46]. For all values of n, m, we prove lower bounds as a function of n and m on the worst-case space requirement of representations of $n \times n$ 0-1 matrices containing m ones supporting `access` and `successor` queries as described in definition 5.1.

There is a trivial information theory bound which comes directly from a counting argument and it holds for any representation regardless of queries it can support:

Theorem 5.2. *Any representation of $n \times n$ 0-1 matrices containing m ones requires at least $\lg \binom{n^2}{m}$ bits for some matrices.* □

The main purpose of this section is to show that for a reasonably large range of values of m in comparison to n (namely for values $n^\delta < m < n^{2-\delta}$ for an arbitrary small constant δ), it is infeasible to achieve the information theory bound in theorem 5.2 to within lower order terms while supporting the desired queries in constant time. This implies that in this range, the space requirement must be a constant multiplicative factor (larger than one) away from the information theory bound. We give a representation in section 5.4 that has this multiplicative factor arbitrarily close to one and tight to $1 + \epsilon$ for any constant $\epsilon > 0$.

We distinguish two cases depending on relative values of n, m and consider them separately: (1) the moderate case: $m = \Omega(n)$ and $m = O\left(n^{2-\delta}\right)$ for any constant $\delta > 0$, and (2) the sparse case: $m = o(n)$. In either case, we give matching upper bounds in section 5.4. A discussion is in order in section 5.6 for the remaining range of values of m which are extremely close to n^2.

Table 5.1 summarizes the space lower bound and upper bounds we show in this chapter for various ranges of values of m relative to n.

5.3.1 Lower bound for the moderate case

This section proves lower bounds for the moderate case where $m = \Omega(n)$ and $m = O\left(n^{2-\delta}\right)$ for a constant $\delta > 0$. We use ideas from Golynski [27], who proved an infeasibility result for succinct representation of permutations. Specifically, he proved that not all permutations

$\mathbf{m}$	Space Lower bound	Space Upper bound
$\forall\delta > 0;\ \mathbf{m} < n^{\delta}$	$\lg\binom{n^2}{m}$	$\lg\binom{n^2}{m}$
$\exists\delta > 0;\ n^{\delta} < \mathbf{m} = o(n)$	$(1+\epsilon)\lg\binom{n^2}{m}$	$(1+\epsilon)\lg\binom{n^2}{m}$
$\exists\delta > 0;\ \Omega(n) = \mathbf{m} < n^{2-\delta}$	$(1+\epsilon)\lg\binom{n^2}{m}$	$(1+\epsilon)\lg\binom{n^2}{m}$
$\forall\delta > 0;\ n^{2-\delta} < \mathbf{m} = O\left(\frac{n^2}{\sqrt{\log n}}\right)$	$\lg\binom{n^2}{m}$	$(1+\epsilon)\lg\binom{n^2}{m}$
$\omega\left(\frac{n^2}{\sqrt{\log n}}\right) = \mathbf{m} \le n^2$	$\lg\binom{n^2}{m}$	$\lg\binom{n^2}{m}$

Table 5.1: Space lower and upper bounds for representing a directed graph with n vertices and m edges which supports the queries in constant time. All the upper bounds are up to lower order terms.

$\pi : [n] \to [n]$ can be encoded in the information theory optimal bound of $n\lg n$ bits to within lower order terms supporting π and π^{-1} in constant time. In fact, this result directly translates to a lower bound for our problem where $m = n$. Therefore, our lower bound is, in a sense, an extension of that work.

Theorem 5.3. *If $m = \Omega(n)$ and $m = O\left(n^{2-\delta}\right)$ for a constant $\delta > 0$, there is an $n \times n$ 0-1 matrix containing m ones which cannot be encoded in $\lg\binom{n^2}{m} + o\left(\lg\binom{n^2}{m}\right)$ bits supporting the* `access` *and* `successor` *queries as described in definition 5.1 in constant time.*

Proof. Assume such structure S exists in the cell probe model for any such 0-1 matrix M which supports the queries in constant time with the stated number of bits. We derive a contradiction by showing an encoding for M (and therefore, all 0-1 matrices containing m ones) in less than $\lg\binom{n^2}{m}$ bits.

Structure S works in the cell probe model with $\Theta(\log n)$-bit cells and requires $\lg\binom{n^2}{m} + o\left(\lg\binom{n^2}{m}\right) = m\lg(n^2/m) + o\left(m\lg(n^2/m)\right)$ bits, and equivalently $m\lg_n(n^2/m) + o\left(m\lg_n(n^2/m)\right) = O(m)$ cells. We show that a constant fraction of these cells can be safely deleted such that the remains of the structure still describes the original matrix uniquely.

On query q (a successor or access), we define $\mathcal{C}[q]$ as the set of cells that are probed in order to compute the response to the query. For entry (i, j), where $M[i, j] = 1$, we define

query $q(i,j)$ as the union of access, row successor, and column successor queries on entry (i,j). Hence, $\mathcal{C}[q(i,j)] = \mathcal{C}[\texttt{access}(i,j)] \cup \mathcal{C}[\texttt{successor}(r,i,j)] \cup \mathcal{C}[\texttt{successor}(c,i,j)]$.

As discussed in definition 5.1, `successor(r/c,i,j)` queries where $i=0$ or $j=0$ provide the initial one entry in a row or a column on which successive applications of `successor` query can be applied to iterate through ones. For the sake of comprehensiveness, we add row and column number zero to the matrix and assume the row and column entirely consist of ones, *i.e.,* entries $M(i,j)$ where $i=0$ or $j=0$ are ones.

Consider all sets $\mathcal{C}[q(i,j)]$ for all entries (i,j) which contain a one in the matrix ($M(i,j) = 1$). There are $m+2n-1 = \Theta(m)$ such sets, each containing a constant number of cells. Since, there are $O(m)$ cells, there are at least a constant fraction of cells which occur in at most r of the sets (for r a suitable large constant). We denote by C_r the set of all such cells which appear in at most r sets ($|C_r| = \Omega(m)$). The goal is to select a large subset $\mathcal{D} \subset C_R$ of these cells to remove from the structure S in such a manner that the matrix is constructible given the rest of the cells.

We initialize $\mathcal{D}$ to empty and augment it by adding cells from C_r in an incremental fashion. We maintain some invariants which allow us to recover the matrix after deletion of cells in $\mathcal{D}$. The first invariant is that for any matrix entry (i,j) such that $M[i,j] = 1$, $\mathcal{C}[q(i,j)] \cap \mathcal{D}$ is labeled with no or at most a single element. Therefore, we can label each one entry in the matrix with $\mathcal{C}[q(i,j)] \cap \mathcal{D}$ such that each entry is labeled either empty or a single cell. Furthermore, we maintain the invariant that if an entry (i,j) is labeled with c, then its predecessor and successor in row and column i as well as its predecessor and successor in row and column j (if they exist) are assigned no label or labeled with c.

We remove an arbitrary cell $c \in C_r$ and include it in $\mathcal{D}$. If $c \in \mathcal{C}[q(i,j)]$ for any (i,j) and $(a,b),(a',b')$ are the predecessor and successor to (i,j) in row/column i (if they exist) and $(c,d),(c',d')$ are the predecessor and successor to (i,j) in row/column j (if they exist), we remove all elements of $\mathcal{C}[q(i,j)]$, $\mathcal{C}[q(a,b)]$, $\mathcal{C}[q(a',b')]$, $\mathcal{C}[q(c,d)]$, and $\mathcal{C}[q(c',d')]$ from C_r, and add another arbitrary cell from C_r to D and continue in this greedy manner. Since $\mathcal{C}[q(i,j)]$ has constant size for all i,j, at each step we remove a constant number of cells from C_r, and hence, the number of cells added to $\mathcal{D}$ is a constant fraction of C_r.

We can verify that the deletions from C_r maintain the invariants throughout the procedure; firstly, immediately after removal of a cell c, we remove from $\mathcal{D}$, the entire content of $\mathcal{C}[q(i,j)]$ where $c \in \mathcal{C}[q(i,i)]$. Hence, a one entry (i,j), $\mathcal{C}[q(i,j)]$ can share at most one

cell with $\mathcal{D}$, and therefore the first invariant is maintained. Secondly, upon removal of a cell c, we remove row and column predecessors and successors of any entry (i, j) labeled with c. Thus, no subsequent one entries in a column or in a row can have the same label. This implies the second invariant is enforced.

We are now ready to delete cells of $\mathcal{D}$ from structure S by adding some auxiliary structures. We form a bit vector `deleted_cells` of length $O(m)$ bits corresponding to the cells of structure S in order; its bits are one if the corresponding cell belongs to $\mathcal{D}$ (*i.e.,* if deleted) and zero otherwise. For a cell $c \in \mathcal{D}$, we refer to a one matrix entry (i, j) as *affected by* c, if $c \in \mathcal{C}[q(i, j)]$. We note that only a constant number (r) of entries are affected by c. We refer to a row or a column as affected by c, if they contain an entry affected by c. For each $c \in \mathcal{D}$, the projected matrix on affected rows and columns by c is stored explicitly (each matrix requires a constant number of bits less than r^2). Finally the cells of $\mathcal{D}$ are removed from S and remaining cells are made compact and stored consecutively.

A constant fraction of the cells $(\Theta(m))$ are deleted $(\Theta(m \log n)$ bits) and auxiliary structures stored require $O(m)$ bits. Therefore, the total size of the final structure is a constant (less than one) fraction of $\lg \binom{n^2}{m}$. The contradiction is derived from the fact that the matrix is fully recoverable from the final structures and thus all matrices can be encoded uniquely with a storage requirement which is less than the information theory bound.

We now argue that the matrix is indeed recoverable from the structures stored. Given the `deleted_cells` vector and the retained cells of the structures, one can simulate a query on the original structure S. The only issue arises where a query *fails* as it wants to probe a cell in $\mathcal{D}$ which is deleted. To recover the matrix given these structures, we exhaustively perform all `access` queries over all possible entries (i, j) and fill in the matrix where the queries succeed. We now simulate all `successor` queries on previously discovered ones in the matrix repeatedly to exhaust all one entries these queries can recover. We form an affected list for all rows and columns. These lists are initially empty, and if a `successor` query fails on an entry (i, j) as it needs access to a deleted cell c, we add c to the affected list of cells of row i and column j.

The invariants guarantee that each row/column completely discovers the set of cells by which it is affected; since, these cells are the labels which are assigned to the one entries in the row/column, and the second invariant guarantees different labels are separated by empty labels within an individual row or column. Positions with no labels can be discovered by

`access` queries, and by repeated applications of *successor* queries, all labels are discovered. Furthermore, by the first invariant, each entry can have a single label and therefore, all cells that affect rows and columns are exposed. For each cell $c \in \mathcal{D}$, we fill in the sub-matrix projected on the rows and columns affected by c, using the explicitly stored sub-matrix for c.

We now argue that the matrix is fully recovered correctly. Since we only consult the remaining cells and the explicitly stored sub-matrices, we do not place a one in a wrong position. We need only argue that all ones in the matrix are indeed discovered in the recovery procedure. For the sake of contradiction, we assume there is an edge (i, j) that is not discovered. In the original matrix, this entry cannot be unlabeled as it would be discovered by an `access` query otherwise, and thus is labeled by a cell c (and not by more than a cell due to the first invariant). We show the recovery procedure realizes that row i is affected by cell c. We denote by (i, j') as the predecessor one to (i, j) in row i. Entry (i, j') must have a label, as no label means it is discoverable by `access` query and moreover its successor (i, j) is also discoverable which is a contradiction. If entry (i, j') is discovered by the recovery procedure, then the procedure must stop short of discovering (i, j) as it needs to access to a cell which must be c due to the first invariant. Therefore, the recovery procedure realizes row i is affected by cell c. If entry (i, j') is not discovered during the recovery, we repeat the same argument on the predecessor (i, j'') to entry (i, j'). Since entry zero ($M[i, 0] = 1$) is given and readily discovered by the recovery procedure at some stage. The repetition of the argument must halt at some point which shows that the recovery procedure realizes that row i is affected by cell c.

Similarly, we show that the recovery procedure learns that column j is affected by cell c. Finally, since both row i and column j are realized affected by c, the entry (i, j) is consulted from the explicitly stored sub-matrix associated with c and therefore is discovered which is a contradiction. □

5.3.2 Lower bound for the sparse case

In this section, we give a lower bound for the case where $m = o(n)$. We break the range into two subranges and consider them individually.

The first range is $\forall \delta > 0;\ m < n^{\delta}$. For such values of m, the space lower bound is simply

the information theory space lower bound of $\lg \binom{n^2}{m}$.

For values of $m = o(n)$ where there exists a constant $\delta > 0$ such that $n^\delta < m$, we prove a space lower bound stronger than the information theory bound. The proof is a modification of lower bound result of Golynski [27] for representing permutations:

Theorem 5.4. *If there exists a constant $\delta > 0$ such that $m > n^\delta$ and $m = o(n)$, there is a $n \times n$ boolean matrix containing m ones whose encoding requires at least $(1+\epsilon)\lg \binom{n^2}{m}$ bits for some constant $\epsilon > 0$, if the encoding supports the* `access` *and* `successor` *queries as described in definition 5.1 in constant time.*

Proof. Since there exists a constant δ that $m > n^\delta$ and $m = o(n)$, we have that $\log m = \Theta(\log n)$. We consider only a special subset of all such matrices which are matrices that have at most a single one entry in each row and column, and show there exist at least a matrix in this subset that requires at least $(1+\epsilon)\lg \binom{n^2}{m}$ bits. Assume for the sake of contradiction that all such matrices can be encoded in space at most $\lg \binom{n^2}{m}$ bits to within lower order terms (*i.e.*, $\epsilon = 0$ in the bound of the statement of the theorem). We prove the lower bound in cell probe model with cells of $\Theta(\log n)$ bits. The encoding consists of $\lg \binom{n^2}{m} / \Theta(\log m) = \Theta(m)$ cells.

The contradiction is achieved in the same manner as the proof of theorem 5.3 in that repeatedly cells are deleted such that the matrix is recoverable from the remaining cells. In the end, we demonstrate that the storage requirement of the remaining cells together with auxiliary structures built during the elimination procedure is less than what the information theory suggests for the space of such matrix which is impossible.

For $i = 1, \ldots, n$, we denote by r_i the query `successor(r,i,0)`, and by c_i the query `successor(c,0,i)` (as defined in definition 5.1). The answer to queries r_i (respectively c_i) is empty for $n - m$ rows (respectively columns) and is non-empty for m rows (respectively columns); these are rows (respectively columns) which contain a one. If $M[i,j] = 1$, then answer to both r_i and c_j is entry (i,j) and we define queries r_i and c_j as reciprocal of each other. We define set T as the set of queries r_i or c_j which have a reciprocal ($|T| = 2m$).

For a query q, define $\mathcal{C}[q]$ as the set of cells one needs to probe to respond to the query. Since r_i's and c_i's perform in constant time, for all i, $|\mathcal{C}[r_i]| = |\mathcal{C}[c_i]| = O(1)$. There are m queries in set T, and each probes a constant number of cells, therefore there is at least a constant fraction of cells which occur in at most r of sets $\mathcal{C}[t]$ such that $t \in T$ (for a suitable

large constant r); we denote this set of cells as C_r ($|C_r| = \Theta(m)$). Furthermore, there are $2n$ queries r_i's and c_i's and each probes a constant number cells, therefore there is at least a constant fraction of cells of C_r such that they are probed by at most $O(n/m)$ of these queries; we denote this subset of C_r as C_t ($|C_t| = \Theta(m)$).

The goal is to select a large subset $\mathcal{D}$ of cells from C_t. We initialize $\mathcal{D}$ to empty and augment it by adding cells from C_t in an incremental fashion. We remove an arbitrary cell $c \in C_t$ and include it in $\mathcal{D}$. if $c \in \mathcal{C}[r_i]$ (or $c \in \mathcal{C}[c_j]$) for any i (or j), and $r_i \in T$ (respectively $c_j \in T$), we mark cells of $\mathcal{C}[q]$ as well as cells of $\mathcal{C}[c_i]$ (or $\mathcal{C}[c_i]$) as unremovable where q is the reciprocal to r_i (respectively c_j). If $\mathcal{C}[q]$ also contains cell c, we form an auxiliary structure `reciprocal-index` consisting of two pointers which store the indices of queries r_i (or c_j) and q; the key fact is that the index is not from a universe of size $O(n)$ but from one of size $O\left(\frac{n}{m}\right)$. We remind that at most n/m of $r_1, \ldots, r_n$ and n/m of $c_1, \ldots, c_n$ probe cell c, and the indices are kept from this universe (those that probe c).

We now select another arbitrary cell $c' \in C_t$ which is not marked so far as unremovable and add it to $\mathcal{D}$ and repeat. Since, by including a cell in $\mathcal{D}$, we mark only a constant number of cells as unremovable, the procedure iterates for a constant fraction of m times and therefore $\mathcal{D} = \Theta(m)$. We now delete cells of $\mathcal{D}$ from the structure by adding the bit vector `deleted_cells` of length $O(m)$ bits which has a bit representing each cell and the bit is set if the cell belongs to $\mathcal{D}$.

As many as $\Theta(m)$ cells are deleted (each of length $\Theta(\log n)$ bits) and for each deleted cell, we explicitly store a structure of size $O\left(\log \frac{n}{m}\right)$ bits, and the `deleted_cells` structure requires $O(m)$ bits. Therefore, the net reduction of space is $O(m \log m) = O(m \log n)$ bits. We show that the matrix is uniquely recoverable from the remaining cells together with added structure, and thus, we have a contradiction as the reduction in space implies that any such matrix can be encoded in less space than the information theory space lower bound.

To demonstrate that the matrix is recoverable from the remaining cells, we need only show we can discover all one entries of the original matrix (the rest of the matrix is filled out with zeros). Using the `deleted_cell` structure and the remaining cells, we can simulate r_i, c_i queries and fill in the matrix by ones. The only issue is when a query (say r_i) fails as it needs access to a deleted cell. We show that if there exists a one in row i, we can discover its position. Suppose entry (i, j) in that row is a one; query r_i either reports this fact or fails as it needs access to a deleted cell. Similarly, either c_j completes, or it stops by needing

access to a deleted cell. Suppose cell c is the earliest cell deleted in $\mathcal{C}[r_i] \cup \mathcal{C}[c_j]$. In the cell elimination procedure, upon deleting cell c, the set of cells in the reciprocal query (r_i or c_j) become unremovable and therefore the reciprocal query can complete and report entry (i, j) as a one. The only possibility is that c is in both $\mathcal{C}[r_i]$ and $\mathcal{C}[c_j]$ and therefore neither of the queries r_i or c_j completes. However, in this case we built structure `reciprocal-index`, and among all queries $r_1, \ldots, r_n$ and $c_1, \ldots, c_n$ that do not complete by needing access to c, the indices i and j can be retrieved from the pair of pointers stored and therefore matrix entry (i, j) can be recognized as a one entry. □

Theorems 5.3 and 5.4 together imply the grand result of the lower bound section:

Corollary 5.5. *If $n^\delta < m < n^{2-\delta}$ for some constant $\delta > 0$, there exists an $n \times n$ boolean matrix containing m ones whose encoding requires at least $(1+\epsilon)\lg\binom{n^2}{m}$ bits of storage requirement for some constant $\epsilon > 0$ to support the* `access` *and* `successor` *queries as described in definition 5.1 in constant time.*

5.4 Upper bound: the representation

Given the abstraction of an $n \times n$ 0-1 matrix A containing m ones, this section shows how to represent the matrix succinctly so it supports the `access`, `successor`, and `degree` queries in constant time.

Depending on the density of the matrix, we distinguish four cases which we discuss in four subsequent sections:

1. Almost-full: $m > n^2 - o(n)$.
2. Extremely dense: $n^2 - \Omega(n) \geq m > n^2\left(1 - O\left(\frac{1}{\sqrt{\log n}}\right)\right)$.
3. Dense: $n^2\left(1 - \omega\left(\frac{1}{\sqrt{\log n}}\right)\right) \geq m = \omega\left(\frac{n^2}{\sqrt{\log n}}\right)$.
4. Moderate: $O\left(\frac{n^2}{\sqrt{\log n}}\right) = m > \frac{n}{2}$.
5. Sparse: $\frac{n}{2} \geq m$.

We devise a universal scheme to show support for `degree` queries in constant time. We store two variable-length-field arrays `r[1..n]` and `c[1..n]` that encode the row degrees and column degrees respectively. Each entry of these arrays encodes in binary the degree of the corresponding row or column. Where $m > \frac{n^2}{2}$, we store the row and column degree of the complement of the matrix. Therefore the total length of these two arrays is $L = \min \Theta(m+n), \Theta(n \log n), n^2 - m + n$ since each entry is at most $O(\log n)$ bits long and entries sum to m and there are n entries. L is a lower order term of space in representations except where $m = o(n)$ or $m = n^2 - o(n)$. The former and latter case coincide with the sparse case and almost-full case respectively. In sections 5.4.1 and 5.4.5, we demonstrate how `degree` query is supported in constant time in these cases.

5.4.1 The almost-full case

We denote the number of zero entries in the matrix by m'. In the almost-full case we have $m' = o(n)$. We define a row/column as zero-containing if there is at least one zero entry in the row/column and denote by r and c, the number of zero-containing rows and columns respectively.

From the universe of all rows, the r zero-containing rows are represented using the ID structure (lemma 2.2) using $r \lg \frac{n}{r} + o\left(r \lg \frac{n}{r}\right) \leq m' \lg \frac{n}{m'} + o\left(m' \lg \frac{n}{m'}\right)$ bits and similarly from the universe of all columns, the c zero-containing columns are representing using the ID structure using $c \lg \frac{n}{c} + o\left(c \lg \frac{n}{c}\right) \leq m' \lg \frac{n}{m'} + o\left(m' \lg \frac{n}{m'}\right)$ bits.

We denote by M' the $r \times c$ submatrix which is formed by zero-containing rows and columns. We explicitly store values r and c and add $m'-r$ rows and $m'-c$ columns consisting merely of one entries to the bottom and right ends of matrix M' to make M' a $m' \times m'$ matrix. M' has exactly m' zeros and size $m' \times m'$, therefore this matrix classifies as an extremely-dense matrix. We use the representation for extremely dense matrices of section 5.4.2 to encode M'. This representation requires $\lg \binom{m'^2}{m'} + o\left(\lg \binom{m'^2}{m'}\right) = m' \lg m' + o(m' \log m')$ bits and support `access`, `successor`, and `degree` queries in constant time.

The total space requirement of our representation calculates as follows (where $m' = o(n)$):

$$\begin{aligned}
2m' \lg \frac{n}{m'} + o\left(m' \lg \frac{n}{m'}\right) + m' \lg m' + o\left(m' \log m'\right) &\leq m' \lg \frac{n^2}{m'} + o\left(m' \lg n\right) \\
&= \lg \binom{n^2}{m'} + o\left(\lg \binom{n^2}{m'}\right) \\
&= \lg \binom{n^2}{m} + o\left(\lg \binom{n^2}{m'}\right).
\end{aligned}$$

It remains only to show how the queries are supported in constant time using our representation. To answer `degree` queries, it only suffices to check the ID structures to determine whether the row/column is zero-containing and in that event, we use the support for `degree` query in matrix M'. To support `access` and `successor` queries we note that one can translate an entry position in the original matrix M to the corresponding position within sub-matrix M' and vice versa using the `rank` and `select` query capability of ID structures for zero-containing rows and columns. Query `access` can be answered by using the ID structures to determine whether the entry belongs to M' or not. If not, then the entry is one and if so, we use `access` query for matrix M' to answer.

Support for `successor` queries is more complicated. We form two bit vectors `R` and `C` of length m' as follows. Positions of `R` correspond to zero-containing rows and a position is set to one if the corresponding row has an adjacent row which is not zero-containing. Bit vector `C` is formed over zero-containing columns analogously. Vectors `R` and `C` are encoded using the FID structure (lemma 2.1) and therefore they use only $O(m')$ bits of space which is negligible as it is $o\left(\lg \binom{n^2}{m}\right)$.

To answer `r-successor` queries (row successors) on position (i, j) we check the immediate neighbor entry $(i, j+1)$ and if it is contains a one rather than a zero, we are done. Otherwise, if $(i, j+1)$ contains a zero, it must belong to matrix M' in which case, we translate the position to within M' (i', j') and obtain the `r-successor` position. If the position is at a column numbered larger than c (*i.e.,* if it belongs to a pad column), we set the column number to $+\infty$. The position is translated back to a position (i, j_1) in matrix M. The key issue is that this position is not necessarily the first available one entry following (i, j) as there could be a column not belonging to M' between j and j_1 which entirely consists of ones, and in this case, we must report first such column. We consult the FID structure for `C` to determine a

successor j_2' one entry to position j' which indicates the end of a block of zero-containing columns. Subsequently, we translate j_2' in matrix M' to position j_2 in M. The answer to `r-successor` query is $(i, \min\{j_1, j_2+1\})$. `c-successor` queries are supported in constant time analogously:

Theorem 5.6. *A 0-1 matrix of size $n \times n$ with $m > n^2 - o(n)$ ones can be represented in $\lg \binom{n^2}{m} + o\left(\lg \binom{n^2}{m}\right)$ bits supporting* `degree`*,* `access` *and* `successor` *queries in constant time, if any $n \times n$ matrix with $m < n^2 - \Omega(n)$ ones can be represented so.* □

5.4.2 The extremely-dense case

Where $n^2 - \Omega(n) \geq m > n^2\left(1 - O\left(\frac{1}{\sqrt{\log n}}\right)\right)$, we traverse the matrix in a row-major fashion and create a universe of size n^2. In this universe we represent the zero entries, of which there is $\Omega(n) = m' = O\left(\frac{n^2}{\sqrt{\log n}}\right)$, using the ID structure of theorem 2.2. The storage requirement of the structure is

$$\lg \binom{n^2}{m'} + o\left(\lg \binom{n^2}{m'}\right) = \lg \binom{n^2}{m} + o\left(\lg \binom{n^2}{m}\right),$$

by theorem 2.2. This structure supports `access` queries in constant time. Constant time support for `degree` queries was shown in the beginning of section 5.4. It only remains to support `successor` queries in constant time.

To support `c-successor` queries in constant time we need to add auxiliary structures. We define *r-successor-offset* at a zero entry (i, j) as zero if entry $(i, j-1)$ is a zero entry or if $j = 1$ and otherwise we define *r-successor-offset* as the offset t of the one successor to (i, j) (in other words, the c-successor-offset is t if first one entry in row i after (i, j) is $(i, j+t)$). We form a bit vector `offsets` by traversing the matrix in a row-major fashion and for each zero entry, we append to `offsets` the unary encoding of its r-successor-offset.

We argue that the length of bit vector `r-offsets` is at most $2m'$. This is since each zero entry contributes at most two bits to `r-offsets`; if a zero entry belongs to a block of consecutive zeros in a row, it contributes a single bit to the unary encoding of r-successor-offset for the entry at the beginning of the block and another bit when encoding zero as the r-successor-offset if it is not at the beginning of the block. We represent `r-offsets` using the FID structure (theorem 2.1) which requires $O(m')$ bits and $O(m')$ bits is of lower order

terms compared to $\lg\binom{n^2}{m'} = \lg\binom{n^2}{m}$ where $m > n^2\left(1 - O\left(\frac{1}{\sqrt{\log n}}\right)\right)$.

Using the `r-offsets` structure, it is easy to support `r-successor` queries in constant time. Given an entry (i, j) to determine the `r-successor`, entry $(i, j+1)$ is checked first; if it is one, we are done. Otherwise, it is a zero and we use `rank` capability of the ID structure to determine its rank among zeros and we use this index to find the corresponding entry in structure `r-offsets`. Since entry $(i, j+1)$ is preceded by a one entry (position (i, j)), `r-offsets` stores the offset of the successor one. Hence, we can perform `r-successor` queries in constant time.

Analogously, we define *c-successor-offset* and construct `c-offsets` structures which adds only $O(m)$ bits to the storage requirement and enables us to support `c-successor` in constant time as well.

Theorem 5.7. *A 0-1 matrix of size $n \times n$ with $n^2 - \Omega(n) \geq m > n^2\left(1 - O\left(\frac{n^2}{\sqrt{\log n}}\right)\right)$ ones can be represented in $\lg\binom{n^2}{m} + o\left(\lg\binom{n^2}{m}\right)$ bits supporting* `degree`*,* `access` *and* `successor` *queries in constant time.* □

5.4.3 The dense case

There are $n^2\left(1 - \omega\left(\frac{1}{\sqrt{\log n}}\right)\right) \geq m = \omega\left(\frac{n^2}{\sqrt{\log n}}\right)$ ones in the matrix, we divide the matrix into tiny square matrices of size $\frac{1}{2}\sqrt{\lg n} \times \frac{1}{2}\sqrt{\lg n}$ and represent each tiny square by a reference to a look-up table.

The look-up table exhaustively lists all square matrices of size $\frac{1}{2}\sqrt{\lg n} \times \frac{1}{2}\sqrt{\lg n}$ ordered by the number of ones. It moreover contains answers to `access` and `successor` queries for all different possible parameters of these queries. The space of the table is clearly $o(n)$ and negligible.

A tiny matrix is encoded by a reference to a look-up table. The reference is a pair (t, i) which t is the number of ones in the matrix and i distinguishes the matrix among all matrices with the same number of ones. To account for the space of references, we calculate the space for t and i fields separately. Fields t amount to $O(n^2 \log\log n / \log n) = o(m)$ which is negligible. The i fields have a space requirement which is dominant in the overall space bound. In a tiny matrix T_i with k_i ones, the i field requires $\left\lceil \lg\binom{\lg n/4}{k_i} \right\rceil$ bits. Therefore, over all the tiny matrices, these fields occupy $\sum_{i=1}^{4n^2/\lg n} \left\lceil \lg\binom{\lg n/4}{k_i} \right\rceil < \lg\binom{n^2}{m} + o\left(\log\binom{n^2}{m}\right)$ bits.

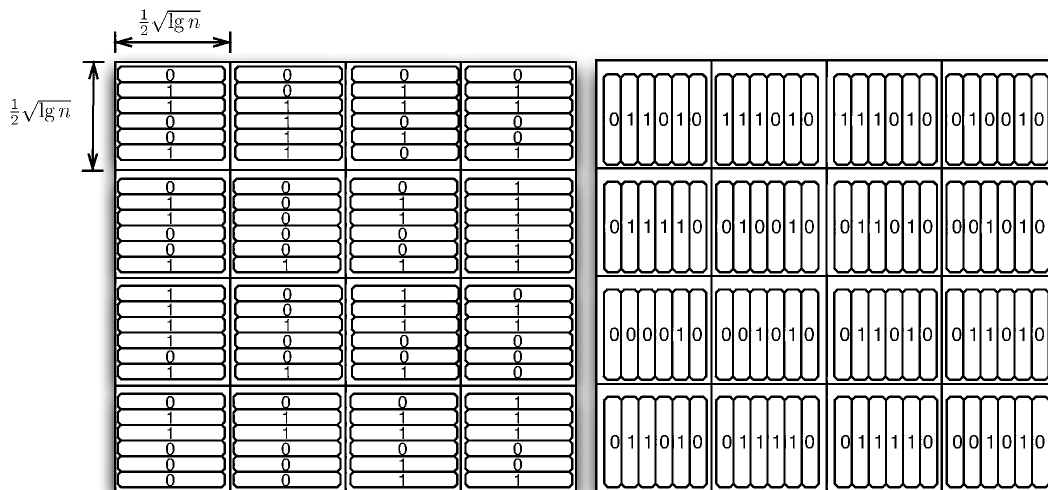


Figure 5.1: Summary bits are stored for each tiny row and column.

Thus far, we can answer our desired queries in tiny matrices. To extend this power to the entire matrix, we simply use summary bits for rows and columns of tiny matrices as illustrated in figure 5.1. We form a bit vector of length $2n/\sqrt{\lg n}$ for each row of the matrix. Bit i of the vector is a summary bit for the corresponding row of the tiny matrix; it is set to one if there is at least a one in that row. Similarly, we form a bit vector for each column of the matrix which contains summary bits for the columns of tiny matrices. We represent these bit vectors using the FID structure of theorem 2.1. These structures together require $O\left(n^2/\sqrt{\log n}\right)$ bits of space which is a lower order term with respect to $\lg \binom{n^2}{m}$ where $n^2\left(1-\omega\left(\frac{1}{\sqrt{\log n}}\right)\right) \geq m = \omega\left(\frac{n^2}{\sqrt{\log n}}\right)$.

Implementation of `access(i,j)` queries is trivial; the tiny matrix containing cell i, j is found and the answer to the query is in the look-up table. `successor` queries are implemented using the summary bit vectors. Using select query supported by the FIDs, we find in constant time the tiny row containing the next one on a column or a row and once there, we use the look-up table of the containing tiny matrix to report ones. Constant time support for `degree` queries was shown in the beginning of section 5.4.

Theorem 5.8. *A 0-1 matrix of size $n \times n$ with $n^2\left(1-\omega\left(\frac{1}{\sqrt{\log n}}\right)\right) \geq m = \omega\left(\frac{n^2}{\sqrt{\log n}}\right)$ ones can be represented in $\lg \binom{n^2}{m} + o\left(\lg \binom{n^2}{m}\right)$ bits supporting* `access`*,* `successor`*, and* `degree` *queries in constant time.* □

5.4.4 The moderate case

This is the case where $O\left(\frac{n^2}{\sqrt{\log n}}\right) = m > n/2$. We divide the matrix into smaller square matrices of size $n^2/(2m) \times n^2/(2m)$. As in the dense case (section 5.4.3), we store summary bits to be able to reduce the problem to within individual $n^2/(2m) \times n^2/(2m)$ matrices. For each row of the matrix, we form a bit vector of length $n/(n^2/2m) = 2m/n$. Bit i of the vector is one if there is at least a one in the corresponding row of small matrix i in the row. Similarly, there is a bit vector of the same length for each column of the matrix.

As in section 5.4.3, these bit vectors are represented using the FID structure of theorem 2.1. To account for the space of these structures, it suffices to observe that each bit vector has space $O(2m/n)$ and there is $O(n)$ of these vectors and total space is $O(m)$ bits and since $m = O\left(\frac{n^2}{\sqrt{\log n}}\right)$, this is a lower order space term.

Using these FIDs, one can navigate away from any cell to the small matrix containing the next one in the row or the column in constant time. Therefore `successor` queries reduce to `successor` queries within small matrices. Trivially, `access` queries reduce to those within small matrices as well. Hence, the entire problem reduces to representing small matrices. We state the bounds of our representation for small matrices and prove it subsequently:

Lemma 5.9. *A 0-1 matrix of size $u \times u$ containing r ones can be encoded in $(1+\epsilon) r \lg u + O(u)$ bits for any constant $\epsilon > 0$ supporting* `access` *and* `successor` *queries in constant time $O(1/\epsilon)$.*

Given the lemma which we prove in the following section, we use its encoding to represent the small matrices of size $n^2/(2m) \times n^2/(2m)$. We showed that `access` and `successor` queries reduce to within individual small matrices and the representation of the lemma performs these queries in constant time. Furthermore, constant time support for `degree` queries was shown in the beginning of section 5.4.

To account for the space, each small matrix with r_i ones requires $(1+\epsilon) r_i \lg n^2/(2m) + O(n^2/m)$. Hence, the total space over all small matrices can be computed as follows (given that $m = O\left(\frac{n^2}{\sqrt{\log n}}\right)$):

$$\sum_{i=1}^{4m^2/n^2} (1+\epsilon) r_i \lg \frac{n^2}{2m} + O\left(\frac{n^2}{m}\right) = (1+\epsilon) m \lg \frac{n^2}{2m} + O(m) = (1+\epsilon) \lg \binom{n^2}{m} + o\left(\lg \binom{n^2}{m}\right).$$

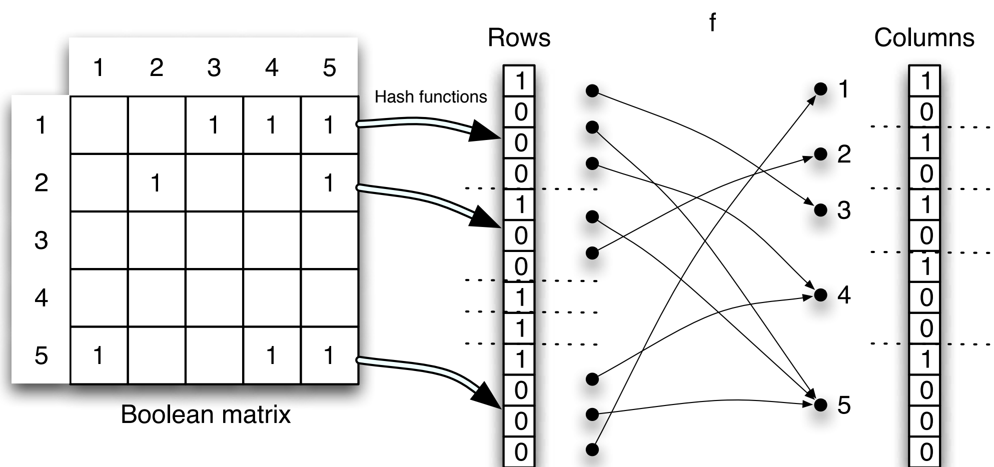


Figure 5.2: Structures used to represent a small matrix

Theorem 5.10. *A 0-1 matrix of size $n \times n$ with m ones such that $O\left(\frac{n^2}{\sqrt{\log n}}\right) = m > n/2$ can be represented in $(1+\epsilon)\lg\binom{n^2}{m}$ bits for any constant $\epsilon > 0$ supporting* `access`*,* `successor`*, and* `degree` *queries in constant time $O(1/\epsilon)$.* □

Representing small matrices (lemma 5.9)

We showed in the previous section that we can focus our attention on small matrices. We give the small matrix representation to encode a 0-1 matrix of size $u \times u$ containing r ones and prove lemma 5.9 in this section.

We use a minimal perfect hash function on each row that hashes u cells on the row to a set of r_i cells where r_i is the number of ones in the cell. The matrix is traversed in a row-major fashion and the ones in each row are traversed in the order that the perfect hash function dictates so after the traversal each one is (implicitly) assigned a number from $[r]$.

We store a function $f : [r] \to [u]$ where ones in the matrix map their assigned numbers from $[r]$ to their corresponding columns. In other words, an entry in row i and column j which is assigned t in the traversal, maps t to j ($f(t) = j$). This function is represented using theorem 5.1 which supports `access`, `select`, and `partial-rank` operations in constant time $O(1/\epsilon)$. Structures of this section are illustrated in figure 5.2.

Furthermore, a bit vector `Rows` of length $u + r$ is stored which contains descriptions of

r_i's in order in unary format; for each row i in order, there is a one bit followed by r_i zero bits. Similarly, we stored a bit vector `Columns` of length $u + r$ which contains c_i's in order in unary format where c_i is the number of ones in column i. These bit vectors are encoded using the FID structure of theorem 2.1.

We now explain how queries can be implemented. To respond to an `access(i, j)` query, we perform a `select(i)` on bit vector `Rows` to find the first assigned number on row i. We use the perfect hash function h_i of row i to get the offset from which the entry (i, j) is hashed to; the assigned number to entry (i, j) is $t =$`select(i)`$+h_i(j)$. Now the function is used and if $f(t) = j$ then the content (i, j) of the matrix is one and it is zero otherwise.

To implement `successor` queries, we need to find ones in a row or a column in constant time per element. We first explain how we can translate a location in `Rows` to the corresponding location in `Columns` and vice versa. Given a location in `Rows`, we determine the corresponding row by a rank query and determine the corresponding column by using the function. We use the `partial_rank` capability and determine the rank of the cell in the column. The corresponding entry in `Columns` bit vector is determined by a select query in `Columns` to go to the start of the appropriate column and we add to that the rank in the column to find the correct entry. Backward translation is performed similarly using the `select` capability of the function.

Determining a successor one in a row (*i.e.,* `r-successor`) is easy as consecutive locations in `Rows` contain all ones in a row and similarly determining a successor one in a column is easy as consecutive locations in `Columns` contain all ones in a column. Given the translation capability, we can find the row and column number of an entry in `Rows` or `Columns`.

Space requirement of our structures can be accounted as follows. The perfect hash functions together require $O(u + r \log \log u)$ bits (see for example [36]). Representation of bit vectors uses $\lg \binom{r+u}{u} + o(r + u)$ by theorem 2.1. The representation of the function uses $(1 + \epsilon) r \log u$ bits. Thus all together the space requirement of our representation is $(1 + \epsilon) r \log u + O(u)$ which concludes the proof of lemma 5.9. □

5.4.5 The sparse case

In the extreme case where there is $m \leq n/2$ ones in the matrix, the representation is analogous the other extreme case where $m = n^2 - o(n)$ (*i.e.,* almost-full case).

There exist columns or rows containing no ones. We handle this case simply by projecting the matrix into non-empty rows and columns and representing the projected matrix using the moderate case.

To project the matrix, we form two bit vectors `R`,`C` of length n which encode non-empty rows and columns respectively. Bit i of `R` is one if row i contains an entry which is one and the bit is zero otherwise. Similarly, bit i of `C` is one if column i is non-empty and one if it is empty. We represent these bit vectors using the ID structure of theorem 2.2 using $2m \lg(n/m) + o(m)$ bits. We denote by M' the sub-matrix formed on non-empty rows and columns. We add extra rows and columns to M' to make it an $m \times m$ matrix and fill in these rows and columns by zeros. Matrix M' has size $m \times m$ and contains m ones, therefore its density classifies it as a moderate-case matrix. We use the representation of section 5.4.4 to encode M'. This representation uses at most $(1 + \epsilon')m \lg m$ bits for any constant $\epsilon' > 0$. Hence, the storage requirement of all structures calculates as follows:

$$2m \lg(n/m) + o(m) + (1 + \epsilon')m \lg m = \lg \binom{n^2}{m} + \epsilon m \lg m,$$

where $m \leq n/2$ for any arbitrary small $\epsilon > 0$.

Given the representation of the sub-matrix and encodings of `R` and `C` to retrieve empty rows, we demonstrate how to implement queries in constant time $O(1/\epsilon)$. To answer an `access` query on a position (i, j), we determine using `R,C` whether row i or column j are empty. If either the row or the column is empty, then (i, j) contains a zero. Otherwise, we translate position (i, j) in matrix M to position (i', j') in matrix M' using the rank capability of ID structures of `R,C`, and to answer the query, it suffices to return the answer to `access` query on location (i', j') in matrix M' which is supported in $O(1/\epsilon)$ time.

To answer a `successor` query on position (i, j), we translate the position to position (i', j') in matrix M' and find the successor (i'_s, j'_s) in matrix M'. We can translate position (i_s, j_s) to position (i^*, j^*) in matrix M using select capability of the ID structures for `R,C`. We return position (i_s, j_s) as the successor.

To answer `degree` queries, it suffices to check whether the row/column is empty by a membership test in `R` or `C`. If the row/column is empty then the degree is zero, otherwise, we translate the row/column number to row/column number in matrix M' and use `degree` query in matrix M' which is supported in constant time.

Theorem 5.11. *A 0-1 matrix of size $n \times n$ with m ones such that $m \leq n/2$ can be represented in $\lg \binom{n^2}{m} + \epsilon m \lg m$ bits for any constant $\epsilon > 0$ supporting* `access`*,* `successor`*, and* `degree` *queries in constant time $O(1/\epsilon)$.* □

Since $m \lg m = o\left(\lg \binom{n^2}{m}\right)$ if $m < n^\delta$ for any constant $\delta > 0$, and $m \lg m = \Theta\left(\lg \binom{n^2}{m}\right)$ if otherwise (in the sparse case), theorem 5.11 implies the following corollaries:

Corollary 5.12. *For $m < n^\delta$ for any constant $\delta > 0$, a 0-1 matrix of size $n \times n$ with m ones can be represented in $\lg \binom{n^2}{m} + o\left(\lg \binom{n^2}{m}\right)$ bits supporting* `access`*,* `successor`*, and* `degree` *queries in constant time.* □

Corollary 5.13. *For $m \leq n/2$ and $m > n^\delta$ for some constant $\delta > 0$, a 0-1 matrix of size $n \times n$ with m ones can be represented in $(1+\epsilon) \lg \binom{n^2}{m}$ bits for any constant $\epsilon > 0$ supporting* `access`*,* `successor`*, and* `degree` *queries in constant time $O(1/\epsilon)$.* □

5.5 Undirected graphs

In this section we argue that the results on succinct representations of graphs we have provided thus far for directed graphs extend to undirected graphs. As we explained in section 5.2, representing a directed graph with n vertices and m edges can be viewed as the representing the equivalent $n \times n$ boolean matrix containing m ones. Analogously, representing an undirected graph with n vertices and m edges is equivalent to encoding the $n \times n$ upper-triangular adjacency matrix which contains m ones.

In section 5.5.1, we show that lower bound results of section 5.3 extend to undirected graphs and subsequently section 5.5.2 extends the upper bound results of section 5.4 to undirected graphs.

5.5.1 Space lower bounds for undirected graphs

Analogous to theorem 5.2, there is a trivial information theory bound for undirected graphs which comes directly from a counting argument and it holds for any representation regardless of the queries it can support:

Theorem 5.14. *Any representation of $n \times n$ upper triangular boolean matrices contain m ones requires $\lg \binom{\frac{n^2+n}{2}}{m}$ bits in the worse case for some matrices.* □

In section 5.5.1, we distinguished two cases depending on relative values of n, m and studied them separately: (1) the moderate case: $m = \Omega(n)$ and $m = O\left(n^{2-\delta}\right)$ for any constant $\delta > 0$, and (2) the Sparse case: $m = o(n)$. We provide the following general reduction theorem:

Lemma 5.15. *If any $n \times n$ upper-triangular boolean matrix containing m ones can be encoded in $f(n,m)$ bits supporting* `access`*,* `successor`*, and* `degree` *queries in constant time, then any $\frac{n}{2} \times \frac{n}{2}$ boolean matrix (not necessarily upper-triangular) containing m ones can be encoded in $f(n,m)$ bits supporting the same set of queries in constant time.*

Proof. Given a $\frac{n}{2} \times \frac{n}{2}$ boolean matrix M(not necessarily upper-triangular) containing m ones, we build a $n \times n$ upper-triangular matrix M' as follows. M' is divided into four quadrants by splitting in two vertically and horizontally. All quadrants consist entirely of zeros except the top-right one which holds a copy of M'. Matrix M is upper-triangular and therefore can be encoded in $f(n,m)$ bits supporting the queries in constant time. It is trivial to observe that support for these queries in M' follows from that in M and therefore M, is represented in $f(n,m)$ bits. □

Given the reduction, it is easy to extend the lower bound results of section 5.3 to upper-triangular matrices (*i.e.,* undirected graphs):

Theorem 5.16. *If $n^\delta < m < n^{2-\delta}$ for some constant $\delta > 0$, there exists an upper-triangular matrix $n \times n$ containing m ones that cannot be represented in $\lg\binom{\frac{n^2+n}{2}}{m} + o\left(\lg\binom{\frac{n^2+n}{2}}{m}\right)$ with constant time support for* `access` *and* `successor` *queries.*

Proof. We prove by contradiction: assume there is a representation for all such matrices with such space. We note that $\lg\binom{\frac{n^2+n}{2}}{m}$ equals $\lg\binom{n^2}{m}$ to within lower order terms where $n^\delta < m < n^{2-\delta}$ for some constant $\delta > 0$. One can use the reduction lemma 5.15 to derive a contradiction with corollary 5.5. □

5.5.2 Space upper bounds for undirected graphs

In section 5.4, we categorized matrices according to their densities into five categories. In all categories other than the dense case the value of m relative to n is such that $\lg\binom{\frac{n^2+n}{2}}{m}$ equals $\lg\binom{n^2}{m}$ to within lower order terms. Therefore in those four categories, one can treat the

upper-triangular matrix as a general matrix and achieve a representation with the desired storage requirement. In this section, we detail the required modifications in representations of dense-case matrices to achieve the optimal information-theory space bound.

Theorem 5.17. *if* $n^2\left(1-\omega\left(\frac{1}{\sqrt{\log n}}\right)\right) \geq m = \omega\left(\frac{n^2}{\sqrt{\log n}}\right)$, *an* $n \times n$ *upper-triangular matrix containing* m *ones can be encoded in* $\lg\binom{\frac{n^2+n}{2}}{m} + o\left(\lg\binom{\frac{n^2+n}{2}}{m}\right)$ *bits such that* `access`*,* `successor`*, and* `degree` *queries can be supported in constant time.*

The representation is analogous to that of section 5.4.3 as the matrix is divided into $\frac{1}{2}\sqrt{\lg n} \times \frac{1}{2}\sqrt{\lg n}$ tiny matrices. The summary bits are stored identically. The only deviation from that representation is that since we know that tiny matrices that are strictly below the diagonal of the matrix are entirely zeros, we do not explicitly store them. Furthermore, the tiny matrices that intersect the diagonal are represented such that only the half portion that is above the diagonal is encoded as the other half is by definition empty. For this purpose, an auxiliary look-up table is set to catalogue half filled matrices. We use the leading bit in an encoding of a tiny matrix to specify whether it is a entirely or half filled, and use a reference as in section 5.4.3 to within the appropriate look-up table.

We now argue that the storage requirement is $\lg\binom{\frac{n^2+n}{2}}{m}$. Any structures stored other than the look-up references of tiny matrices contribute to lower order terms. To account for the space of tiny matrix encodings, we note that the space can be computed analogously to section 5.4.3 and extra contribution of half tiny matrices is $O(n/\log n)$. Hence, the storage requirement can be calculated to at most:

$$\lg\binom{\frac{n^2+n}{2}}{m} + o\left(\lg\binom{\frac{n^2+n}{2}}{m}\right) + O\left(\frac{n}{\log n}\right) = \lg\binom{\frac{n^2}{2}}{m} + o\left(\lg\binom{\frac{n^2}{2}}{m}\right),$$

where $n^2\left(1-\omega\left(\frac{n^2}{\sqrt{\log n}}\right)\right) \geq m = \omega\left(\frac{n^2}{\sqrt{\log n}}\right)$ (*i.e.*, the dense case).

Table 5.2 summarizes the space lower bound and upper bounds we have proved for undirected graphs.

m	Space Lower bound	Space Upper bound
$\forall\delta > 0;\ \mathbf{m} < n^{\delta}$	$\lg \binom{\frac{n^2+n}{2}}{m}$	$\lg \binom{\frac{n^2+n}{2}}{m}$
$\exists\delta > 0;\ n^{\delta} < \mathbf{m} = o(n)$	$(1+\epsilon)\lg \binom{\frac{n^2+n}{2}}{m}$	$(1+\epsilon)\lg \binom{\frac{n^2+n}{2}}{m}$
$\exists\delta > 0;\ \Omega(n) = \mathbf{m} < n^{2-\delta}$	$(1+\epsilon)\lg \binom{\frac{n^2+n}{2}}{m}$	$(1+\epsilon)\lg \binom{\frac{n^2+n}{2}}{m}$
$\forall\delta > 0;\ n^{2-\delta} < \mathbf{m} = O\left(\frac{n^2}{\sqrt{\log n}}\right)$	$\lg \binom{\frac{n^2+n}{2}}{m}$	$(1+\epsilon)\lg \binom{\frac{n^2+n}{2}}{m}$
$\omega\left(\frac{n^2}{\sqrt{\log n}}\right) = \mathbf{m}$	$\lg \binom{\frac{n^2+n}{2}}{m}$	$\lg \binom{\frac{n^2+n}{2}}{m}$

Table 5.2: Space lower and upper bounds for representing a directed graph with n vertices and m edges which supports the queries in constant time. All upper bounds are up to lower order terms.

5.6 Conclusion and final remarks

We considered the problem of encoding a labeled graph (directed or undirected) with n vertices and m edges succinctly supporting adjacency, neighborhood and degree queries. Measuring the storage requirement of representations as a function of n and m, we showed that the information-theoretic space lower bound of $\lg \binom{n^2}{m}$ for directed graphs and $\lg \binom{\frac{n^2+n}{2}}{m}$ for undirected graphs is not achievable for most values of m relative to n by proving a better lower bound. More precisely, We proved impossible to achieve the information-theory lower bound within lower order terms unless the number of edges in the graph is such that $m = o\left(n^{\delta}\right)$ or $m = \omega\left(n^{2-\delta}\right)$ for any constant $\delta > 0$.

We furthermore matched the lower bound in its applicable range by presenting an encoding with the worst case storage requirement of $(1+\epsilon)\lg \binom{n^2}{m}$ for any $\epsilon > 0$ for directed graphs and $(1+\epsilon)\lg \binom{\frac{n^2+n}{2}}{m}$ for any $\epsilon > 0$ for undirected graphs. Where $m = o\left(n^{\delta}\right)$ for any constant $\delta > 0$, the representation matches the information theoretic lower bound tightly to within lower order terms. The information-theory lower bound is also matched where $m = \omega\left(\frac{n^2}{\sqrt{\log n}}\right)$. This leaves a small gap where $n^{2-\delta} < m = O\left(\frac{n^2}{\sqrt{\log n}}\right)$ for any constant $\delta > 0$ where our lower and upper bounds are a multiplicative factor $1+\epsilon$ apart for any arbitrary small constant $\epsilon > 0$, which we leave as an open problem.

Chapter 6

Conclusion

Succinct representations of combinatorial objects are compact encodings that support a reasonable set of queries on the objects in constant time in the word RAM model. Furthermore, the storage requirement of the encoding is to match the information theory lower bound to within lower order terms. We studied trees and graphs as the objects for succinct representation in this thesis.

We started the thesis by proposing a novel uniform approach towards succinct representation of trees in chapter 3. The key difference in the approach in comparison with other partition-based succinct encodings of trees is in the tree decomposition part. Our tree decomposition greatly isolates subtree components from each other in order to simplify the representation of interconnection between different components. We showed that all families of trees with a previously-existing succinct encoding can be represented using our framework. Moreover, using the approach we could improve the best existing representations for some families of trees (cardinal and free trees). In the case of ordinal trees, the new representation greatly simplifies implementation of various operations. For cardinal trees, the representation improves on the existing ones in that it allows support of both cardinal and ordinal type of operations. The approach easily produces a succinct representation for free trees, a class of trees that had not been studied previously for succinct representations. The succinct representation can also be readily made adaptive to a variety of desired entropy measures such as degree-distribution entropy.

Subsequently in chapter 4, we focused on ordinal trees and presented a universal representation that combined the query-support power of various succinct encodings. The existing

approaches to succinctly represent an ordinal tree are (1) level order unary degree sequence (LOUDS), (2) balanced parentheses (BP), (3) depth first unary degree sequence (DFUDS), and (4) tree covering (TC). We argued that the power of LOUDS approach is limited and it is intrinsically different from the other approaches. We gave a new universal succinct representation of ordinal trees that is able to simultaneously emulate BP, DFUDS, and TC representations. The emulation is by the way of producing any logarithm-long word of the BP or the DFUDS sequences (operations `BP-word`,`DFUDS-word`), and producing any micro-tree of the TC representation (operation `TC-microtree`) all in constant time. We argued that support for these operations means that the universal representation can emulate either of these representations at the same time on demand. Moreover we showed that, an important implication of the emulation capability is that any query supported in constant time by any of these representations can be supported in constant time by the universal representation. The main problem we leave open is to whether the universal representation can be improved to also emulate the LOUDS representation.

We moved from trees—which are special type of graphs— to arbitrary graphs in chapter 5. We considered the problem of encoding a labeled graph (directed or undirected) with n vertices and m edges succinctly supporting adjacency, neighborhood, and degree queries in constant time. The adjacency query is to test whether an edge exists between two given vertices and neighborhood query is to scan through the edges incident to a given vertex in constant time per such edge.

Measuring the storage requirement of representations as a function of n and m, we showed that the information-theoretic lower bound for either directed ($\left\lceil \lg \binom{n^2}{m} \right\rceil$) or undirected graphs ($\left\lceil \lg \binom{n^2+n}{m} \right\rceil$) is not achievable to within lower order terms for most values of m relative to n (namely where these exists a $\delta > 0$ such that $n^\delta < m < n^{2-\delta}$) if we insist on constant time query support. We furthermore matched the lower bound in its applicable range by presenting an encoding with the worst case storage requirement of $(1+\epsilon)$ multiplicative factor away from the information theory bound where $\epsilon > 0$ is any arbitrarily small constant. Where the graph is too sparse ($\forall \delta > 0;\ m < n^\delta$) or the graph is too dense ($n^2/o\left(\sqrt{\log n}\right) < m$), we give a representation with storage requirement which tightly matches the information theory lower bound to within lower order terms. The space lower and upper bounds according to different values of m relative to n are listed in table 5.1 for directed graphs and in table 5.2 for undirected graphs. There is a small range of values of m where the given lower and upper bounds do not match which we leave as an open problem.

Bibliography

[1] *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*. Springer, 2007.

[2] Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas, editors. *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I*, volume 5555 of *Lecture Notes in Computer Science*. Springer, 2009.

[3] Luca Castelli Aleardi, Olivier Devillers, and Gilles Schaeffer. Optimal succinct representations of planar maps. In *Proceedings of the Symposium on Computational Geometry*, pages 309–318. ACM, 2006.

[4] Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors. *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*. Springer, 2003.

[5] Jérémy Barbay, Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In Moshe Lewenstein and Gabriel Valiente, editors, *The Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 4009 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 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 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.

[7] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computers and Systems Sciences*, 65(1):38–72, 2002.

[8] 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.

[9] F.R. Bernhart. Catalan, motzkin, and riordan numbers. *Discrete Mathematics*, 204:72–112, 1999.

[10] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 679–688, 2003.

[11] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[12] Yi-Ting Chiang, Ching-Chi Lin, and Hsueh-I Lu. Orderly spanning trees with applications to graph encoding and graph drawing. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 506–515, Philadelphia, PA, USA, 2001.

[13] 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 *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 118–129, 1998.

[14] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.

[15] David Richard Clark. *Compact pat trees.* PhD thesis, University of Waterloo, Ontario, Canada, 1998.

[16] I. M. H. Etherington. Non-associate powers and a functional equation. *The Mathematical Gazette*, 21(242):36–39, 1937.

[17] Arash Farzan and J. Ian Munro. Succinct representations of arbitrary graphs. In Dan Halperin and Kurt Mehlhorn, editors, *ESA*, volume 5193 of *Lecture Notes in Computer Science*, pages 393–404. Springer, 2008.

[18] Arash Farzan and J. Ian Munro. A uniform approach towards succinct representation of trees. In *SWAT (11th Scandinavian Workshop on Algorithm Theory)*, Lecture Notes in Computer Science, pages 173–184. Springer, 2008.

[19] Arash Farzan and J. Ian Munro. Dynamic succinct ordered trees. In Albers et al. [2], pages 439–450.

[20] Arash Farzan, Rajeev Raman, and S. Srinivasa Rao. Universal succinct representations of trees? In Albers et al. [2], pages 451–462.

[21] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

[22] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly FM-index. In Alberto Apostolico and Massimo Melucci, editors, *String Processing and Information Retrieval Symposium (SPIRE)*, volume 3246 of *Lecture Notes in Computer Science*, pages 150–160. Springer, 2004.

[23] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[24] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *Journal of ACM*, 31(3):538–544, 1984.

[25] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.

[26] Alexander Golynski. Optimal lower bounds for rank and select indexes. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *International Colloquium on Automata, Languages, and Programming (ICALP) (1)*, volume 4051 of *Lecture Notes in Computer Science*, pages 370–381. Springer, 2006.

[27] Alexander Golynski. *Upper and lower bounds for Text Indexing Data Structures*. PhD thesis, Waterloo, Ontario, Canada., 2007.

[28] Alexander Golynski. Cell probe lower bounds for succinct data structures. In *SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms*, pages 625–634, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[29] Alexander Golynski, Roberto Grossi, Ankur Gupta, Rajeev Raman, and S. Srinivasa Rao. On the size of succinct indices. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *European Symposium on Algorithms (ESA)*, volume 4698 of *Lecture Notes in Computer Science*, pages 371–382. Springer, 2007.

[30] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 368–373, New York, NY, USA, 2006. ACM.

[31] Alexander Golynski, Rajeev Raman, and S. Srinivasa Rao. On the redundancy of succinct data structures. In Joachim Gudmundsson, editor, *Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 5124 of *Lecture Notes in Computer Science*, pages 148–159. Springer, 2008.

[32] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[33] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[34] 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.

[35] Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. A framework for dynamizing succinct data structures. In *International Colloquium on Automata, Languages, and Programming (ICALP)* [1], pages 521–532.

[36] Torben Hagerup and Torsten Tholey. Efficient minimal perfect hashing in nearly minimal space. In Afonso Ferreira and Horst Reichel, editors, *Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 2010 of *Lecture Notes in Computer Science*, pages 317–326. Springer, 2001.

[37] Frank Harary and Edgar M. Palmer. *Graphical Enuemration*. Academic Press, New York, 1973.

[38] Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct ordinal trees based on tree covering. In *International Colloquium on Automata, Languages, and Programming (ICALP)* [1], pages 509–520.

[39] Guy Joseph Jacobson. *Succinct static data structures*. PhD thesis, Pittsburgh, PA, USA, 1988.

[40] Guy Joseph Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[41] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 575–584. SIAM, 2007.

[42] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. *SIAM Journal on Discrete Mathematics*, 5(4):596–603, 1992.

[43] Keeler and Westbrook. Short encodings of planar graphs and maps. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 58, 1995.

[44] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, third edition, 1997.

[45] Hsueh-I Lu and Chia-Chi Yeh. Balanced parentheses strike back. *ACM Trans. Algorithms*, 4(3):1–13, 2008.

[46] Peter Bro Miltersen. Cell probe complexity - a survey. In *In 19th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 1999. Advances in Data Structures Workshop*, 1999.

[47] J. Ian Munro. Succinct data structures. *Electronic Notes on Theoretical Computer Science*, 91:3, 2004.

[48] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations. In Baeten et al. [4], pages 345–356.

[49] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 118–126, 1997.

[50] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal Computing*, 31(3):762–776, 2001.

[51] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[52] J. Ian Munro, Venkatesh Raman, and Adam J. Storm. Representing dynamic binary trees succinctly. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 529–536, 2001.

[53] J. Ian Munro and S. Srinivasa Rao. Succinct representations of functions. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 3142 of *Lecture Notes in Computer Science*, pages 1006–1015. Springer, 2004.

[54] A. M. Odlyzko. Some new methods and results in tree enumeration, May 04 1984.

[55] Richard Otter. The number of trees. *The Annals of Mathematics, 2nd Ser.*, 49(3):583–599, 1948.

[56] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[57] Mihai Pătraşcu. Succincter. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 305–313. IEEE Computer Society, 2008.

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

[59] Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In Baeten et al. [4], pages 357–368.

[60] S. Srinivasa Rao. Time-space trade-offs for compressed suffix arrays. *Inf. Process. Lett.*, 82(6):307–311, 2002.

[61] G. Rote. Binary trees having a given number of nodes with 0,1, and 2 children. *Sminaire Lotharingien de Combinatoire*, 38, 1997.

[62] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 225–232, 2002.

[63] G. Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8:289–294, 1984.

[64] J. H. M. Wedderburn. The functional equation $g(x^2) = 2ax + [g(x)]^2$. *The Annals of Mathematics, 2nd Ser.*, 24(2):121–140, 1922.

Index

access query, 61, 62, 69, 78
adjacency matrix, 58
adjacency query, 59
adjaceny list, 58

balanced parenthesis, 2, 7, 37–38
binary relation, 9, 60
  label-access, 9
  label-rank, 9
  label-select, 9
  object-access, 9
  object-rank, 9
  object-select, 9
bit probe model, 37
bit vector, 4
BP, *see* balanced parentheses
BP-word, 40

Catalan numbers, 12
  generalized, 12
cell probe model, 8, 65, 69

degree query, 59
depth first unary degree sequence, 2, 7, 38–39
DFUDS-word, 40
dictionary, 4

entropy
  degree distribution, 13, 31
  k-th order, 10
  tree encoding, 31–34
  zeroth order, 8
Etherington-Wedderburn sequence, 30

FID, *see* fully indexable dictionary
FKS dictionary, 5
full rank, 62
fully indexable dictionary, 5–6
function, 8, 62, 78

graph, 10
  decomposable, 10, 60
  limited arboricity, 3, 10, 60
  planar, 3, 10, 60
  separable, 3, 10, 60
  triangulated, 60
  triangulated planar, 10
  triconnected, 10, 60
  undirected, 81–83

ID, *see* indexable dictionary
in-neighbor, 59
indexable dictionary, 5–6

left-leaning paths, 51
level order unary degree sequence, 7, 36–37

LOUDS, *see* level order unary degree sequence

micro-trees, 20
mini-trees, 20

opportunistic data structures, 10
out-neighbor, 59

partial rank, 61, 62, 78, 79
permutation, 7, 62, 64
  cycle representation, 7
predecessor query, 5–6, 66

RAM, 2, 27
rank, 4–6
reciprocal (of a query), 69
right-leaning paths, 51
Riordan numbers, 33

select, 4–6, 62, 78
skeleton, 44
skinny chunks, 51
skinny trees, 44–50
string, 8, 61
successor query, 61, 66, 69
suffix array, 10
suffix tree, 10

TC, *see* tree covering
TC-microtree, 40
text index, 10
tree
  AVL, 34
  binary, 6, 7
  cardinal, 7, 11, 27–29
  DNA, 11
  free, 11, 29–31
  free binary, 12
  free trees, 2
  k-ary, *see* cardinal
  ordinal, 6, 11, 20–27
  wavelet, 8
tree covering, 7, 39
tree decomposition, 14–19