Space Efficient Graph Algorithms

By

Sankar Deep Chakraborty

MATH10201105004

The Institute of Mathematical Sciences, Chennai

A thesis submitted to the

Board of Studies in Mathematical Sciences

In partial fulfillment of requirements

for the Degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

March, 2018

Homi Bhabha National Institute

Recommendations of the Viva Voce Committee

As members of the Viva Voce Committee, we certify that we have read the dissertation prepared by Sankar Deep Chakraborty entitled "Space Efficient Graph Algorithms" and recommend that it may be accepted as fulfilling the thesis requirement for the award of Degree of Doctor of Philosophy.

____ Date: March 16, 2018

Chairman - Meena Mahajan

____ Date: March 16, 2018

Guide/Convenor - Venkatesh Raman

____ Date: March 16, 2018

Examiner - Jaikumar Radhakrishnan

____ Date: March 16, 2018

Member 1 - Saket Saurabh

____ Date: March 16, 2018

Member 2 - Samir Datta

Final approval and acceptance of this thesis is contingent upon the candidate's submission of the final copies of the thesis to HBNI.

I hereby certify that I have read this thesis prepared under my direction and recommend that it may be accepted as fulfilling the thesis requirement.

Date: March 16, 2018

Place: Chennai Guide

STATEMENT BY AUTHOR

Sankar Deep Chakraborty

DECLARATION

I hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and has not been submitted earlier as a whole or in part for a degree / diploma at this or any other Institution / University.

Sankar Deep Chakraborty

LIST OF PUBLICATIONS ARISING FROM THE THESIS

Journal

1. Biconnectivity, *st*-numbering and Other Applications of DFS Using $O(n)$ bits, Sankardeep Chakraborty, Venkatesh Raman and Srinivasa Rao Satti, *Journal of Computer and System Sciences* (JCSS), 2017, Vol. 90, pp. 63–79. Preliminary version appeared in the proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC) 2016.

2. Space Efficient Linear Time Algorithms for BFS, DFS and Applications, Niranka Banerjee, Sankardeep Chakraborty, Venkatesh Raman and Srinivasa Rao Satti, Accepted to *Theory of Computing Systems* (TOCS), 2018, pp. 1–27, *Online.* Preliminary version appeared in the proceedings of the 22nd International Computing and Combinatorics Conference (COCOON) 2016.

3. Space-efficient algorithms for maximum cardinality search, its applications, and variants of BFS, Sankardeep Chakraborty and Srinivasa Rao Satti, Accepted to *Journal of Combinatorial Optimization* (JCO), 2018, pp. 1–17, *Online.* Preliminary version appeared in the proceedings of the 23rd International Computing and Combinatorics Conference (COCOON) 2017.

Conferences

1. Improved Space-efficient Linear Time Algorithms for Some Classical Graph Problems, Sankardeep Chakraborty, Seungbum Jo and Srinivasa Rao Satti, In Proceedings of the 15th Cologne Twente Workshop on Graphs and Combinatorial Optimization (CTW), 2017.

2. Time-Space Tradeoffs for Dynamic Programming Algorithms in Trees and Bounded Treewidth Graphs, Niranka Banerjee, Sankardeep Chakraborty, Venkatesh Raman, Sasanka Roy and Saket Saurabh, In Proceedings of the 21st Annual International Computing and Combinatorics Conference (COCOON), 2015.

Others

1. Two Frameworks for Designing In-Place Graph Algorithms, Sankardeep Chakraborty, Anish Mukherjee, Venkatesh Raman and Srinivasa Rao Satti, Manuscript 2017.

Sankar Deep Chakraborty

To my parents.

ACKNOWLEDGEMENTS

Firstly, I wish to thank my advisor Venkatesh Raman for his advice, encouragement, understanding and support throughout the years. Venkatesh taught me both basic and advanced data structure courses during my first and second semester at IMSc, and I found the subject beguiling. Research with him was only more so, both because of his choice of problems and his way of attacking them. He also gave me complete freedom to consider a variety of topics, and was always there to help me see the big picture in everything, while preventing me from neglecting important details. This thesis would not have been possible without his supervision and support. Thank you Venkatesh, for all your support. I owe a lot to you, not only for your scientific advice but also for all the generous helps that I have received from you throughout these years. I could not have asked for a better advisor.

Secondly, I would like to thank Srinivasa Rao Satti for being my "friend, philosopher and guide". He was always there to discuss anything, be it the problem we were working on or otherwise. His breadth of knowledge in data structures has inspired me to broaden my own horizons. His ideas and insights have also greatly contributed to this thesis. I also thank him for inviting me to visit Seoul and being a perfect host. I have thoroughly enjoyed all the food that Jyoti prepared and all the sightseeing that we went to. I learned a great deal from you, and thanks for that.

I also want to thank all the faculty members of the TCS group: V. Arvind, Kamal Lodaya, Meena Mahajan, R. Ramanujam, Saket Saurabh, C.R. Subramanian, Venkatesh Raman, Vikram Sharma for their wonderful courses and encouragement.

I have been fortunate enough to have some great friends at IMSc, and I would like to thank all of them. It is for them that my stay at IMSc was truly memorable. I want to thank Gaurav for coming with me in numerous overnight cycling trips to Mahabalipuram, Kolli hills and other places. I want to thank Niranka and Anish for being my coauthors, travel buddies and for all the interesting discussions we had, academics and mostly otherwise. I also want to specially thank Varun, Joydeep, Anil, Diptapriyo, Anish for all their help and time. Special thank goes to Nidhi. Had it not been for you I may not have had the opportunity to know (and thoroughly enjoy) about many of the exquisite vegetable cuisines and different ice creams flavours and brands. And thanks also for all the long walks and discussions! I want to especially thank friends from my college days – Biman, Aveek, Monosij, Anjan, Saikat, Ananya for always being there when I needed them for any help or discussions. Thanks for putting up with me this long. Thank you Junhee for taking me to many places of Seoul and making me taste some spicy Korean foods.

Finally, it was all made possible for me by my family's continual and unconditional love and support. I want to thank my sister for her love and care. I am deeply indebted to my parents for being there always, for loving me, caring for me and encouraging me to pursue whatever I am interested in. I especially admire my mother for all her courage after the sad demise of my father during my Ph.D. I do not know whether it is possible for me to return an iota of what I have received from them - perhaps it is not. Still, I will feel content to dedicate this little piece of work to them, and I do so with humility.

Contents

Synopsis **21**

List of Figures **37**

1 Introduction **39**

1.1 Introduction 39

1.1.1 ROM and In-place model 40

1.1.2 Other related models 41

1.1.3 Motivation 42

1.2 Our results and organization of the thesis 44

1.2.1 Space Efficient Linear Time Algorithms 44

1.2.2 Time Efficient Linear Bits Algorithms 46

1.2.3 Space Efficient Algorithms for Optimization Problems in Bounded Treewidth Graphs 48

1.2.4 In-place Graph Algorithms 50

1.3 Conclusion 52

2 Preliminaries **53**

2.1 Graph theoretic terminology 53

2.2 Input graph representations 55

3 Space Efficient Linear Time Algorithms for BFS, DFS and Applications **59**

3.1 Introduction 59

3.1.1 Our results and organization of this chapter 60

3.1.2 Related work 63

3.1.3 Preliminaries 63

3.2 Maintaining dictionaries under findany operation 64

3.2.1 Findany dictionary 65

3.2.2 Extension 69

3.3 Breadth First Search (BFS) 70

3.3.1 Using $2n + o(n)$ bits 70

3.3.2 Using $n \lg 3 + o(n)$ bits 71

3.4 Minimum Spanning Tree (MST) 74

3.5 DFS and its applications using $O(m + n)$ bits 76

3.5.1 DFS 76

3.5.2 Applications of DFS 79

3.6 DFS and its applications using $O(n \lg(m/n))$ bits 84

3.7 Other applications using $O(n \lg(m/n))$ bits 88

3.8 DFS and its applications using $O(n \lg \lg n)$ bits 89

3.9 Concluding remarks and open problems 91

4 Time Efficient Linear Bits Algorithms for DFS, MCS and Applications 93

4.1 Introduction 93

4.1.1 Our results and organization of the chapter 94

4.1.2 Preliminaries 96

4.2 Some simple applications of DFS using $O(n)$ bits 96

4.2.1 Strongly Connected Components 97

4.2.2 Topological Sort 98

4.2.3 Finding a sparse biconnected subgraph of a biconnected graph . . 100

4.3 Tree cover and space efficient construction 104

4.4 Applications of DFS using tree-covering technique 109

4.4.1 Chain decomposition 110

4.4.2 Testing biconnectivity and finding cut vertices 112

4.4.3 Testing 2-edge connectivity and finding bridges 115

4.4.4 *st*-numbering 117

4.5 Applications of *st*-numbering 120

4.5.1 Two-partitioning problem 120

4.5.2 Vertex-subset-two-partition problem 121

4.5.3 Two independent spanning trees 122

4.6 Maximum Cardinality Search (MCS) 123

4.6.1 The MCS algorithm and its implementation 124

4.7 Space efficient implementations of MCS 126

4.8 Applications of MCS 129

4.8.1 Connection with other problems 129

4.8.2 Finding Independent set, Vertex cover and Proper coloring 131

4.8.3 Recognition of chordal graphs 133

4.9 Concluding remarks and open problems 135

5 Space Efficient Algorithms for Optimization Problems in Bounded Treewidth Graphs 137

5.1 Introduction 137

5.1.1 Motivation 138

5.1.2 Related Work 139

5.1.3 Problem Definitions 141

5.1.4 Organization of the chapter 142

5.2 Generalized Stack Framework 143

5.3 DFS on trees 149

5.4 Optimization problems 151

5.4.1 Unweighted Trees 152

5.4.2 Weighted Trees 155

5.4.3 Reporting Solution Set in the Weighted Case 159

5.5 Algorithms for graphs of bounded treewidth 161

5.5.1 Treewidth 162

5.5.2 Counting Monadic Second Order Logic. 163

5.5.3 k-terminal recursive graphs 166

5.5.4 Regular properties 167

5.6 Concluding remarks and open problems 175

6 Two Frameworks for Designing In-place Graph Algorithms 177

6.1 Introduction 177

6.1.1 In-place model for graph algorithms 179

6.1.2 The complexity of BFS and DFS 181

6.1.3 Our main results and the organization of this chapter 182

6.1.4 Techniques 185

6.1.5 Simulations across models 186

6.1.6 Consequences of our BFS and DFS results 186

6.2 DFS Algorithms in the rotate model 188

6.2.1 Lex-DFS using $n \lg 3 + O(\lg^2 n)$ bits for undirected graphs 189

6.2.2 Lex-DFS using $n \lg 3 + O(\lg^2 n)$ bits for directed graphs 190

6.2.3 General-DFS using $n + O(\lg n)$ bits for undirected graphs 191

6.2.4 General-DFS using $n + O(\lg n)$ bits for directed graphs 193

6.2.5 General-DFS using $O(\lg n)$ bits for undirected graphs 193

6.2.6 General-DFS using $O(\lg n)$ bits for directed graphs 195

6.3 BFS Algorithms in the rotate model 199

6.3.1 BFS using $n + O(\lg n)$ bits 200

6.3.2 BFS using $O(\lg n)$ bits 201

6.4 Simulation of algorithms for rotate model in the implicit model 202

6.5 DFS Algorithms in the implicit model 204

6.5.1 Lex-DFS using $O(\lg n)$ bits 205

6.5.2 General-DFS using $O(\lg n)$ bits 206

6.6 BFS algorithms in the implicit model 207

6.7 Minimum Spanning Tree 210

6.8 Improved algorithm for DFS and applications in read-only memory . . . 213

6.9 Solving NP-hard problems in in-place models 215

6.10 Concluding remarks and open problems 216

7 Conclusion **219**

Bibliography **222**

SYNOPSIS

Introduction

In the last several years, the sharp decrease in the cost of memory in computer systems has temporarily taken the focus away from space efficient algorithms. However, the proliferation of specialized handheld devices with limited supply of memory and the astronomical explosion of data has brought the focus back again on the need to pay attention to the memory usage of algorithms. Even if mobile devices and embedded systems are designed with large supply of memory, it might be useful to restrict the number of write operations. For example, on flash memory, writing is a costly operation in terms of speed, and it also reduces the reliability and longevity of the memory. Write-access to removable memory devices might also be limited for technical or security reasons. Whenever multiple concurrent algorithms are working on the same data set, write operations also become troublesome and complicated due to concurrency problems. Keeping all these constraints in mind, it makes sense to consider algorithms that do not modify the input and use only a limited amount of work space. Although many variations of this principle exist in the literature, the general idea remains the same: the input is in some kind of read-only data structure, the output must be produced in a write-only structure, and in addition to these two structures, we can only use a fixed amount of memory to compute the solution. This memory should be enough to cover all space requirements of the algorithm (including all the local and global variables used by the algorithm, space used during recursion, invoking various procedures, etc.). In the following we list the most commonly considered limitations that have received considerable attention in the literature for both the input and the workspace.

- One of the most restrictive models that has been considered is the *one-pass* (or *streaming*) model. In this setting the elements of the input can only be scanned once in a sequential fashion. These algorithms have limited memory available to them (much less than the input size and typically of size $O(n^{\alpha})$ bits where α is less than one or even $O(\text{poly} \lg n)$ bits[1] where n is the input size) and also limited processing time per item. Given these limitations, the usual aim is to approximate the solution and ideally obtain some kind of worst-case approximation ratio with respect to the optimal solution. The natural extension of the above constraint is the *multi-pass* model, in which the input can be scanned sequentially a constant number of times. In here we look for trade-off between the number of passes and either the size of the workspace or the quality of the approximation. Early work on these models focus on processing numerical data such as estimating quantiles, heavy hitters, or the number of distinct elements in the stream [4, 17, 77], and later focused on graph problems [91, 106]. Another relaxation of this setting is the *semi-streaming* model where most of the work on graph streams has occurred in the last decade [76, 114]. In this model the data stream algorithm is permitted $O(\text{n poly} \lg n)$ bits of space where n is the number of nodes in the graph. This is because most problems are provably intractable if the available space is sub-linear in n, whereas many problems become feasible once there is memory roughly proportional to the number of nodes in the graph. See [106] for more details.

- The next natural step is to allow input to be scanned any number of times and even allowing random access to the input values. Thus in the *random access read-only* memory (ROM) model, we assume that the input is given in a read-only memory

[1]We use lg to denote logarithm to the base 2 throughout this thesis.

which can be randomly accessed along with a limited random access workspace and the output of an algorithm is written on to a separate write-only memory which can not be read or modified again. The data on this workspace is manipulated wordwise as on the standard word RAM (i.e. random access memory), where the machine consists of words of size $\Omega(\lg n)$ bits, and any logical, arithmetic, and bitwise operations involving a constant number of words take a constant amount of time. We count space in terms of the number of bits used by the algorithms in workspace. Generally research for this model focuses on either computability (i.e., determining whether or not a particular problem is solvable with a workspace of fixed size) or the design of efficient algorithms whose running time is not much worse (when compared to the case in which no space constraints exist). Early work on this model was on designing lower bounds [21, 29, 30], for designing algorithms for selection and sorting [45, 70, 82, 110, 111, 116] and problems in computational geometry [8, 11, 18, 44, 57].

- A more relaxed model considered is the *in-place* model where the input elements are given in an array, and the algorithm may use the input array as working space. Hence, the algorithm may modify (i.e. rearrange or sometimes even overwrite) the input array during its execution. After the execution, all the input elements should be present in the array (maybe in a permuted order) and the output may be put in the same array or sent to an output stream. The amount of extra space usage during the entire execution of the algorithm is limited to $O(\lg n)$ bits, although sometimes poly-logarithmic words of extra space is also allowed. By making an appropriate permutation of the input, we can usually encode different data structures. Thus, algorithms under this model often achieve the running times comparable to those in unconstrained settings. A prominent example of an in-place algorithm is the classic heap-sort. Other than in-place sorting [81], searching [79, 113] and selection [103] algorithms, many in-place algorithms have been designed in areas such as computational geometry [34] and stringology [80].

- Chan et al. [46] introduced the *restore* model which is a more relaxed version of read-only memory (and a restricted version of the in-place model), where the input is allowed to be modified, but at the end of the computation, the input has to be restored to its original form. They also gave space efficient algorithms for selection and sorting on integer arrays in this model. This has motivation, for example, in scenarios where the input (in its original form) is required by some other application.

- Buhrman et al. [35, 36, 101] introduced and studied the *catalytic-space* model where a small amount (typically $O(\lg n)$ bits) of clean space is provided along with additional auxiliary space, with the condition that the additional space is initially in an arbitrary, possibly incompressible, state and must be returned to this state when the computation is finished. The input is assumed to be given in ROM. They show various interesting complexity theoretic consequences in this model and designed significantly better (in terms of space) algorithms in comparison with ROM model.

Even though these models were introduced in the literature with the aim of designing and/or implementing various algorithms space efficiently, *space efficient graph algorithms* have been designed only in the (semi)-streaming and ROM model. In the streaming and semi-streaming model, researchers have studied several basic and fundamental algorthmic problems such as connectivity, minimum spanning tree, matching. See [106] for a comprehensive survey in this field. Research on these two models (i.e., streaming and semi-streaming) is relatively new and has been going on for last decade or so whereas the study in ROM could be traced back to almost 40 years ago. In fact there is already a rich history of designing space efficient algorithms in the read-only memory model. The complexity class L (also known as $\mathsf{DLOGSPACE}$) is the class containing decision problems that can be solved by a deterministic Turing machine using only logarithmic amount of work space for computation. There are several important algorithmic results [59, 64, 65, 66] for this class, the most celebrated being Reingold's method [121] for checking *st*-reachability in an undirected graph, i.e., to determine if there is a path between two given vertices s

and t. NL is the non-deterministic analogue of L and it is known that the *st*-reachability problem for *directed* graphs is NL-complete (with respect to log space reductions). Using Savitch's algorithm [7], this problem can be solved in $n^{O(\lg n)}$ time using $O(\lg^2 n)$ bits. Savitch's algorithm is very space efficient but its running time is superpolynomial. Among the deterministic algorithms running in polynomial time for directed *st*-reachability, the most space efficient algorithm is due to Barnes et al. [19] who gave a slightly sublinear space (using $n/2^{\Theta(\sqrt{\lg n})}$ bits) algorithm for this problem running in polynomial time. We know of no better polynomial time algorithm for this problem with better space bound. Moreover, the space used by this algorithm matches a lower bound on space for solving directed *st*-reachability on a restricted model of computation called Node Naming Jumping Automata on Graphs (NNJAG) [50, 63]. This model was introduced especially for the study of directed *st*-reachability and most of the known sublinear space algorithms for this problem can be implemented on it. Thus, to design any polynomial time ROM algorithm taking space less than $n/2^{\Theta(\sqrt{\lg n})}$ bits requires significantly new ideas. Recently there has been some improvement in the space bound for some special classes of graphs like planar and H-minor free graphs [10, 37].

A drawback, however, for all these graph algorithms using small space i.e., sublinear bits, is that their running time is often some polynomial of high degree. For example, to the best of our knowledge, the exact running time of Reingold's algorithm [121] for undirected s-t connectivity is not analysed, yet we know it admits a large polynomial running time. In fact this phenomenon is not unusual, as Edmonds et al. [63] have shown in the so-called NNJAG model that only a slightly sublinear working-space bound is possible for an algorithm that solves the reachability problem when required to run in polynomial time. Tompa [132] showed a surprising result that for directed s-t connectivity, if the number of bits available is $o(n)$ then some natural algorithmic approaches to the problem require superpolynomial time.

Motivated by these impossibility results from complexity theory and inspired by the practical applications of these fundamental graph algorithms, recently there has been a surge of interest in improving the space complexity of the fundamental graph algorithms without paying too much penalty in the running time i.e., reducing the working space of the classical graph algorithms to $O(n)$ bits with little or no penalty in running time. Thus the goal is to design space-efficient yet reasonably time-efficient graph algorithms on ROM model. Generally most of the classical linear time graph algorithms take $O(n)$ words or equivalently $O(n \lg n)$ bits of space. This field started with the paper by Asano et al. [9] where they showed that a depth-first search (DFS) can be performed in ROM using $O(m \lg n)$ time and $O(n)$ bits of space, where n and m denote the the number of vertices and edges, respectively, of the input graph. This is followed by the paper of Elmasry et al. [69] where they obtained space efficient algorithms for several other fundamental graph problems including BFS, minimum spanning tree, (strongly) connected components, topological sort etc. In this thesis, not only do we improve these results, we also design and present new results for various other fundamental graph algorithms. These results can be categorized primarily into two types:

- In one direction, we improved the space bounds of several fundamental graph algorithms while keeping the runtimes asymptotically same as in the classical setting in whose design economy of space was not a primary concern. This includes performing BFS, DFS in sparse graphs and determining the 2-edge connected and biconnected components, performing topological sort etc.
- On the other hand, we focused on obtaining linear (i.e., $O(n)$) bit or sometimes even sublinear bits algorithms (improving upon the $O(n \lg n)$ bits classical implementation) for various fundamental graph algorithms without paying too much penalty in the running time. This includes DFS for dense graphs, strong connectivity, *st*-numbering, various optimization problems on bounded treewidth graphs.

Most of these aforementioned fundamental polynomial time graph algorithms on a graph with n vertices seem to need almost $\Theta(n)$ bits of space in the ROM model. In order to break this inherent space bound barrier we sought inspiration from the classical in-place sorting algorithms, and devised two frameworks for designing efficient in-place graph algorithms (i.e., with $O(\lg n)$ bits of extra space). To the best of our knowledge, this has not been done in the literature before. We designed various fundamental graph algorithms in this model and showed many surprising complexity theoretic consequences of such results. In the next section, we describe the contents/organization of the thesis and also briefly mention the main results obtained.

Thesis Outline and Main Results

In this section, we describe the results in the chapters of the thesis in more detail. The technical content of the thesis is organized in the chapters which are divided into roughly four parts as described in the subsections below.

Space Efficient Linear Time Algorithms

Recent works by Asano et al. [9] and Elmasry et al. [69], reconsidered classical fundamental graph algorithms focusing on improving the space complexity in ROM. Elmasry et al. gave, among others, an implementation of breadth first search (BFS) in a graph G with n vertices and m edges, taking the optimal $O(m + n)$ time using $O(n)$ bits of space improving the naïve $O(n \lg n)$ bits implementation. Similarly, Asano et al. provided several space efficient implementations for performing depth first search (DFS) in a graph G. We continue this line of work focusing on improving the space requirement of several fundamental graph algorithms while keeping the runtimes asymptotically same as in the classical setting in the ROM model.

Towards this goal our first result is a simple data structure that can maintain any subset S of a universe of n elements using just $n+o(n)$ bits and supports in constant time, apart from the standard insert, delete and membership queries, the operation *findany* that finds and returns any element of the set (or outputs that the set is empty). It can also enumerate all elements present currently in the set in no particular order in $O(k+1)$ time where k is the number of elements currently belonging to the set. While this data structure supports a weaker set of operations than that of Elmasry et al. [69], it is simple, more space efficient and is sufficient to support a BFS implementation optimally in $O(m+n)$ time using at most $2n+o(n)$ bits. Later, we further improve the space requirement of BFS to at most $n \lg 3 + o(n)$ bits albeit with a slight increase in running time to $O(m \lg n f(n))$ time where $f(n)$ is any extremely slow growing function of n, and the o term in the space is a function of $f(n)$.

For DFS in a directed or undirected graph G, we provide an implementation taking $O(m+n)$ time and $O(n \lg(m/n))$ bits in ROM. This partially answers at least for sparse graphs (where $m = O(n)$), a question asked by Asano et al. [9] whether DFS can be performed in $O(m+n)$ time and using $O(n)$ bits, and also simultaneously improves (for sparse graphs) the DFS result of Elmasry et al. [69]. Building on top of this DFS algorithm and other observations, we show how to efficiently compute the *chain decomposition* of a connected undirected graph. This lets us perform a variety of applications of DFS (including testing biconnectivity and 2-edge connectivity, finding cut vertices and edges among others) within the same time and space bound. Our algorithms for these applications improve the space requirement (for sparse graphs) of all the previous algorithms from $\Omega(n \lg n)$ bits to $O(n)$ bits, preserving the same linear runtime. For the dense graphs (where $m = O(n^2)$), we show that all these applications can be performed using $O(n \lg \lg n)$ bits and linear time whereas earlier classical linear time algorithms for these problems used $\Omega(n \lg n)$ bits of space. These results are reported in [14, 16, 38].

Time Efficient Linear Bits Algorithms

Continuing our pursuit on designing space efficient graph algorithms, we focus this time in our work on designing fast/time efficient algorithms taking linear i.e., $O(n)$ bits for the classical applications of DFS in the ROM model. Starting point of this came from a recent paper of Asano et al. [9] who showed that Depth First Search (DFS) in a directed or an undirected graph can be performed in $O(m \lg n)$ time and $O(n)$ bits of space. Elmasry et al. [69] improved the time to $O(m \lg \lg n)$ still using $O(n)$ bits of space. We build upon these results to give space efficient implementations of several classical applications of DFS. First, as a warm up, we start with some simple applications of the space efficient DFS to show the following.

- An $O(m \lg n \lg \lg n)$ time and $O(n)$ bits of space algorithm to compute the strongly connected components of a directed graph.

In addition, we also give

- an algorithm to output the vertices of a directed acyclic graph in a topologically sorted order, and
- an algorithm to find a sparse (with $O(n)$ edges) spanning biconnected subgraph of an undirected biconnected graph.

both using asymptotically the same time and space used for DFS, i.e., using $O(n)$ bits and $O(m \lg \lg n)$ time.

To develop fast and space efficient algorithms for other non-trivial graph problems which are also applications of DFS, we develop a space efficient tree covering technique which, roughly speaking, partitions the DFS tree into connected smaller sized subtrees which can be stored using less space. Finally we solve the corresponding graph problem on these smaller sized subtrees and merge the solutions across the subtrees to get an overall solution. All of these can be done using less space and not paying too much penalty in the running time. Some of these ideas are borrowed from succinct tree representation literature.

As the first application, we consider a space efficient implementation of chain decomposition of an undirected graph. This is an important preprocessing routine for an algorithm to find cut vertices, biconnected components, cut edges, and also to test 3-connectivity [123] among others. We provide an algorithm that takes $O(m \lg^2 n \lg\lg n)$ time using $O(n)$ bits of space, improving on previous implementations that took $\Omega(n \lg n)$ bits [124] or $\Theta(m + n)$ bits [14] or $O(n \lg m/n)$ bits [41] of space. This is follwed by the improved space efficient algorithms for testing whether a given undirected graph G is biconnected and/or 2-edge connected, and if G is not biconnected and/or 2-edge connected, we also show how one can find all the cut vertices and/or bridges of G. For this, we provide a space efficient implementation of Tarjan's classical lowpoint algorithm [128]. Our algorithms take $O(m \lg n \lg\lg n)$ time and $O(n)$ bits of space.

Given a biconnected graph, and two distinguished vertices s and t, st-numbering is a numbering of the vertices of the graph so that s gets the smallest number, t gets the largest and every other vertex is adjacent both to a lower-numbered and to a higher-numbered vertex. Finding an st-numbering is an important preprocessing routine for a planarity testing algorithm [72] among others. We present an algorithm to determine an st-numbering of a biconnected graph that takes $O(m \lg^2 n \lg\lg n)$ time using $O(n)$ bits. This improves the earlier implementations that take $\Omega(n \lg n)$ bits [31, 62, 72, 130]. Using this as a subroutine, we also provide improved space effcient implementation for two-partitioning and two independent spanning tree problem among others.

Moving on from the space efficient implementations of DFS and its several applications in ROM, this time we focus on designing space efficient algorithms for another graph method, which is known in the literature as Maximum Cardinality Search (MCS), and a few of its applications. Tarjan, in an unpublished note [127], defined MCS, and later

Tarjan and Yannakakis [131] presented its applications to recognize chordal graphs and test acyclicity of hypergraphs, etc. We refer the interested readers to the excellent text of Golumbic [89] for thorough coverage of chordal graph recognition, MCS and many other related topics. We show that using only $O(n)$ bits and $O(m^2/n)$ time in ROM, we can perform MCS in a given input graph G, improving on the naive $O(n \lg n)$ bit classical implementation. Using this as a subroutine, we also provide improved space efficient implementations for finding an independent set, vertex cover, proper coloring etc in a given chordal graph. These results are reported in [40, 41, 42, 43].

Space Efficient Algorithms for Optimization Problems in Bounded Treewidth Graphs

Barba et al. [18] introduce the *compressed stack technique*, a procedure to transform algorithms whose main memory consumption takes the form of a stack into memory-constrained algorithms in ROM, and show various applications of this method by designing space efficient algorithms for problems in computational geometry. In what follows, we briefly explain the set up and state their main result.

Let A be a class of deterministic algorithms which uses a stack and optionally other auxiliary data structures of constant size. The operations that can be supported are push, pop and accessing the k topmost element of the stack for a constant k. Let $x \in A$ be any deterministic algorithm. We call x a stack based algorithm if, given a set of ordered input I, x processes $i \in I$ one by one in order and based on i, the top k elements (for some constant k) of the stack and the auxiliary data structure's current configuration, it decides to either push i (or some function of i and some constant words of information) or pop some elements off the stack. While popping some element v off the stack, x can also output v (or some function of v) as a part of the final solution for the problem considered. In [18], they assume that the output is what is left in the stack at the end, while we use an output array to output elements during the execution of the algorithm itself. Any algorithm following this structure is called a *stack based algorithm.* The main result of their paper is the following:

Theorem 0.1. *Any stack algorithm which takes* $O(n)$ *time and* $\Theta(n)$ *space can be adapted so that, for any parameter* $2 \leq p \leq n$*, it solves the problem in* $O(n^{1+(1/\lg p)})$ *time using* $O(p \lg_p n)$ *variables.*

We explore applying the technique mentioned above for various optimization problems in trees and bounded treewidth graphs in the ROM model. En route we also generalize the above mentioned stack compression framework to a broader class of algorithms, which we believe can be of independent interest [15] and may find other applications as well. Using this extended stack compression framework, we show that we can solve various optimization problems like *Minimum vertex cover, Maximum independent set, Minimum Dominating set* etc in bounded treewidth graphs using $O(\lg^2 n)$ bits of space. In fact we prove the following more general meta theorem which roughly says, for bounded treewidth graphs, if any graph problem can be described in monadic second order (MSO) logic, we can obtain a smooth deterministic time-space trade-off from logarithmic words to linear space. Our result can be seen as a generalization of the results of Elberfeld et al. [64] and the famous Courcelle's theorem [53]. Broadly speaking, the well-known Courcelle's theorem states that many graph properties (that are expressible in monadic second order logic) can be solved in linear time on graphs of bounded treewidth. Elberfeld et al. [64] show the logspace version of the same result. We develop an alternate methodology using the standard table-based dynamic programming approach and extended stack-compression technique to give a space efficient version (using $O(\lg^2 n)$ bits of space) of Courcelle's theorem. We only need the input graph and its tree decomposition to be given in ROM such a way that navigating to the parent, leftmost child and right sibling can be done in constant time. While it has been known that the more general Courcelle's theorem can be implemented in logarithmic space and an unspecified yet large polynomial running time (using the result of Elberfeld et al. [64]), our algorithms provide a much simpler, cleaner and more practical approach for these problems using much less time with slightly more space. These results are reported in [15].

In-place Graph Algorithms

Read-only memory (ROM) model is a classical model of computation to study time-space tradeoffs of algorithms. One of the classical results on the ROM model is that any sorting algorithm that uses $O(s)$ words of extra space requires $\Omega(n^2/s)$ comparisons for $\lg n \leq s \leq n/\lg n$ and the bound has also been recently matched by an algorithm. However, if we relax the model (from ROM), we do have sorting algorithms (say Heapsort) that can sort using $O(n \lg n)$ comparisons using $O(\lg n)$ bits of extra space, even keeping a permutation of the given input sequence at any point of time of the algorithm. Such a model is known in the literature as in-place model. Even though sorting, searching, selection and many other algorithms in computational geometry and stringology are known in-place, nothing is known, to the best of our knowledge, for graph algorithms in such a setting.

In our work we initiate a systematic study of designing efficient in-place (i.e., $O(\lg n)$ extra bits) algorithms for fundamental graph problems. In fact we show that a simple natural relaxation of ROM model allows us to beat, by exponential margin, the ROM space bounds for several fundamental graph algorithms like DFS, BFS, shortest path etc in this new model. By simply allowing elements in the adjacency list of a vertex to be permuted, we show that, on an undirected connected graph G having n vertices and m edges, the vertices of G can be output in a

- DFS order using $O(\lg n)$ bits of extra space and $O(m^2/n)$ time if the graph is given in an adjacency list, and in $O(m^2 \lg n/n)$ time if the graph is given in an adjacency array;
- BFS order using $O(\lg n)$ bits of extra space and $O(m)$ time if all vertices have degree

at least $2 \lg n + 3$, in $O(n^2)$ time if there are no degree 2 vertices, and in $O(n^3)$ time otherwise.

Most of these results carry over to directed graphs too, with a slight degradation in running time. Thus we obtain similar bounds for *reachability* and *shortest path distance* (both for undirected and directed graphs). With a little more (but still polynomial) time, we can also output vertices in the *lex-DFS* order. As reachability in directed graphs (even in DAGs) and shortest path distance (even in undirected graphs) are NL-complete problems, and lex-DFS is P-complete, our results show that our model is probably more powerful than ROM.

En route, we introduce and develop algorithms for another relaxation of ROM where the adjacency lists of the vertices are circular lists and we can only modify the heads of the lists. Here we first show a linear time DFS implementation using $n + O(\lg n)$ bits. Improving the space further to only $O(\lg n)$ bits, we also obtain BFS and DFS albeit with a slightly slower running time. Some of these algorithms also translate to improved algorithms for DFS and its applications in ROM. Both the models we propose maintain the graph structure throughout the algorithm, only the order of vertices in the adjacency list changes.

In sharp contrast, for BFS and DFS, to the best of our knowledge, there are no algorithms in ROM that use even $O(n^{1-\epsilon})$ bits of space; in fact, implementing DFS using cn bits for $c < 1$ has been mentioned as an open problem [9]. Furthermore, DFS (BFS) algorithms using $n + o(n)$ (or $o(n)$) bits use Reingold's or Barnes et al's reachability algorithm and hence have high runtime. All our algorithms are simple but quite subtle, and we believe that these models are practical enough to spur interest for other graph problems in these models. These results are reported in [39].

Conclusion

We conclude this synopsis by higlighting our main results and mentioning some open problems below.

- First, we show $O(m+n)$ time and $O(n)$ bit algorithms for BFS, DFS and many of its applications including biconnectivity, 2-edge connectivity, *st*-numbering etc for sparse graphs in ROM. Under some reasonable complexity theoretic assumption, this is the best we can hope for. One very general yet challenging and important open problem in this direction is, can we design sublinear bits algorithms that are reasonably time efficient (i.e., low degree polynomial running time) for all these problems?

- Next, we present $O(\text{m poly} \lg n)$ time and $O(n)$ bit algorithms for DFS and many of its fundamental applications including strong connectivity, topological sorting, biconnectivity etc in the case of dense graphs in ROM. One immediate open problem in this regard is, can we shave off the poly-log factor in the running time from all of our algorithms while retaining the same space bound? Note that, from the previous point, we can achieve such results when the input graph is sparse. A rather challenging problem would be to obtain a sublinear space algorithm with low polynomial running time for these problems.

- Then we provide a simple algorithm showing smooth time-space tradeoffs for various optimization problems on bounded treewidth graphs generalizing the results of Elberfeld et al. [64] and Courcelle [53] in the ROM model. Our algorithm is optimal upto a log factor in the space bound and particularly time efficient. It remains an open problem to make it optimal from space point of view i.e., shave off the multiplicative log factor from the space bound with very little or no compromise in the running time.

- Finally, we formulate two new frameworks for designing in-place graph algorithms for the first time in the literature and exemplify its power by designing efficient algorithms for various fundamental graph algorithms including DFS, BFS, minimum spanning tree etc. In stark contrast to the best space bounds in ROM, these algorithms are very economic in both space and time requirement. One broad future goal in this direction would be to expand the horizon of graph algorithms that can be designed in-place.

List of Figures

3.1 An illustration of the inner working details of our findany data structure. The precomputed table is not shown in the diagram. 67

3.2 Illustration of Chain Decomposition. (a) An input graph G. (b) A DFS traversal of G and the resulting edge-orientation along with DFIs. (c) A chain decomposition D of G. The chains D_2 and D_3 are paths and rest of them are cycles. The edge (V_5, V_6) is bridge as it is not contained in any chain. V_5 and V_6 are cut vertices. 81

4.1 A part of the full DFS tree. The wiggling edges represent tree edges and the edges with arrow heads represent back edges. If $low(v_i) = v_a$, we would come across v_i in the adjacency array of v_a before encountering from the arrays of v_b and v_c. I.e., the back edge (v_a, v_i) will be processed before the other back edges (v_b, v_i) and (v_c, v_i) since we process the vertices (and the backedges incident to them) in their DFS order. 102

4.2 An illustration of Tree Covering technique with $L = 5$. The figure is reproduced from [74]. Each closed region formed by the dotted lines represents a minitree. Note that each minitree has at most one 'child' minitree (other than the minitrees that share its root) in this structure. 105

4.3 (a) The minitree structure of the tree decomposition shown in Figure 2. (b) This array encodes the entire DFS tree using the balanced parenthesis (BP) representation. (c) In this array, we demonstrate how the minitrees are split into a constant number of consecutive chunks in the BP representation. Note that the bottom array can actually be encoded using $O(\lg^2 n)$ bits, by storing, for each of the $O(\lg n)$ minitrees, pointers to all the chunks in BP sequence indicating the starting and ending positions of the chunks corresponding to the minitrees. 106

4.4 An input graph G with n vertices from v_1 to v_n. The numbers in the brackets show one possible output of the MCS algorithm on G when it starts with the vertex v_1. The resulting vertex ordering obtained is $v_1, v_3, v_2, v_4, v_5, v_6, v_7, v_9, v_8, v_{11}, v_{10}, v_{12}$. 124

6.1 (a) An undirected graph G with 5 vertices and 8 edges. (b) A circular list representation of G. To avoid cluttering the picture, we draw the vertices and the pointers to the next node separately as opposed to a single node having two different fields in the circular list. (c) An illustration of a single clockwise rotation in the circular list of vertex 4. 180

6.2 Illustration of the different cases of the possible positions of the vertex y when DFS considers the directed edge (x, y) at some intermediate step. Suppose the root of the DFS tree is the vertex s and the curvy path starting from s and going straight below through x is the current gray path in the DFS tree. Intuitively all the vertices on the left hand side of the path are black, and right hand side are white and yet to be explored. From left to the right are cases when (x, y) is (a) back edge, (b) cross edge, (c) forward edge, and (d) tree edge. 197

Chapter 1

Introduction

1.1 Introduction

With technological revolution, while the cost of memory has come down drastically, the amount of data available in applications is growing even more. So algorithms that pay attention to space usage are becoming increasingly important. Furthermore, the proliferation of specialized handheld devices and embedded systems that have a limited supply of memory provides an additional motivation to design and study space efficient algorithms. Even if mobile devices and embedded systems are designed with large supply of memory, it still might be useful to restrict the number of write operations for several reasons. For example, on flash memory, writing is a costly operation in terms of speed. Write-access to removable memory devices might also be limited for technical or security reasons as whenever multiple concurrent algorithms are working on the same data set, write operations also become troublesome and complicated due to concurrency problems. Keeping all these constraints in mind, it makes sense to consider algorithms that do not modify the input and use only a limited amount of work space. Although many variations of this principle exist in the literature, the general idea remains the same: the input is in some kind of read-only data structure, the output must be produced in a write-only structure, and in addition to these two structures, we can only use a fixed amount of memory to compute the solution. This memory should be enough to cover all space requirements during the execution of the algorithm (including all the local and global variables used by the algorithm, and the space used during recursion, and invoking various procedures).

1.1.1 ROM and In-place model

In what follows we describe two of the most commonly considered models that have received considerable attention in the literature for both the input and the workspace as they were historically developed. Also these are the two main models that are used in this thesis for designing space efficient algorithms.

- In the *random access read-only* memory (ROM) model, we assume that the input is given in a read-only memory which can be randomly accessed along with a limited random access workspace and the output of an algorithm is written on to a separate write-only memory which can not be read or modified again. So the input is "read only" and the output is "write only". The data on the workspace is manipulated wordwise as on the standard word RAM (i.e. random access memory), where the machine consists of words of size $\Omega(\lg n)$ bits, and any logical, arithmetic, and bitwise operations involving a constant number of words take a constant amount of time. We count space in terms of the number of bits used by the algorithms in workspace. This model is also called the *register input model*, and it was introduced by Frederickson [82] while studying some problems related to sorting and selection. Mainly research for this model focuses on either computability (i.e., determining whether or not a particular problem is solvable with a workspace of fixed size) or the design of efficient algorithms whose running time is not much worse (when compared to the case in which no space constraints exist). Early work on this model was on designing lower bounds [21, 29, 30], for designing algorithms for selection and sorting [45, 70, 82, 110, 111, 116] and problems in computational geometry [8, 11, 18, 44, 57].

- A more relaxed model considered is the *in-place* model where the input elements are given in an array, and the algorithm may use the input array as working space. Hence, the algorithm may modify (i.e. rearrange or sometimes even overwrite) the input array during its execution. After the execution, all the input elements should be present in the array (maybe in a permuted order) and the output may be put in the same array or sent to an output stream. The amount of extra space usage during the entire execution of the algorithm is limited to $O(\lg n)$ bits, although sometimes poly-logarithmic words of extra space is also allowed. By making an appropriate permutation of the input, we can usually encode different data structures. Thus, algorithms under this model often achieve the running times comparable to those in unconstrained settings. A prominent example of an in-place algorithm is the classic heap-sort. Other than in-place sorting [81], searching [79, 113] and selection [103] algorithms, many in-place algorithms have been designed in areas such as computational geometry [34] and stringology [80].

1.1.2 Other related models

There exist a few other models in the literature for designing space efficient algorithms as well other than the ones mentioned above. Chan et al. [46] introduced the *restore* model which is a more relaxed version of read-only memory (and a restricted version of the in-place model), where the input is allowed to be modified, but at the end of the computation, the input has to be restored to its original form. Buhrman et al. [35, 36, 101] introduced and studied the *catalytic-space* model where a small amount (typically $O(\lg n)$ bits) of clean space is provided along with additional auxiliary space, with the condition that the additional space is initially in an arbitrary, possibly incompressible, state and must be restored to this state when the computation is finished. The input is assumed to be given in ROM. Thus this model can be thought of as having an auxiliary storage that needs to be 'restored' in contrast to the model by Chan et al. [46] where the input array has to be 'restored'. In the *streaming* model (and in its variants, e.g *multi-pass* and *semi-streaming*) the elements of the input can only be scanned in a sequential fashion [4, 76, 106, 110]. These algorithms have limited memory available to them (much less than the input size and typically of size $O(\text{poly} \lg n)$ bits where n is the input size) and also limited processing time per item. Given these limitations, the usual goal is to approximate the solution and ideally obtain some kind of worst-case approximation ratio with respect to the optimal solution.

1.1.3 Motivation

Even though these models were introduced in the literature with the aim of designing and/or implementing various algorithms space efficiently, *space efficient graph algorithms* have been designed only in the (semi)-streaming and ROM model. In the streaming and semi-streaming model, researchers have studied several basic and fundamental algorthmic problems such as connectivity, minimum spanning tree, matching. See [106] for a comprehensive survey in this field. Research on these two models (i.e., streaming and semi-streaming) is relatively new and has been going on for last decade or so whereas the study in ROM could be traced back to almost 40 years. In fact there is already a rich history of designing space efficient algorithms in the read-only memory model. The complexity class L (also known as $\mathsf{DLOGSPACE}$) is the class containing decision problems that can be solved by a deterministic Turing machine using only logarithmic amount of work space for computation. There are several important algorithmic results [59, 64, 65, 66] for this class, the most celebrated being Reingold's method [121] for checking st-reachability in an undirected graph, i.e., to determine if there is a path between two given vertices s and t. NL is the non-deterministic analogue of L and it is known that the st-reachability problem for *directed* graphs is NL-complete (with respect to log space reductions). Using

Savitch's algorithm [7], this problem can be solved in $n^{O(\lg n)}$ time using $O(\lg^2 n)$ bits. Savitch's algorithm is very space efficient but its running time is superpolynomial. Among the deterministic algorithms running in polynomial time for directed *st*-reachability, the most space efficient algorithm is due to Barnes et al. [19] who gave a slightly sublinear space (using $n/2^{\Theta(\sqrt{\lg n})}$ bits) algorithm for this problem running in polynomial time. We know of no better polynomial time algorithm for this problem with better space bound. Moreover, the space used by this algorithm matches a lower bound on space for solving directed *st*-reachability on a restricted model of computation called Node Naming Jumping Automata on Graphs (NNJAG) [50, 63]. This model was introduced especially for the study of directed *st*-reachability and most of the known sublinear space algorithms for this problem can be implemented on it. Thus, to design any polynomial time ROM algorithm taking space less than $n/2^{\Theta(\sqrt{\lg n})}$ bits requires significantly new ideas. Recently there has been some improvement in the space bound for some special classes of graphs like planar and H-minor free graphs [10, 37].

A drawback, however, for all these graph algorithms using small space i.e., sublinear bits, is that their running time is often some polynomial of high degree. For example, to the best of our knowledge, the exact running time of Reingold's algorithm [121] for undirected s-t connectivity is not analysed, yet we know it admits a large polynomial running time. In fact this phenomenon is not unusual, as Edmonds et al. [63] have shown in the so-called NNJAG model that only a slightly sublinear working-space bound is possible for an algorithm that solves the reachability problem when required to run in polynomial time. Tompa [132] showed a surprising result that for directed s-t connectivity, if the number of bits available is $o(n)$ then some natural algorithmic approaches to the problem require superpolynomial time.

Motivated by these impossibility results from complexity theory and inspired by the practical applications of these fundamental graph algorithms, recently there has been a surge of interest in improving the space complexity of the fundamental graph algorithms without paying too much penalty in the running time i.e., reducing the working space of the classical graph algorithms to $O(n)$ bits with little or no penalty in running time. Thus, the goal is to design space-efficient yet reasonably time-efficient graph algorithms on ROM model. Generally most of the classical linear time graph algorithms take $O(n)$ words or equivalently $O(n \lg n)$ bits of space. This field started with the paper by Asano et al. [9] where they showed that a depth-first search (DFS) of a graph G with n vertices and m edges can be performed in ROM using $O(m \lg n)$ time and $O(n)$ bits of space. This is followed by the paper of Elmasry et al. [69] where they obtained space efficient algorithms for several other fundamental graph problems including breadth-first search (BFS), minimum spanning tree (MST), (strongly) connected components and topological sort. In this thesis, not only do we improve these results, we also design and present new results for various other fundamental graph algorithms which we briefly discuss in the next section.

1.2 Our results and organization of the thesis

In this section, we describe the results in the chapters of the thesis in more detail. The technical content of the thesis is organized in the chapters which are divided into roughly four parts as described in the subsections below.

1.2.1 Space Efficient Linear Time Algorithms

Recent works by Asano et al. [9] and Elmasry et al. [69], reconsidered classical fundamental graph algorithms focusing on improving the space complexity in ROM. Elmasry et al. gave, among others, an implementation of breadth first search (BFS) in a graph G with n vertices and m edges, taking the optimal $O(m + n)$ time using $O(n)$ bits of space improving the naïve $O(n \lg n)$ bits implementation. Similarly, Asano et al. provided several space efficient implementations for performing depth first search (DFS) in a graph

G. We continue this line of work focusing on improving the space requirement of several fundamental graph algorithms while keeping the runtimes asymptotically same as in the classical setting in the ROM model.

Towards this goal our first result, in Chapter 3, is a simple data structure that can maintain any subset S of a universe of n elements using just $n + o(n)$ bits and supports in constant time, apart from the standard insert, delete and membership queries, the operation *findany* that finds and returns any element of the set (or outputs that the set is empty). It can also enumerate all elements present currently in the set in no particular order in $O(k + 1)$ time where k is the number of elements currently belonging to the set. While this data structure supports a weaker set of operations than that of Elmasry et al. [69], it is simple, more space efficient and is sufficient to support a BFS implementation optimally in $O(m + n)$ time using at most $2n + o(n)$ bits. Later, we further improve the space requirement of performing BFS to at most $n \lg 3 + o(n)$ bits albeit with a slight increase in the running time to $O(m \lg n f(n))$ time where $f(n)$ is any extremely slow growing function of n, and the o term in the space is a function of $f(n)$.

For DFS in a directed or undirected graph G, we provide an implementation taking $O(m + n)$ time and $O(n \lg m/n)$ bits in ROM. This partially answers at least for sparse graphs (where $m = O(n)$), a question asked by Asano et al. [9] whether DFS can be performed in $O(m + n)$ time and using $O(n)$ bits, and also simultaneously improves (for sparse graphs) the DFS result of Elmasry et al. [69]. Building on top of this DFS algorithm and other observations, we show how to efficiently compute the *chain decomposition* of a connected undirected graph. This lets us perform a variety of applications of DFS (including testing biconnectivity and 2-edge connectivity, finding cut vertices and edges among others) within the same time and space bound. Our algorithms for these applications improve the space requirement (for sparse graphs) of all the previous algorithms from $\Omega(n \lg n)$ bits to $O(n)$ bits, preserving the same linear runtime. For dense graphs (where $m = \Theta(n^2)$), we show that all these applications can be performed using

$O(n \lg\lg n)$ bits and linear time whereas earlier classical linear time algorithms for these problems used $\Omega(n \lg n)$ bits of space. This chapter combines the results of [14, 16, 38].

1.2.2 Time Efficient Linear Bits Algorithms

Continuing our pursuit on designing space efficient graph algorithms, we focus this time in Chapter 4 on designing fast/time efficient algorithms taking linear i.e., $O(n)$ bits for the classical applications of DFS in the ROM model. The starting point of this came from a recent paper of Asano et al. [9] who showed that Depth First Search (DFS) in a directed or an undirected graph can be performed in $O(m \lg n)$ time and $O(n)$ bits of space. Elmasry et al. [69] improved the time to $O(m \lg\lg n)$ still using $O(n)$ bits of space. We build upon these results to give space efficient implementations of several classical applications of DFS. First, as a warm up, we start with some simple applications of the space efficient DFS to show the following.

- An $O(m \lg n \lg\lg n)$ time and $O(n)$ bits of space algorithm to compute the strongly connected components of a directed graph.

In addition, we also give

- an algorithm to output the vertices of a directed acyclic graph in a topologically sorted order, and
- an algorithm to find a sparse (with $O(n)$ edges) spanning biconnected subgraph of an undirected biconnected graph.

both using asymptotically the same time and space used for linear bits DFS, i.e., using $O(n)$ bits and $O(m \lg\lg n)$ time.

To develop fast and space efficient algorithms for other non-trivial graph problems which are also applications of DFS, we develop a space efficient tree covering technique which, roughly speaking, partitions the DFS tree into connected smaller sized subtrees which can be stored using less space. Finally we solve the corresponding graph problem on these smaller sized subtrees and merge the solutions across the subtrees to get an overall solution. All of these can be done using less space and by not paying too much penalty in the running time. Some of these ideas are borrowed from succinct tree representation literature.

As the first application, we consider a space efficient implementation of chain decomposition of an undirected graph. This is an important preprocessing routine for an algorithm to find cut vertices, biconnected components, cut edges, and also to test 3-connectivity [123] among others. We provide an algorithm that takes $O(m \lg^2 n \lg\lg n)$ time using $O(n)$ bits of space, improving on previous implementations that took $\Omega(n \lg n)$ bits [124] or $\Theta(m+n)$ bits [14] or $O(n \lg m/n)$ bits [41] of space. This is followed by the improved space efficient algorithms for testing whether a given undirected graph G is biconnected and/or 2-edge connected, and if G is not biconnected and/or 2-edge connected, we also show how one can find all the cut vertices and/or bridges of G. For this, we provide a space efficient implementation of Tarjan's classical lowpoint algorithm [128]. Our algorithms take $O(m \lg n \lg\lg n)$ time and $O(n)$ bits of space.

Given a biconnected graph, and two distinguished vertices s and t, st-numbering is a numbering of the vertices of the graph so that s gets the smallest number, t gets the largest and every other vertex is adjacent both to a lower-numbered and to a higher-numbered vertex. Finding an st-numbering is an important preprocessing routine for a planarity testing algorithm [72] among others. We present an algorithm to determine an st-numbering of a biconnected graph that takes $O(m \lg^2 n \lg\lg n)$ time using $O(n)$ bits. This improves the earlier implementations that take $\Omega(n \lg n)$ bits [31, 62, 72, 130]. Using this as a subroutine, we also provide improved space effcient implementation for two-partitioning and two independent spanning tree problem among others.

Moving on from the space efficient implementations of DFS and its several applications in ROM, this time we focus on designing space efficient algorithms for another graph method, which is known in the literature as Maximum Cardinality Search (MCS), and some of its applications. Tarjan, in an unpublished note [127], defined MCS, and later Tarjan and Yannakakis [131] presented its applications to recognize chordal graphs and test acyclicity of hypergraphs, etc. We refer the interested readers to the excellent text of Golumbic [89] for thorough coverage of chordal graph recognition, MCS and many other related topics. We show that using only $O(n)$ bits and $O(m^2/n)$ time in ROM, we can perform MCS in a given input graph G, improving on the naive $O(n \lg n)$ bit classical implementation. Using this as a subroutine, we also provide improved space efficient implementations for finding an independent set, vertex cover, proper coloring in a given chordal graph. These results are reported in [40, 41, 42, 43].

1.2.3 Space Efficient Algorithms for Optimization Problems in Bounded Treewidth Graphs

Barba et al. [18] introduce the *compressed stack technique*, a procedure to transform algorithms whose main memory consumption takes the form of a stack into memory-constrained algorithms in ROM, and show various applications of this method by designing space efficient algorithms for problems in computational geometry. In what follows, we briefly explain the set up and state their main result.

Let A be a class of deterministic algorithms which use a stack and optionally other auxiliary data structures of constant size. The operations that can be supported are push, pop and accessing the k topmost element of the stack for a constant k. Let $x \in A$ be any deterministic algorithm. We call x a stack based algorithm if, given a set of ordered input I, x processes $i \in I$ one by one in order and based on i, the top k elements (for some constant k) of the stack and the auxiliary data structure's current configuration, it decides to either push i (or some function of i and some constant words of information) or pop some elements off the stack. While popping some element v off the stack, x can also output v (or some function of v) as a part of the final solution for the problem considered. In [18], they assume that the output is what is left in the stack at the end, while we use an output array to output elements during the execution of the algorithm itself. Any algorithm following this structure is called a *stack based algorithm.* The main result of their paper is the following:

Theorem 1.1. *Any stack algorithm which takes $O(n)$ time and $\Theta(n)$ space can be adapted so that, for any parameter $2 \leq p \leq n$, it solves the problem in $O(n^{1+(1/\lg p)})$ time using $O(p \lg_p n)$ variables.*

In chapter 5, we explore applying the technique mentioned above for various optimization problems in trees and bounded treewidth graphs in the ROM model. En route we also generalize the above mentioned stack compression framework to a broader class of algorithms, which we believe can be of independent interest and may find other applications as well. Using this extended stack compression framework, we show that we can solve various optimization problems like *Minimum vertex cover, Maximum independent set, Minimum Dominating set* etc in bounded treewidth graphs using $O(\lg^2 n)$ bits of space. In fact we prove the following more general meta theorem which roughly says, for bounded treewidth graphs, if any graph problem can be described in monadic second order (MSO) logic, we can obtain a smooth deterministic time-space trade-off from logarithmic words to linear space. Our result can be seen as a generalization of the results of Elberfeld et al. [64] and the famous Courcelle's theorem [53]. Broadly speaking, the well-known Courcelle's theorem states that many graph properties (that are expressible in monadic second order logic) can be solved in linear time on graphs of bounded treewidth. Elberfeld et al. [64] show the logspace version of the same result. We develop an alternate methodology using the standard table-based dynamic programming approach and extended stack-compression technique to give a space efficient version (using $O(\lg^2 n)$ bits of space) of Courcelle's theorem. We only need the input graph and its tree decomposition to be given in ROM in such a way that navigating to the parent, leftmost child and right sibling can be done in constant time. While it has been known that the more general Courcelle's theorem can be implemented in logarithmic space and an unspecified yet large polynomial running time (using the result of Elberfeld et al. [64]), our algorithms provide a much simpler, cleaner and more practical approach for these problems using much less time with slightly more space. This chapter is based on the paper [15].

1.2.4 In-place Graph Algorithms

So far all the results we discussed are in Read-only memory (ROM) model which is a classical model of computation to study time-space tradeoffs of algorithms. One of the early classical results on the ROM model is that any sorting algorithm that uses $O(s)$ words of extra space requires $\Omega(n^2/s)$ comparisons for $\lg n \leq s \leq n/\lg n$ and the bound has also been recently matched by an algorithm. However, if we relax the model (from ROM), we do have sorting algorithms (say Heapsort) that can sort using $O(n \lg n)$ comparisons using $O(\lg n)$ bits of extra space, even keeping a permutation of the given input sequence at any point of time of the algorithm. Such a model is known in the literature as in-place model. Even though sorting, searching, selection and many other algorithms in computational geometry and stringology are known in-place, nothing is known, to the best of our knowledge, for graph algorithms in such a setting.

In our work we initiate a systematic study of designing efficient in-place (i.e., $O(\lg n)$ extra bits) algorithms for fundamental graph problems. In fact, in Chapter 6, we show that a simple natural relaxation of ROM model allows us to beat, by exponential margin, the ROM space bounds for several fundamental graph algorithms like DFS, BFS, minimum spanning tree (MST) in this new model. By simply allowing elements in the adjacency list of a vertex to be permuted, we show that, on an undirected connected graph G having n vertices and m edges, the vertices of G can be output in a

- DFS order using $O(\lg n)$ bits of extra space and $O(m^2/n)$ time if the graph is given in an adjacency list, and in $O((m^2 \lg n)/n)$ time if the graph is given in an adjacency array;

- BFS order using $O(\lg n)$ bits of extra space and $O(m)$ time if all vertices have degree at least $2\lg n + 3$, in $O(n^2)$ time if there are no degree 2 vertices, and in $O(n^3)$ time otherwise.

Most of these results carry over to directed graphs too, with a slight degradation in running time. Thus we obtain similar bounds for *reachability* and *shortest path distance* (both for undirected and directed graphs). With a little more (but still polynomial) time, we can also output vertices in the *lex-DFS* order. As reachability in directed graphs (even in DAGs) and shortest path distance (even in undirected graphs) are NL-complete problems, and lex-DFS is P-complete, our results show that our model is probably more powerful than ROM.

En route, we introduce and develop algorithms for another relaxation of ROM where the adjacency lists of the vertices are circular lists and we can only modify the heads of the lists. Here we first show a linear time DFS implementation using $n + O(\lg n)$ bits. Improving the space further to only $O(\lg n)$ bits, we also obtain BFS and DFS albeit with a slightly slower running time. Some of these algorithms also translate to improved algorithms for DFS and its applications in ROM. Both the models we propose maintain the graph structure throughout the algorithm, only the order of vertices in the adjacency list changes.

In sharp contrast, for BFS and DFS, to the best of our knowledge, there are no algorithms in ROM that use even $O(n^{1-\epsilon})$ bits of space; in fact, implementing DFS using cn bits for $c < 1$ has been mentioned as an open problem [9]. Furthermore, DFS (BFS) algorithms using $n + o(n)$ (or $o(n)$) bits use Reingold's or Barnes et al's reachability algorithm and hence have high runtime. All our algorithms are simple but quite subtle, and we believe that these models are practical enough to spur interest for other graph problems in these models. The results obtained in this chapter are based on a joint work with Anish Mukherjee, Venkatesh Raman and Srinivasa Rao Satti [39].

1.3 Conclusion

Finally we conclude the thesis in Chapter 7 with some closing remarks and open problems for future direction. This thesis is mainly about designing graph algorithms which are both time as well as space efficient. In this chapter we motivated the study of designing such algorithms and provided a summary of the results obtained in this thesis.

Chapter 2

Preliminaries

In this chapter we lay down the notation, terminology and input representations used elsewhere in the thesis, for the sake of easy reference.

2.1 Graph theoretic terminology

Here we follow the graph terminology used in the textbook by Cormen et al. [51]. A *directed graph* (or *digraph*) G is a pair (V, E), where V is a finite set and E is a binary relation on V. The set V is called the *vertex set* of G, and its elements are called *vertices* (singular: *vertex*). Throughout this thesis, we assume that the vertex set V of G is the set $V = \{1, 2, \cdots, n\}$. The set E is called the edge set of G, and its elements are called *egdes*. We use n and m to denote the number of vertices and the number of edges respectively, in the input graph G. I.e., $|V| = n$ and $|E| = m$. In an undirected graph $G = (V, E)$, the edge set E consists of *unordered* pair of vertices, rather than ordered pairs. That is, an edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$. By convention, we use the notation (u, v) for an edge, rather than the set notation $\{u, v\}$, and we consider (u, v) and (v, u) to be the same edge. The *degree* of a vertex in an undirected graph is the number of edges incident on it. In a directed graph, the *out-degree* of a vertex is the number of edges leaving it, and the *in-degree* of a vertex is the number of edges entering it. An undirected graph is *connected* if every vertex is reachable from all other vertices. The connected components of a graph are the equivalence classes of vertices under the "is reachable from" relation.

In an undirected graph G, a *cut vertex* is a vertex v that when removed (along with its incident edges) from a graph creates more (than what was there before) components in the graph. A (connected) graph with at least three vertices is *biconnected* (also called 2-*connected* or 2-*vertex connected* in the literature) if and only if it has no cut vertex. A *biconnected component* is a maximal biconnected subgraph. These components are attached to each other at cut vertices. Similarly in an undirected graph G, a *bridge* is an edge that when removed (without removing the vertices) from a graph creates more components than previously in the graph. A (connected) graph with at least two vertices is 2-*edge-connected* if and only if it has no bridge. A 2-*edge connected component* is a maximal 2-edge connected subgraph. A graph has a *degeneracy* d if every induced subgraph of the graph has a vertex with degree at most d. An ordering $v_1, v_2, \ldots, v_n$ of the vertices in such a graph is a *degenerate order* if for any i, the i-th vertex has degree at most d among vertices $v_{i+1}, v_{i+2}, \ldots, v_n$. A *topological sort* or *topological ordering* of a directed acyclic graph is a linear ordering of its vertices such that for every directed edge $(u, v) \in E$ from vertex u to vertex v, u comes before v in the ordering. A *minimum spanning tree (MST)* or *minimum weight spanning tree* is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.

Given a biconnected graph G, and two distinguished vertices s and t in V such that $s \neq t$, st-numbering is a numbering of the vertices of the graph so that s gets the smallest number, t gets the largest and every other vertex is adjacent both to a lower-numbered and to a higher-numbered vertex i.e., a numbering $s = v_1, v_2, \cdots, v_n = t$ of the vertices of G is called an st-numbering, if for all vertices $v_j, 1 < j < n$, there exist $1 \leq i < j < k \leq n$

such that $\{v_i, v_j\}, \{v_j, v_k\} \in E$. It is well-known that G is biconnected if and only if, for every edge $\{s, t\} \in E$, it has an st-numbering. In the *k-partitioning problem*, we are given vertices $a_1, \cdots, a_k$ of an undirected graph G and natural numbers $c_1, \cdots, c_k$ with $c_1 + \cdots + c_k = n$, and we want to find a partition of V into sets $V_1, \cdots, V_k$ with $a_i \in V_i$ and $|V_i| = c_i$ for every i such that every set V_i induces a connected graph in G. Given a graph G, we call a set of k rooted spanning trees *independent* if they all have the same root vertex r and, for every vertex $v \neq r$, the paths from v to r in all the k spanning trees are vertex-disjoint (except for their endpoints).

A directed graph G is said to be *strongly connected* if for every pair of vertices u and v in V, both u and v are reachable from each other. If G is not strongly connected, it is possible to decompose G into its strongly connected components i.e., a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, both u and v are reachable from each other. Let T be a depth-first search tree of a connected undirected (or directed) graph G. For each vertex v of T, *preorder number* of v is the number of vertices visited up to and including v during a preorder traversal of T. Similarly, *postorder number* of v is the number of vertices visited up to and including v during a postorder traversal of T. A *chordal* graph is one in which all cycles of four or more vertices have a *chord*, which is an edge that is not part of the cycle but connects two vertices of the cycle. The chordal graphs may also be characterized as the graphs that have *perfect elimination orderings* (PEO). A perfect elimination ordering in G is an ordering of the vertices such that, for each vertex v, v and the neighbors of v that occur after v in the order form a clique.

2.2 Input graph representations

There are two standard ways to represent a graph $G = (V, E)$: as a collection of adjacency lists or as an adjacency matrix. Either way applies to both directed and undirected graphs. Because the adjacency list representation provides a compact way to represent *sparse* graphs – those for which $|E|$ is much less than $|V|^2$ – it is usually the method of choice. Adjacency matrix representation is preferred, however, when the input graph is *dense* – $|E|$ is close to $|V|^2$ or when we need to quickly figure out if there is an edge connecting two given vertices.

The *adjacency list representation* of a graph $G = (V, E)$ consists of an array *Adj* of length $|V|$, where *Adj[i]* stores a pointer to the adjacency list of vertex i, i.e., a list containing all the neighbors of vertex i with *Adj[i]* pointing to the head of the list. If G is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$ whereas for the undirected graphs, the sum of the lengths of all the adjacency lists is $2|E|$, since every edge appears twice in this representation. It's easy to augment the lists with the weights of the edges so that we can represent weighted graphs as well. The *adjacency matrix representation* of a graph $G = (V, E)$ consists of a $n \times n$ matrix $A = (a_{ij})$ such that $a_{ij} = 1$ if $(i, j) \in E$ and 0 otherwise. Thus, the adjacency matrix of a graph requires $\Theta(n^2)$ memory, independent of the number of edges in the graph. By storing the weights of the edges as the entries of the matrix, it's easy to extend this representation for weighted graphs.

While designing space efficient algorithms in read-only memory model, the specific details of the input graph representation are of great significance as we can neither modify the input nor copy the whole input in workspace. Thus space-efficient algorithms [41, 69, 94, 99] assume slightly more powerful form of input representation than what is typically assumed in classical settings.

In some of our algorithms (more specifically, in Chapter 3), we assume that the input graph G is represented using the standard *adjacency list along with cross pointers*, i.e., for undirected graphs given a vertex u and the position in its list of a neighbor v of u, there is a pointer to the position of u in the list of v. In the case of directed graphs, for every vertex u, we have a list of out-neighbors of u and a list of in-neighbours of u. And, finally we augment these two lists for every vertex with cross pointers, i.e., for each $(u, v) \in E$, given u and the position of v in out-neighbors of u, there is a pointer to the position of u in in-neighbors of v. This form of input graph representation was introduced recently in [69] and used subsequently in [14, 94, 99] to design various other space efficient graph algorithms. We also note that some of our algorithms in this chapter will work even with less powerful and the more traditional *adjacency list* representation. We specify these details regarding the exact form of input graph representation at the respective sections while describing our algorithms.

Algorithms of Chapter 4 assume that the input graphs $G = (V, E)$ are represented using *adjacency array*, i.e., G is represented by an array of length $|V|$ where the i-th entry stores a pointer to an array that stores all the neighbors of the i-th vertex. For the directed graphs, we assume that the input representation has both in/out adjacency array for all the vertices i.e., for directed graphs, every vertex v has access to two arrays, one array is for all the in-neighbors of v and the other array is for all the out-neighbors of v. This representation which has now become somewhat standard was also used in [14, 41, 69, 94, 99] recently to design various other space efficient graph algorithms.

Algorithms of Chapter 5 deal with trees mostly, and for this we work with the standard left-most child, right-sibling based representation for trees [51] (this is slightly weaker in contrast to the doubly connected edge list representation used in [12, 23]). I.e., we assume that the tree (of the input or of the tree-decomposition) is given in a representation where the children of a node are organized in a linked list. I.e. given a node label, we can find its left-most child and its right sibling in constant time. In particular, we do not have parent pointers associated with the nodes, unless stated otherwise. For the weighted versions of the algorithms and for graphs of bounded treewidth, we assume that weights or the bag sets of the vertices are given in a separate array indexed by the labels of the vertices of the tree (which we assume are in $\{1, 2, \ldots n\}$). For bounded treewidth graphs, apart from the tree decomposition, we also need the graph in read-only memory, represented in a way that adjacency can be checked in constant time.

All the graph representations we have seen so far are, roughly speaking, slight variations of the classical *adjacency list* representation, and these are used while designing space efficient graph algorithms for ROM. In Chapter 6, we work with the *in-place* model and it requires different input representations. We choose to defer the discussion regarding the details of this representation to the specific chapter for better readability.

Chapter 3

Space Efficient Linear Time Algorithms for BFS, DFS and Applications

3.1 Introduction

Since the early days of designing graph algorithms, researchers have developed several approaches for testing whether a given undirected (or directed) graph G is (strongly connected) biconnected and/or 2-edge connected, and for finding cut vertices and/or bridges of G. All of these methods use *depth-first search* (DFS) as the backbone to design the main algorithm. The classical linear time algorithms due to Tarjan [128, 129] compute the so-called "low-point" values (which are defined in terms of a DFS-tree of G) for every vertex v, and checks some conditions using that to determine whether G has the desired property. There are other linear time algorithms as well for these problems (see [124] and all the references therein). Similary, *breadth-first search* (BFS) has been used to find shortest path in unweighted graphs and testing if a given graph is bipartite. All of these classical algorithms take $O(m + n)$ time and $O(n)$ words (our model of computation is the *register input model*) of space. In this chapter, we focus on improving the space bounds of these fundamental and basic graph algorithms while keeping their running time intact as in the classical implementations. For this we first design space efficient algorithms for BFS and DFS, and later complement these algorithms with other ideas to solve several applications of these two most fundamental and well known graph search methods space efficiently. Throughout this chapter, we assume that the input graph G is represented using the standard *adjacency list along with cross pointers*.

3.1.1 Our results and organization of this chapter

Asano et al. [9] show that DFS of a directed or undirected graph G on n vertices and m edges can be performed using $n + o(n)$ bits and (an unspecified) polynomial time. Using $2n + o(n)$ bits, they bring down the running time to $O(mn)$ time, and using a larger $O(n)$ bits, the running time of their algorithm is $O(m \lg n)$. In a similar vein,

- we show in Section 3.3 that the vertices of a directed or undirected graph can be listed in BFS order using $n \lg 3 + o(n)$ bits and $O(mf(n) \lg n)$ time where $f(n)$ is any (extremely slow-growing) function of n for example $\lg^* n$ (the o term in the space is a function of $f(n)$), while the running time can be brought down to the optimal $O(m + n)$ time using $2n + o(n)$ bits.

  En route to this algorithm, we develop in Section 3.2,

- a data structure that maintains a set of elements from a universe of size n, say $[1..n]$, using $n + o(n)$ bits to support, apart from the standard insert, search and delete operations, the operation *findany* of finding an arbitrary element of the set, and returning its value all in constant time. It can also output all elements of the set in no particular order in $O(k + 1)$ time where k is the number of elements currently belonging to the set.

In what follows, in Section 3.3.2, we improve the space for BFS further at the cost of slightly increased runtime. Next, we digress slightly from our theme of linear time algorithms and provide a time/space tradeoff like BFS for the minimum spanning tree problem in Section 3.4. In particular, we provide an implementation to find a minimum weight spanning tree in a weighted undirected graph (with weights bounded by polynomial in n) using $n + O(n/f(n))$ bits and $O(m \lg n f(n))$ time, for any function $f(n)$ such that $1 \leq f(n) \leq n$.

- For DFS, Asano et al. [9] showed that DFS in a directed or undirected graph can be performed in $O(m \lg n)$ time and $O(n)$ bits of space, and Elmasry et al. [69] improved the time to $O(m \lg \lg n)$ time still using $O(n)$ bits of space. We show the following:
  - In Section 3.5, we first show that, we can perform DFS in a directed or undirected graph in linear time using $O(m + n)$ bits. This, for example, improves the runtime of the earlier known results for sparse graphs (where m is $O(n)$) while still using the same asymptotic space. Building on top of this DFS algorithm and other observations, we show how to efficiently compute the *chain decomposition* of a connected undirected graph. This lets us perform a variety of applications of DFS (including testing biconnectivity and 2-edge connectivity, finding cut vertices and edges among others) within the same time and space bound. Our algorithms for these applications improve the space requirement (for sparse graphs) of all the previous algorithms from $\Theta(n \lg n)$ bits to $O(m + n)$ bits, preserving the same linear runtime.
  - in Section 3.6, for all the problems mentioned above and dealt in Section 3.5, we improve the space even further to $O(n \lg(m/n))$ bits keeping the same $O(m+n)$ running time. The space used by these algorithms, for some ranges of m (say $\Theta(n(\lg \lg n)^c$ for some constant c), is even better than that of the recent work by Kammer et al. [99], that computes cut vertices using $O(n + \min\{m, n \lg \lg n\})$

bits.

- In Section 3.7, we give an implementation of computing a topological sort of a directed acyclic graph in $O(m+n)$ time and $O(n\lg(m/n))$ bits of space. We can even detect if the graph is not acyclic within the same time and space bounds. This implementation contrasts with an earlier bound of $O(m+n)$ time and $O(n\lg\lg n)$ bits of space [69], and is more space efficient for sparse directed graphs (that includes those directed graphs whose underlying undirected graph is planar or has bounded treewidth or degeneracy).

  A graph has a degeneracy d if every induced subgraph of the graph has a vertex with degree at most d (for example, planar graphs have degeneracy 5, and trees have degeneracy 1). An ordering $v_1, v_2, \dots v_n$ of the vertices in such a graph is a degenerate order if for any i, the i-th vertex has degree at most d among vertices $v_{i+1}, v_{i+2}, \dots v_n$. There are algorithms [20, 71] that can find the degeneracy order in $O(m+n)$ time using $O(n)$ words. We show that, given a d, we can output the vertices of a d-degenerate graph in $O(m+n)$ time using $O(n\lg(m/n))$ bits of space in the degeneracy order. We can even detect if the graph is d-degenerate in the process. As m is $O(nd)$, we have an $O(nd)$-bit algorithm which is more space efficient if d is $o(\lg n)$ (this is the case, for example, in planar graphs or trees).

- In Section 3.8, we show an even improved algorithm for these problems using some very recent reults of [99] and some new observations. These algorithms fare well for dense graphs than the previously described results. Finally we conclude with an unified linear time and space efficient result concerning DFS and many of its applications.

3.1.2 Related work

Regarding the data structure we develop to support the *findany* operation, Elmasry et al. [Lemma 2.1, [69]] state a data structure (without proof) that supports all the operations i.e., insert, search, delete and findany (they call it *some_id*) among others, in constant time. But their data structure takes $O(n)$ bits of space where the constant in the O term is not explicitly stated. Since the publication of our result, Hagerup and Kammer [94] have independently reported a structure with $n + o(n)$. Though their lower order (the little "oh") term $O(n/\lg n)$ is better than ours which is $O(n \lg\lg n/\lg n)$, we believe that our structure is a lot simpler and supports a smaller set of operations, yet sufficient for the space efficient BFS implementation. Furthermore, we do provide a few other applications of our structure also. Recently, Poyias et al. [117] considered the problem of compactly representing a rewritable array of bit-strings, and to achieve that they used our findany structure in their algorithms.

Brodal et al. [32] considered a version of the *findany* operation where the goal was to find any element of the set and return its rank (the number of elements smaller than that). For that they gave a non-constant lower bound, though they don't assume that the elements are from a bounded universe. They give a randomized data structure that takes a constant amortized time per operation. However their main objective was to provide time tradeoffs between operations supported by the data structure and they didn't worry about space considerations. We note that this operation and their setup is different from the *findany* query we support.

3.1.3 Preliminaries

Representing a Vector: We will use the following theorem from [60]:

Theorem 3.1. *[60] On a Word RAM, one can represent a vector* $A\ [1..n]$ *of elements* *from a finite alphabet Σ using $n \lg |\Sigma| + O(\lg^2 n)$ bits*[1]*, such that any element of the vector can be read or written in constant time.*

Rank-Select: We also make use of the following well-known theorem.

Theorem 3.2. *[49, 92, 109] We can store a bitstring O of length n with additional $o(n)$ bits such that rank and select operations (defined below) can be supported in $O(1)$ time. Such a structure can also be constructed from the given bitstring in $O(n)$ time.*

Here the rank and select operations are defined as following:

- $rank_a(O, i)$ = number of occurrences of $a \in \{0, 1\}$ in $O[1, i]$, for $1 \leq i \leq n$;
- $select_a(O, i)$ = position in O of the i-th occurrence of $a \in \{0, 1\}$.

3.2 Maintaining dictionaries under findany operation

We consider the data structure problem of maintaining a set S of elements from $\{1, 2, \ldots n\}$ to support the following operations in constant time.

- *insert (i)*: Insert element i into the set.
- *search (i)*: Determine whether the element i is in the set.
- *delete (i)*: Delete the element i from the set if it exists in the set.
- *findany*: Find any element from the set and return its value. If the set is empty, return a NIL value.

Note that there exist several solutions for this problem in the data structure literature already even though their main focus is different than ours. For example it is trivial

[1]The data structure requires $O(\lg n)$ precomputed word constants, thus the second order $O(\lg^2 n)$ bits.

to support the first three operations in constant time using n bits of a characteristic bit vector (CBV) where the i-th bit of the vector is set to 1 if $i \in S$ and is set 0 otherwise. We could support the *findany* operation by keeping track of one of the elements, but once that element is deleted, we need to find another element to answer a subsequent *findany* query. This might take $O(n)$ time in the worst case. But this is easy to support in constant time if we have the elements stored in a linked list which takes $O(n \lg n)$ bits. One could also use the dynamic rank-select (DRS) structure where insert, delete and findany operations take $O(\lg n/ \lg\lg n)$ time, and search takes $O(1)$ time [96, 118]. Another approach would be to use the classical balanced binary search tree (BBST) to support these operations where the dictionary operations take $O(\lg n)$ time and findany takes $O(1)$ time. This can be slightly improved by using the van Emde Boas tree (vEB) [133]. One can further improve the results for balanced binary search trees and van Emde Boas trees by using the 'dynamic range report' structure (DRR) of Mortensen et al. [108], though it still lacks the time and space bound we want here. Our main result in this section is that the *findany* operation, along with the other three, can be supported in constant time using $o(n)$ additional bits.

3.2.1 Findany dictionary

Theorem 3.3. *A set of elements from a universe of size n can be maintained using $n + o(n)$ bits to support insert, delete, search and findany operations in constant time. We can also enumerate all elements of the set (in no particular order) in $O(k + 1)$ time where k is the number of elements in the set. The data structure can be initialized in $O(1)$ time.*

Proof. Let S be the characteristic bit vector of the set having n bits. We follow a two level blocking structure of S, as in the case of succinct structures supporting rank and select [49, 109]. However, as S is 'dynamic' (in that bit values can change due to insert and delete), we need more auxiliary information. In the discussion below, sometimes we omit floors and ceilings to keep the discussion simple, but they should be clear from the context.

We divide the bit vector S into $n/\lg^2 n$ blocks of consecutive $\lg^2 n$ bits each, and divide each such block into up to $2\lg n$ sub-blocks of size $\lceil(\lg n)/2\rceil$ bits each. We call a block (or sub-block) non-empty if it contains at least a 1. We maintain the non-empty blocks, and the non-empty sub-blocks within each block in linked lists (not necessarily in order). Within a sub-block, we find the first 1 or the next 1 by a table look up. We provide the specific details below.

First, we maintain an array *number* indicating the number of 1s in each block, i.e., $number[i]$ gives the number of 1s in the i-th block of S. It takes $O(n \lg\lg n/\lg^2 n)$ bits as each block can have at most $\lg^2 n$ elements of the given set. Then we maintain a queue (say implemented in a space efficient resizable array [33]) *block-queue* having the block numbers that have a 1 bit, and new block numbers are added to the list as and when new blocks get 1. It can have at most $n/\lg^2 n$ elements and so has $O(n/\lg^2 n)$ indices taking totally $O(n/\lg n)$ bits. In addition, every element in *block-queue* has a pointer to another queue of sub-block numbers of that block that have an element of S. Each such queue has at most $2\lg n$ elements each of size at most $\lg\lg n$ bits each (for the sub-block index). Thus the queue *block-queue* along with the queues of sub-block indices takes $O(n\lg\lg n/\lg n)$ bits. We also maintain an array, *block-array*, of size $n/\lg^2 n$ where *block-array*$[i]$ points to the position of block i in *block-queue* if it exists, and is a NIL pointer otherwise and array, *sub-block-array*, of size $2n/\lg n$ where *sub-block-array*$[i]$ points to the position of the subblock i in its block's queue if its block was present in *block-queue*, and is a NIL pointer otherwise. So, *block-array* takes $n/\lg n$ bits and *sub-block-array* takes $2n\lg\lg n/\lg n$ bits.

We also maintain a global table T precomputed that stores for every bitstring of size $\lceil(\lg n)/2\rceil$, and a position i, the position of the first 1 bit after the i-th position. If there is no ‘next 1’, then the answer stored is -1 indicating a NIL value. The table takes $O(\sqrt{n}(\lg\lg n)^2)$ bits. This concludes the description of the data structure that takes $n + O(n\lg\lg n/\lg n)$ bits. See Figure 2.1 for an illustration.

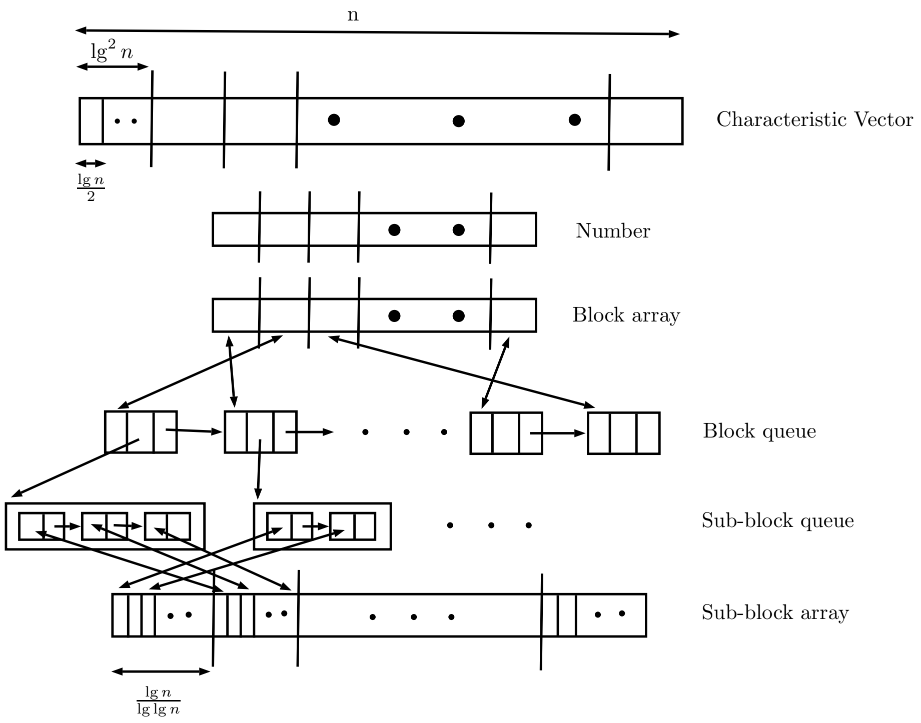

Figure 3.1: An illustration of the inner working details of our findany data structure. The precomputed table is not shown in the diagram.

Now we explain how to support each of the required operations. Membership is the easiest: just look at the i-th bit of S and answer accordingly. In what follows, when we say the ‘corresponding bit or pointer’, we mean the bit or the pointer corresponding to the block or the sub-block corresponding to an element, which can be determined in constant time from the index of the element. To insert an element i, first determine from the table T, whether there is a 1 in the corresponding sub-block (before the element is inserted), set the i-th bit of S to 1, and increment the corresponding value in *number*. If the corresponding pointer of *block-array* was NIL, then insert the block index to *block-queue* at the end of the queue, and add the sub-block corresponding to the i-th bit into the queue corresponding to the index of the block in *block-queue*, and update the corresponding pointers of *block-array* and *sub-block-array.* If the corresponding bit of *block-array* was not NIL (the big block already had an element), and if the sub-block did not have an element before (as determined using T), then find the position of the block index in *block-queue* from *block-array*, and insert the sub-block index into the queue of that block at the end of the queue. Update the corresponding pointer of *sub-block-array.*

To support the delete operation, set the i-th bit of S to 0 (if it was already 0, then there is nothing more to do) and decrement the corresponding number in *number.* Determine from the table T if the sub-block of i has a 1 (after the i-th bit has been set to 0). If not, then find the index of the sub-block from the arrays *block-array* and *sub-block-array* and delete that index from the block's queue from *block-queue.* If the corresponding number in *number* remains more than 0, then there is nothing more to do. If the number becomes 0, then find the corresponding block index in *block-queue* from the array *block-array*, and delete that block (along with its queue that will have only one sub-block) from *block-queue.* Update the pointers in *block-array* and *sub-block-array* respectively. As we don't maintain any order in the queues in *block-queue*, if we delete an intermediate element from the queue, we can always replace that element by the last element in the queue updating the pointers appropriately.

To support the findany operation, we go to the tail of the queue *block-queue*, if it is NIL, we report that there is no element in the set, and return the NIL value. Otherwise, go to the block at the tail of *block-queue*, and get the first (non-empty) sub-block number from the queue, and find the first element in the sub-block from the table T, and return the index of the element.

To enumerate the elements of the set, we traverse the list *block-queue* and the queues of each element of *block-queue*, and for each sub-block in the queues, we find the next 1 in constant time using the table T and output the index.

To enable initialization in $O(1)$ time, with each entry of the block (sub-block) queue, we also store the block index corresponding to that entry – analagous to the "on-the-fly array initialization" technique of [Exercise 2.12 of [3], Section III.8.1 of [107], [83]], for example. The (sub) block index stored with the entries in the (sub) block queue act as the "back pointers" to the (sub) block-array. Also, instead of storing the precomputed tables to compute the 'first 1 bit after the i-th position' operation, we can support the operation using $O(1)$ word operations [67]. $\square$

3.2.2 Extension

We generalize the data structure of the last section to maintain a collection of more than one disjoint subsets of the given universe to support the insert, delete, membership and findany operations. In this case, insert, delete and findany operations should come with a set index (to be searched, inserted or deleted).

Theorem 3.4. *A collection of c disjoint sets that partition the universe of size n can be maintained using $n \lg c + o(n)$ bits to support insert, delete, search and findany operations in constant time. We can also enumerate all elements of any given set (in no particular order) in $O(k+1)$ time where k is the number of elements in the set.*

Proof. The higher order term is for representing the (generalized) characteristic vector S where $S[i]$ is set to the number (index) of the set where the element is present. From Theorem 3.1, S can be represented using $n \lg c + o(n)$ bits so that the i-th value can be retrieved or set in constant time. The rest of the data structures and the algorithms are as in the proof of Theorem 3.3 (hence the extra $o(n)$ bits), we have a copy of such structures for each of the c sets. $\square$

3.3 Breadth First Search (BFS)

3.3.1 Using $2n + o(n)$ bits

We explain how breadth first search (BFS) can be performed in a space efficient manner using the data structure of Theorem 3.4. Our goal is to output the vertices of the graph in the BFS order. We start as in the textbook BFS by coloring all vertices white. The algorithm grows the search starting at a vertex s, making it grey and adding it to a queue. Then the algorithm repeatedly removes the first element of the queue, and adds all its white neighbors at the end of the queue (coloring them grey), coloring the element black after removing it from the queue. As the queue can store up to $O(n)$ elements, the space for the queue can be $O(n \lg n)$ bits. To reduce the space to $O(n)$ bits, we crucially observe the following two properties of BFS:

- Elements in the queue are only from two consecutive levels of the BFS tree.
- Elements belonging to the same level can be processed in any order, but elements of the lower level must be processed before processing elements of the higher level.

The algorithm maintains four colors: white, grey1, grey2 and black, and represents the vertices with each of these colors as sets W, S_1, S_2 and B respectively using the data structure of Theorem 3.4. It starts with initializing S_1 (grey1) to s, S_2 and B as empty sets and W to contain all other vertices. Then it processes the elements in each set S_1 and S_2 switching between the two until both sets are empty. As we process an element from S_i, we add its white neighbor to $S_{i+1 \ mod 2}$ and delete it from S_i and add it to B. When S_1 and S_2 become empty, we scan the W array to find the next white vertex and start a fresh BFS again from that vertex. As insert, delete, membership and findany operations take constant time, and we are maintaining four sets, we have from Theorem 3.4,

Theorem 3.5. *Given a directed or undirected graph G, its vertices can be output in a BFS order starting at a vertex using $2n + o(n)$ bits in $O(m + n)$ time.*

Note that, we don't need to build the findany structure on top of B and W i.e., they can be implemented as plain bitmaps. The findany structures are only required on sets S_1 and S_2 respectively to efficiently find grey vertices.

3.3.2 Using $n \lg 3 + o(n)$ bits

Here we slightly deviate from the theme of the chapter and show that we can improve the space further for performing BFS if we are willing to settle for more than a linear amount of time. We provide the algorithms next. We will have three colors, one for the white unexplored vertices and two colors for those explored including those currently being explored. The two colors indicate the parity of the level (the distance from the starting vertex) of the explored vertices. Thus the starting vertex s is colored 0 to mark that its distance from s is of even length and every other vertex is colored 2 to mark them as unexplored (or white). We simply have these values stored in the representation of Theorem 3.1 using $n \lg 3 + O(\lg^2 n)$ bits and we call this as the color array. The algorithm repeatedly scans this array and in the i-th scan, it changes all the 2 neighbors of i *mod* 2 to $i + 1$ *mod* 2. i.e., in one scan of the array, the algorithm changes all the 2-neighbors of all the 0 vertices to 1, and in the next, it changes all the 2-neighbors of all of the 1 vertices to 0. The exploration (of the connected component) stops when in two consecutive scans of the list, no 2 neighbor is found. Note that for those vertices which would have been colored black in the normal BFS, none of its neighbors will be marked 2, and so the algorithm will automatically figure them as black. The running time of $O(mn)$ follows because each scan of the list takes $O(m)$ time (to go over neighbors of vertices with one color, some of which could be black) and at most $n + 2$ scans of the list are performed as in each scan (except the last two), the color of at least one vertex marked 2 is changed.

Thus we have the following theorem,

Theorem 3.6. *Given a directed or undirected graph, its vertices can be output in a BFS order starting at a vertex using* $n \lg 3 + O(\lg^2 n)$ *bits and in* $O(mn)$ *time.*

The $O(m)$ time for each scan of the previous algorithm is because while looking for vertices labelled 0 that are supposed to be 'grey', we might cross over spurious vertices labelled 0 that are 'black' (in the normal BFS coloring). To improve the runtime further, we maintain two queues Q_0 and Q_1 each storing up to $n/\lg^2 n$ values to find the grey 0 and grey 1 vertices quickly, in addition to the color array that stores the values 0, 1 or 2. We also store two boolean variables, *overflow-Q0, overflow-Q1*, initialized to 0 and to be set to 1 when more elements are to be added to these queues (but they don't have room). Now the algorithm proceeds in a similar fashion as the previous algorithm except that, along with marking corresponding vertices 0 or 1 in the color array, we also insert them into the appropriate queues. i.e. when we expand vertices from Q_0 (Q_1), we insert their (white) neighbors colored 2 to Q_1 (Q_0 respectively) apart from setting their color entries to 1 (0 respectively). Due to the space restriction of these queues, it is not always possible to accomodate all the vertices of some level during the execution of BFS. So, when we run out of space in any of these queues, we continue to make the changes (i.e. 2 to 1 or 2 to 0) in the color array directly without adding those vertices to the queue, and we also set the corresponding overflow bit.

Now instead of scanning the color array for vertices labelled 0 or 1, we traverse the appropriate queues spending time proportional to the sum of the degree of the vertices in the level. If the overflow bit in the corresponding queue is 0, then we simply move on to the next queue and continue. When the overflow bit of a queue is set to 1, then we switch to our previous algorithm and scan the array appropriately changing the colors of their white neighbors and adding them to the appropriate queue if possible. It is easy to see that this method correctly explores all the vertices of the graph. The color array uses $n \lg 3 + O(\lg^2 n)$ bits of space. And, for other structures, Q_0, Q_1 and other variables, we need at most $O(n/\lg n)$ bits. So, overall the space requirement is $n \lg 3 + o(n)$ bits. To analyse the runtime, notice that as long as the overflow bit of a queue is 0, we spend time proportional the number of neighbors of the vertices in that level, and we spend $O(m)$ time otherwise. When an overflow bit is 1, then the number of nodes in the level is at least $n/\lg^2 n$ and this can not happen for more than $\lg^2 n$ levels where we spend $O(m)$ time each. Hence, the total runtime is $O(m \lg^2 n)$.

Theorem 3.7. *Given a directed or undirected graph, its vertices can be output in a BFS order starting at a vertex using $n \lg 3 + o(n)$ bits of space and in $O(m \lg^2 n)$ time.*

Remark: We can slightly optimize the previous algorithm by observing the fact that a vertex v belongs to an overflowed level then v is expanded twice i.e., first time when the algorithm was expanding vertices from the queue and deleting them. And, secondly, as the overflow bit is set, the algorithm switches to our previous algorithm of Theorem 3.6 and scan the color array appropriately changing the colors of their white neighbors and adding them to the appropriate queue if possible. We can avoid this double expansion by checking the overflow bit first and if this bit is set, instead of taking vertices out of the corresponding queue and expanding, the algorithm can directly start working with the color array. We can still correctly retrieve all the vertices by checking the same condition. This ensures that all the vertices which belong to an overflowed level won't be expanded twice.

Note that by making the sizes of the two queues to $O(n/(f(n) \lg n))$ for any (slow growing) function $f(n)$, the space required for the queues will be $O(n/f(n))$ bits and the running time will be $O(mf(n) \lg n)$.

Theorem 3.8. *Given a directed or undirected graph, its vertices can be output in a BFS order starting at a vertex using $n \lg 3 + O(n/f(n))$ bits and in $O(mf(n) \lg n)$ time where $f(n)$ is any function of n such that $1 \leq f(n) \leq n$.*

Since the appearance of our result in [14], Hagerup et al. [94] presented an implementation of BFS taking $n \lg 3 + O(n/\lg n)$ bits of space and the optimal $O(m+n)$ time, thus improving the result of our Theorem 3.8. But, note that, in our algorithm of Theorem 3.6, we improved the space (in the second order) even further than what is needed in their algorithm [94] albeit with degradation in time. We do not know whether we can reduce the space further (to possibly $n + o(n)$ bits) while still maintaining the runtime to $O(m \lg^c n)$ for some constant c or even $O(mn)$. We leave this as an open problem. However, in the next section we provide such an algorithm for the Minimum Spanning Tree (MST) problem.

3.4 Minimum Spanning Tree (MST)

In this section, we give a space efficient implementation of Prim's algorithm [51] to find a minimum spanning tree. Here we are given a weight function $w : E \to Z$. We also assume that the weights of non-edges are ∞ and that the weights can be represented using $O(\lg n)$ bits. In particular, we show the following,

Theorem 3.9. *A minimum spanning forest of a given undirected weighted graph, where the weights of any edge can be represented in $O(\lg n)$ bits, can be output using $n + O(n/f(n))$ bits and in $O(m \lg n f(n))$ time, for any function $f(n)$ such that $1 \leq f(n) \leq n$.*

Proof. Our algorithm is inspired by the MST algorithm of [69], but we work out the constants carefully. Prim's algorithm starts with initializing a set S with a vertex s. For every vertex v not in S, it finds and maintains $d[v] = \min\{w(v,x) : x \in S\}$ and $\pi[v] = x$ where $w(v,x)$ is the minimum among $\{w(v,y) : y \in S\}$. Then, it repeatedly deletes the vertex with the smallest d value from $V \setminus S$ adding it to S. Then, the d values are updated by looking at the neighbors of the newly added vertex.

The space for d values can take up to $O(n \lg n)$ bits. To reduce the space to $O(n)$ bits, we find and keep, in $O(n)$ time, the set M of the smallest $n/(f(n)\lg n)$ values among the d values of the elements of $V \setminus S$ in a binary heap. This takes $O(n/f(n))$ bits and $O(n)$ time. We maintain the set S in a bit vector taking n bits. We maintain the indices of M in a balanced binary search tree, and each node (index v) has a pointer to its position in the heap of the d values, and also stores the index $\pi[v]$. Thus we can think of M as consisting of triples $(v, d[v], \pi[v])$ where $d[v]$ is actually a pointer to $d[v]$ in the heap. The storage for M takes $O(n/f(n))$ bits. We also find and store the max value of M in a variable Max that also has the vertex label that achieves the maximum. Now we execute Prim's algorithm by repeatedly deleting elements only from M and updating (decreasing) values in M until M becomes empty. In particular, while updating the values we check if the new value is larger than the variable Max. In such cases, we don't do anything. Otherwise, we insert the new value in M and delete the current vertex realizing the maximum value and we proceed further till M becomes empty. Then (for $f(n)\lg n$) times) we find the next smallest $n/(\lg n f(n))$ values from $V \setminus M \setminus S$ and continue the process.

Finding the d values of every element in $L = V \setminus S \setminus M$ requires an overall $O(m)$ time (for finding the minimum among all edges incident with vertices in S), and finding the smallest $n/f(n)\lg n$ values among them take $O(n)$ time. These steps are repeated $O(f(n)\lg n)$ times resulting in the overall runtime of $O(mf(n)\lg n)$.

In the heap, $n-1$ deletemins and up to m decrease key operations are executed which take $O((m+n)\lg n)$ time by using a binary heap. Note that more sophisticated (for example, Fibonacci heap) implementations are unnecessary as the other operations dominate the running time. □

Note that, for all the algorithms discussed in this section, we can assume that the input graph is represented as the standard *adjacency list* [51], instead of the more powerful adjacency array with cross pointers representation.

3.5 DFS and its applications using $O(m+n)$ bits

The classical and standard implementation of DFS using a stack and color array takes $O(m+n)$ time and $O(n \lg n)$ bits of space. Improving on this, recently Elmasry et al. [69] showed the following,

Theorem 3.10. *A DFS traversal of a directed or undirected graph G with n vertices and m edges can be performed using $O(n \lg\lg n)$ bits of space and $O(m+n)$ time.*

In this section, we start by improving upon the results of Theorem 3.10 of Elmasry et al. [69] and Theorem 4 of Asano et al. [9] by showing an $O(n)$-bit DFS traversal method for sparse graphs that runs in linear time. Using this DFS as backbone, we provide a space efficient implementation for computing several other useful properties of an undirected graph.

3.5.1 DFS

In what follows we describe how to perform DFS in $O(n+m)$ time using $O(n+m)$ bits of space. Note that, this is better (in terms of time) than both the previous solutions for sparse graphs (when $m = o(n \lg\lg n)$) with same space bounds. The class of sparse graphs includes a large class of graphs including planar graphs, bounded genus, bounded treewidth, bounded degree graphs, and H-minor-free graphs. These are also the majority of the graph classes which arise in practice, thus we believe that an implementation of our algorithm would be very useful.

Recall that, our input graphs $G = (V, E)$ are represented using the standard adjacency array along with cross pointers. We describe our algorithm for directed graphs, and mention the changes required for undirected graphs. Central to our algorithm is an encoding of the out-degrees of the vertices in unary. Let $V = \{1, 2, \cdots, n\}$ be the vertex set. The unary degree sequence encoding O of the directed graph G has n 0s to represent the n vertices and each 0 is followed by a number of 1s equal to the out-degree of that vertex. Moreover, if d is the degree of vertex v_i, then d 1s following the i-th 0 in the O array corresponds to d out-neighbors of v_i (or equivalently the edges from v_i to the d out-neighbors of v_i) in the same order as in the out-adjacency array of v_i. Clearly O uses $n + m$ bits and can be obtained from the out-neighbors of each vertex in $O(m + n)$ time. We use another bit string E of the same length where every bit is initialized to 0. The array E will be used to mark the tree edges of the DFS as we build the DFS tree, and will be used to backtrack when the DFS has finished exploring a vertex. The bits in E are in one-to-one correspondence with bits in O. If (v_i, v_j) is an edge in the DFS tree where v_i is the parent of v_j, and suppose k is the index of the edge (v_i, v_j) in O, then the corresponding location in the E array is marked as 1 during DFS. Thus once DFS finishes traversing the whole graph, the number of ones in the E array is exactly the number of tree edges. We also store another array, say C, having entries from {*white, gray, black*} with the usual meaning i.e., each vertex v remains white until it is visited, is colored gray when DFS visits v for the first time, and is colored black when its out-adjacency array has been checked completely. We can represent C using Theorem 3.1 in $n \lg 3 + o(n)$ bits so that individual entries can be accessed or modified in constant time. The bitvector O is represented using the static rank-select data structure of Theorem 3.2 that uses additional $o(m + n)$ bits. So overall we need $2m + (\lg 3 + 2)n + o(m + n)$ bits to represent the arrays O, E and C.

Suppose v_j is a child of v_i in the final DFS tree. We can think of the DFS procedure as performing the following two steps repeatedly until all the vertices are explored. First step takes place when DFS discovers a vertex v_j for the first time, and as a result v_j's color changes to gray from white. We call this phase as forward step. When DFS completes exploring v_j i.e. the subtree rooted at v_j in the DFS tree, it performs two tasks subsequently. First, it backtracks to its parent v_i, and then finds in v_i's list the next white neighbor to explore. The latter part is almost similar to the forward step described before. We call the first part alone as backtrack step. In what follows, we describe how to implement each step in detail.

We start our DFS with the starting vertex, say r, changing its color to gray in the color array C. Then, as in the usual DFS algorithm, we scan the out-adjacency list of r, and find the first white neighbor, say v, to make it gray. When the edge (r, v) is added to the DFS tree, we mark the position corresponding to the edge (r, v) in E to 1[2]. We continue the process with the new vertex making it gray until we encounter a vertex w that has no white out-neighbors. At this point, we will color the vertex w black, and we need to backtrack.

To find the vertex to backtrack, we do the following. We go to w's in-neighbor list to find a gray vertex which is its parent. For each gray vertex t in w's in-neighbor list, we follow the cross pointers to reach w in t's out-adjacency list and check its corresponding entry (t, w) in E array (using select operation to find w after t-th 0). Observe that, among all these gray in-neighbors of w, only one edge out of them to w will be marked in E as this is the edge that DFS traversed while going in the forward direction to w. So once we find an in-neighbor t such that the position corresponding to (t, w) in E is marked and t is gray, we know that w's parent is t in the DFS tree. Also the cross pointer puts us in the position of w in t's out-neighbor list, and we start from that position to find the next white vertex to explore DFS. So the only extra computation from the standard DFS we do is to spend time proportional to the degree of each black vertex (to find its parent to backtrack) and so overall there is an extra overhead of $O(m)$ time. The navigation we do to determine the tree edges are on O which is a static array, and so from Theorem 3.2, all these operations can be performed in constant time. Thus we have

Theorem 3.11. *A DFS traversal of a directed graph G can be performed in $O(n + m)$ time using $(2m + (\lg 3 + 2)n) + o(m + n)$ bits.*

For undirected graphs, first observe that the unary degree sequence encoding O takes $2m + n$ bits as each edge appeares twice. As E also takes $2m + n$ bits, overall we require

[2]Note that to implement this step, we only require the *select*$_0$ operation in Theorem 3.2.

$(4m + (\lg 3 + 2)n) + o(m + n)$ bits of space. As for DFS, observe that the forward step, as defined before, can be implemented in exactly the same manner. It is crucial to mention one subtle point that, while marking an edge (v_i, v_j), we don't mark its other entry i.e. (v_j, v_i). So when DFS finishes, *for tree edges exactly one of the two entries will be marked one in E array.* Backtracking step is now little easier as we don't have to switch between two lists. We essentially follow the same steps in the adjacency array to check for a vertex t in w's array such that t is gray and the corresponding entry for the edge (t, w) is marked in E. Once found, we start with the next white vertex. Hence,

Theorem 3.12. *A DFS traversal of an undirected graph G can be performed in $O(n+m)$ time using $(4m + (\lg 3 + 2)n) + o(m + n)$ bits.*

We can decrease the space slightly by observing that, we are not really using the third color *black*. More specifically, we can continue to keep a vertex gray even after its subtree has been explored. As we only explore white vertices always and never expand gray or black, the correctness follows immediately. Note that this observation is true in the standard DFS implementation as well. This gives us the following.

Theorem 3.13. *A DFS traversal of a directed graph G can be performed in $O(n + m)$ time using $(2m + 3n) + o(m + n)$ bits. For undirected graphs, the space required is $(4m + 3n) + o(m + n)$ bits.*

3.5.2 Applications of DFS

One of the classical applications of DFS is to determine, in a connected undirected graph, all the cut vertices and bridges which are defined as, respectively, the vertices and edges whose removal results in a disconnected graph. Since the early days of designing graph algorithms, researchers have developed several approaches to test biconnectivity and 2-edge connectivity, find cut vertices and bridges of a given undirected graph. Most of these methods use depth-first search as the backbone to design the main algorithm. For biconnectivity and 2-edge connectivity, the classical algorithm due to Tarjan [128, 129] computes the so-called "low-point" values (which are defined in terms of a DFS-tree) for every vertex v, and checks some conditions using that to determine cut vertices, bridges of G and check whether G is 2-edge connected or biconnected. Brandes [31] and Gabow [85] gave considerably simpler algorithms for testing biconnectivity by using simple path-generating rules instead of low-points; they call these algorithms path-based. All of these algorithms take $O(m + n)$ time and $O(n)$ words of space. Another algorithm due to Schmidt [124] is based on chain decomposition of graphs to determine biconnectivity and 2-edge connectivity. Implementing this algorithm takes $O(m + n)$ time and $O(m)$ words of space. In what follows, we present a space efficient implementation for Schmidt's algorithm based on the DFS algorithm we designed in the previous section. We summarize our result in the theorem below.

Theorem 3.14. *Given a connected undirected graph G, in $O(m + n)$ time and using $O(m+n)$ bits of space we can determine whether G is 2-vertex (and/or edge) connected. If not, in the same amount of time and space, we can compute all the bridges and cut vertices of the graph.*

Schmidt [123] introduced a decomposition of the input graph that partitions the edge set of the graph into cycles and paths, called chains, and used this to design an algorithm to find cut vertices and biconnected components [124] and also to test 3-connectivity [123] among others. We briefly recall Schimdt's algorithm and its main ingredient of *chain decomposition.* The algorithm first performs a depth first search on G. Let r be the root of the DFS tree T. DFS assigns an index to every vertex v i.e. the time vertex v is discovered for the first time (discovery time) during DFS. Call it depth-first-index ($DFI(v)$). Imagine that the the back edges are directed away from r and the tree edges are directed towards r. The algorithm decomposes the graph into a set of paths and cycles called chains as follows. See Figure 2.2 for an illustration.

First we mark all the vertices as unvisited. Then we visit every vertex starting at r

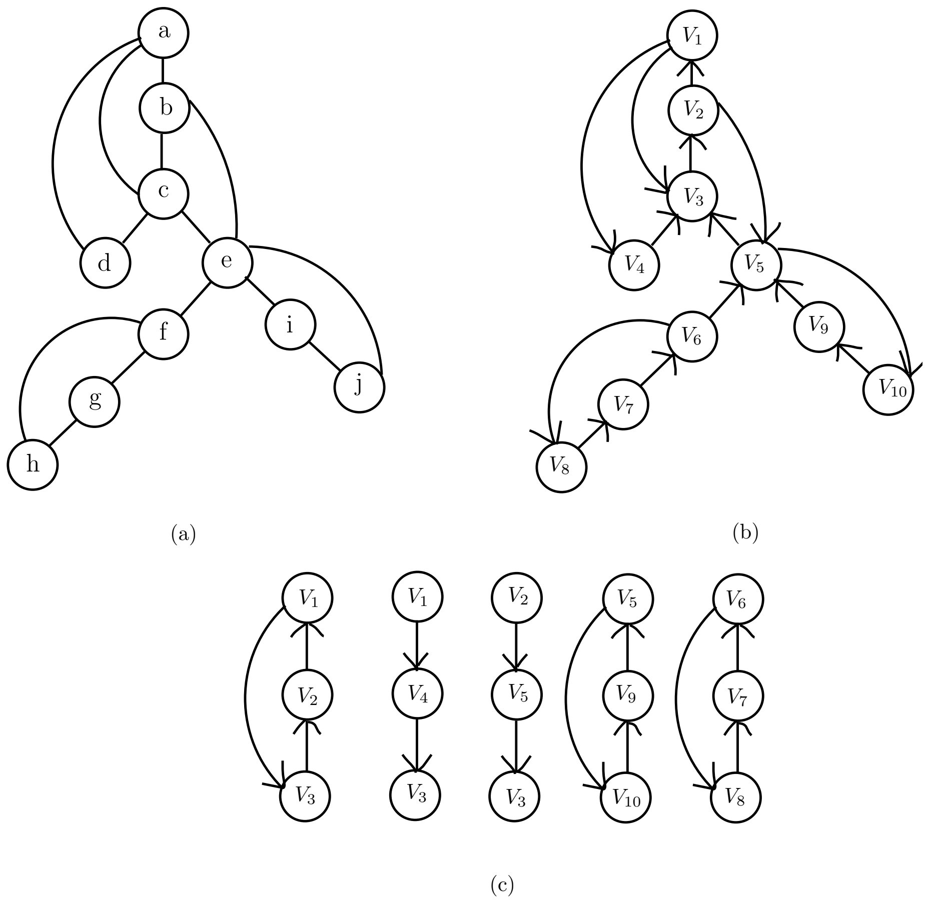

Figure 3.2: Illustration of Chain Decomposition. (a) An input graph G. (b) A DFS traversal of G and the resulting edge-orientation along with DFIs. (c) A chain decomposition D of G. The chains D_2 and D_3 are paths and rest of them are cycles. The edge (V_5, V_6) is bridge as it is not contained in any chain. V_5 and V_6 are cut vertices.

in increasing order of DFI, and do the following. For every back edge e that originates at v, we traverse a directed cycle or a path. This begins with v and the back edge e and proceeds along the tree towards the root and stops at the first visited vertex or the root. During this step, we mark every encountered vertex visited. This forms the first chain. Then we proceed with the next back edge at v, if any, or move towards the next v in increasing DFI order and continue the process. Let D be the collection of all such cycles and paths. Notice that, the cardinality of this set is exactly the same as the number of back edges in the DFS tree as each back edge contributes to one cycle or a path. Also as initially every vertex is unvisited, the first chain would be a cycle as it would end in the starting vertex. Schmidt proved the following theorem.

Theorem 3.15. *[124] Let D be a chain decomposition of a connected graph $G(V, E)$. Then G is 2-edge-connected if and if the chains in D partition E. Also, G is 2-vertex-connected if and if $\delta(G) \geq 2$ (where $\delta(G)$ denotes the minimum degree of G) and D_1 is the only cycle in the set D where D_1 is the first chain in the decomposition. An edge e in G is bridge if and if e is not contained in any chain in D. A vertex v in G is a cut vertex if and if v is the first vertex of a cycle in $D \setminus D_1$.*

The algorithm (the tests in Theorems 3.15) can be implemented easily in $O(m + n)$ time using $O(m + n)$ words as we can store the DFIs and entire chain decomposition D. To reduce the space to $O(m + n)$ bits, we first perform a depth first search of the graph G (using Theorem 3.12) and recall that at the end of the DFS procedure, we have the color array C with all colors black and the array E which encodes the DFS tree. Here for a tree edge (i, j) where i is closer to the root, the position corresponding to the edge (i, j) is marked 1 in E and that corresponding to (j, i) is marked 0, and the backedges are marked 0. To implement the chain decomposition, we do not have space to store the chains or the DFS indices. To handle the latter (DFI), we (re)run DFS and then use Schmidt's algorithm along with DFS in an interleaved way. Towards the end, we recolor all the vertices to white. To handle the former, we use two more arrays, one to mark the vertices visited in the chain decomposition, called *visited* and another array M, to mark the edges visited during the chain decomposition. The array M has size $(n + 2m)$ bits, and it has the same initial structure as E i.e. 0's separated by 1's where 0's denote edges and 1's denote vertices. The details of forming the chain decomposition and finding all cut vertices and bridges using these arrays O (original outdegree encoding), E (the DFS tree), C (color array), *visited* and M (to mark edges) are explained below.

Proof. of Theorem 3.14. We start at the root vertex r, and using the array E, find the first 'back edge' (non-tree edge) (r, x) to r. This can be found by going to the r-th 0 in O and then to the corresponding position in E that represents the vertex r (note that E has a lot more zeroes, and so we should get to the corresponding 0 of r in E by first getting to the corresponding position in O). If O has 1s after the corresponding 0, then we look for the first 0 after the corresponding position in E to find the back edge (as all the tree edges are marked 1). We mark r and x visited (if they were unvisited before) and mark both copies of the edge (r, x) (unlike what we do in the forward step of DFS) using the cross pointer in M. Now to obtain the chain, we need to follow the tree edges from x. We use the 'backtracking' procedure we used earlier for DFS. We look for an (the only) edge marked 1 in E out of the edges incident on x by scanning the adjacency list, and that gives the parent y of x (Here is where we use the fact we only mark one copy of the edge as we explore the DFS tree.).

We continue after marking y visited, and the edge (x, y) (both copies) in M until we reach r or a visited vertex when we complete the chain. Now we continue from where we left of in r's neighborhood to look for the next back edge and continue this process. Once we are done with back edges incident on r, we need to proceed to the next vertex in DFS order. As we have not stored the Depth First Indices, we essentially (re)run the DFS using the color array C. For this, we flush out the color array to make every vertex white again. Note that we don't make any changes to array E and O respectively. As this DFS procedure is deterministic, it will follow exactly the same sequence of paths like before, ultimately leading to the same DFS tree structure, and note that, this structure is already saved in array E. Conceptually, this whole process could be thought of as one step of DFS followed by multiple backtracking (for each back edge coming out of a vertex) and repeat till we visit all the vertices.

Clearly, the amount of space taken is $O(m + n)$ bits. To analyze the runtime, note that, we first perform a DFS traversal which takes linear time. At the second step, we basically perform one more round of DFS but in lazy fashion as this step comprises of one forward step followed by some backtracking. As a visited node is never explored (using the *visited* array), the overall runtime is $O(m + n)$. Edge connectivity (Theorem 3.15) can easily be checked using the array M once we have the chain decomposition. The bridges are the edges marked 0 in the array M. Cut vertices can be obtained and listed out if and and when we reach the starting vertex while forming a chain, except at the first chain (if exists). This completes the proof. $\square$

Combining all the main results from this section, we summarize our results in the following theorem below,

Theorem 3.16. *A DFS traversal of an undirected or directed graph G can be performed in $O(m + n)$ time using $O(m + n)$ bits. In the same amount of time and space, given a connected undirected graph G, we can perform a chain decomposition of G, and using that we can determine whether G is 2-vertex (and/or edge) connected. If not, in the same amount of time and space, we can compute all the bridges and cut vertices of G.*

In what follows, we show in Section 3.6 how to improve the space bounds of Theorem 3.16 keeping the same running time by applying different bookeeping technique but essentially using the same algorithm itself.

3.6 DFS and its applications using $O(n \lg(m/n))$ bits

As mentioned previously, one can easily implement the tests in Theorem 3.15 in $O(m+n)$ time using $O(m)$ words, by storing the DFIs and the entire chain decomposition, D. Theorem 3.16 shows how to perform the tests using $O(m + n)$ bits and $O(m + n)$ time. The central idea there is to maintain the DFS tree using $O(m + n)$ bits using an unary encoding of the degree sequence of the graph. And later, build on top of it another $O(m+n)$ bits structure to perform chain decompositions and the other tests of Schimdt's algorithm. We first show how the space for the DFS tree representation can be improved to $O(n \lg m/n)$ bits. Also note that, all the algorithms from the last section assume that the input graph must be respresented as adjacency array with cross pointers. Our algorithms in this section also get rid of this assumption. Here we only assume that the input graph is represented as a standard adjacency array i.e., given a vertex v and an integer k, we can access the k-th neighbor of vertex v in constant time. We remark that this input representation was also used in [41, 99] recently to design various other space efficient graph algorithms. We start by proving the following useful lemma.

Lemma 3.17. *Given the adjacency array representation of an undirected graph G on n vertices with m edges, using $O(m)$ time, one can construct an auxiliary structure of size $O(n \lg(m/n))$ bits that can store a "pointer" into an arbitrary position within the adjacency array of each vertex. Also, updating any of these pointers (within the adjacency array) takes $O(1)$ time.*

Proof. We first scan the adjacency array of each vertex and construct a bitvector B as follows: starting with an empty bitvector B, for $1 \leq i \leq n$, if d_i is the length of the adjacency array of vertex v_i (i.e., its degree), then we append the string $0^{\lceil \lg d_i \rceil - 1}1$ to B. The length of B is $\sum_{i=1}^{n} \lceil \lg d_i \rceil$, which is bounded by $O(n \lg(m/n))$. We construct auxiliary structures to support *select* queries on B in constant time, using Theorem 3.2. We now construct another bitvector P of the same size as B, which stores the required pointers into the adjacency arrays of each vertex. The pointer into the adjacency array of vertex v_i is stored using the $\lceil \lg d_i \rceil$ bits in P from position $\mathit{select}(i-1, B)+1$ to position $\mathit{select}(i, B)$, where $\mathit{select}(0, B)$ is defined to be 0. Now, using the select operations on B array and using constant time word-level read/write operations, one can easily access and/or modify these pointers in constant time. □

Given that we can maintain such pointers into the lists of every vertex, the following lemma shows that, within the same time and space bounds, we can actually maintain the DFS tree of a given graph G.

Lemma 3.18. *Given a graph G with n vertices and m edges, in the adjacency array representation in the read-only memory model, the representation of a DFS tree can be stored using $O(n \lg(m/n))$ additional bits, which can be constructed on the fly during the*

DFS algorithm.

Proof. We use the representation of Lemma 3.17 to store *parent* pointers into the adjacency array of each vertex. In particular, whenever the DFS outputs an edge (u, v), where u is the parent of v, we scan the adjacency array of v to find u and store a pointer to that position (within the adjacency array of v). The additional time for scanning the adjacency arrays adds upto $O(m)$ which would be subsumed by the running time of the DFS algorithm. $\square$

We call the representation of the DFS tree of Lemma 3.18 as the *parent pointer representation.* Now given Lemma 3.17 and 3.18, we can simulate the DFS algorithm of Theorem 3.12 to obtain an $O(n \lg(m/n))$ bits and $O(m + n)$ time DFS implementation. The proof of Theorem 3.14 then uses another $O(m + n)$ bits to construct the chain decomposition of G and to perform the tests as mentioned in Theorem 3.15. We show here how even the space for the construction of a chain decomposition and performing the tests can be improved. We summarize our results in the following theorem below:

Theorem 3.19. *A DFS traversal of an undirected or directed graph G can be performed in $O(m + n)$ time using $O(n \lg(m/n))$ bits of space. In the same amount of time and space, given a connected undirected graph G, we can perform a chain decomposition of G, and using that we can determine whether G is 2-vertex (and/or edge) connected. If not, in the same amount of time and space, we can compute and report all the bridges and cut vertices of G.*

Proof. Using Lemma 3.18 we can simulate the DFS algorithms of Theorem 3.11 and 3.12 to obtain an $O(n \lg(m/n))$ bits and $O(m+n)$ time DFS implementation. In what follows we use this DFS algorithm to perform the tests in Theorem 3.15. With the help of the parent pointer representation, we can visit every vertex, starting at the root r of the DFS tree, in increasing order of DFI, and enumerate (or traverse through) all the non-tree (back) edges of the graph as required in Schimdt's algorithm as follows: for each node v

in DFI order, and for each node u in its adjacency list, we check if u is a parent of v. If so, then (u, v) is a tree edge, else it is a back edge. We maintain a bit vector *visited* of size n, corresponding to the n vertices, initialized to all zeros meaning all the vertices are unvisited at the beginning. We use *visited* array to mark vertices visited during the chain decomposition. When a new back edge is visited for the first time, the algorithm traverses the path starting with the back edge followed by a sequence of tree edges (towards the root) untill it encounters a marked vertex, and also marks all the vertices on this path. By checking whether the vertices are marked or not, we can also distinguish whether an edge is encountered for the first time or has already been processed. Note that this procedure constructs the chains on the fly.

To check whether an edge is a bridge or not, we first note that only the tree edges can be bridges (back edges always form a cycle along with some tree edges). Also, from Theorem 3.15, it follows that any (tree) edge that is not covered in the chain decomposition algorithm is a bridge. Thus, to report these, we maintain a bitvector M of length n, corresponding to the n vertices, initialized to all zeros. Whenever a tree edge (u, v) is traversed during the chain decomposition algorithm, if v is the child of u, then we mark the child node v in the bit vector M. After reporting all the chains, we scan the bitvector M to find all unmarked vertices v and output the edges (u, v), where u is the parent of v, as bridges. If there are no bridges found in this process, then G is 2-edge connected. To check whether a vertex is a cut vertex (using the characterization in Theorem 3.15), we keep track the starting vertex of the current chain (except for the first chain, which is a cycle), that is being traversed, and report that vertex as a cut vertex if the current chain is a cycle. If there are no cut vertices found in this process then G is 2-vertex connected. Otherwise, we keep one more array of size n bits to mark which vertices are cut vertices. This completes the proof. $\square$

Note that, all of our algorithms in this section do not use cross pointers and hence, we can just assume that the input graph G is represented via *adjacency array* i.e., given a vertex v and an integer k, we can access the k-th neighbor of vertex v in constant time.

3.7 Other applications using $O(n \lg(m/n))$ bits

The following lemma can be proved along the same lines as the proof of Lemma 3.17.

Lemma 3.20. *Given an undirected graph G on n vertices with m edges, one can construct an auxiliary structure of size $O(n \lg(m/n))$ bits that can store $O(\lg d_i)$-bit satellite information with vertex v_i having degree d_i, in $O(m)$ time. Also, the satellite information associated with any vertex can be updated in $O(1)$ time.*

Using the above lemma, we show the following:

Theorem 3.21. *Given a directed acyclic graph G, its vertices can be output in topologically sorted order using $O(m + n)$ time using $O(n \lg(m/n))$ bits of space. The algorithm can also detect if G is not acyclic.*

Proof. A standard algorithm repeatedly outputs a vertex with indegree zero and deletes that vertex along with its outgoing edges, until there are no more vertices. To implement this, we maintain first the set Z of indegree 0 vertices in the data structure of Theorem 3.3 to support findany operation in constant time. This takes $O(m + n)$ time and $n + o(n)$ bits. We also represent the indegree sequence of the vertices using the data structure of Lemma 3.20 where the satellite information stored with vertex v_i is its indegree. The algorithm repeatedly finds any element from Z, outputs and deletes it from Z. Then, it decrements the indegree of its out-neighbors, and includes them in Z if any of them has become 0 (that can be determined in constant time as the indegrees are decremented) in the process. If Z becomes empty even before all elements are output (that can be checked using a counter or a bit vector), then at some intermediate stage of the algorithm, we did not encounter a vertex with indegree zero which means that the graph is not acyclic. □

An undirected graph is d-degenerate if every induced subgraph of the graph has a vertex with degree at most d. For example, a graph with degree at most d is d-degenerate. A planar graph is 5-degenerate as every planar graph has a vertex with degree at most 5. The degeneracy order of a d-degenerate graph is an ordering $v_1, v_2, \dots v_n$ of the vertices such that v_i has degree at most d among $v_{i+1}, v_{i+2}, \dots v_n$. We show the following using our data structure developed in this section.

Theorem 3.22. *Given a d-degenerate graph G, its vertices can be output in d-degenerate order using $O(n \lg(m/n))$ bits and $O(m + n)$ time. The algorithm can also detect if the given graph is not d-degenerate.*

Proof. As in the topological sort algorithm, we maintain the set Z of vertices whose degree in the entire graph is at most d, using our findany data structure. This takes $O(m + n)$ time and $n + o(n)$ bits. Then, we represent the degree sequence of the vertices using the data structure of Lemma 3.20 where the satellite information stored with vertex v_i is $x_i = max\{0, d_i - d\}$ where d_i is the degree of the i-th vertex. The algorithm repeatedly finds any element from Z, outputs and deletes it from Z. Then, it decrements the degree of its neighbors, and includes them in Z if any of them has become 0. If Z becomes empty even before all elements are output (that can be checked using a counter or a bitvector), then at some intermediate stage of the algorithm, we did not encounter a vertex with degree less than d, which means that the graph is not d-degenerate. □

Note that, both the algorithms discussed in this section assume that the input graph is represented as the standard *adjacency list* [51].

3.8 DFS and its applications using $O(n \lg \lg n)$ bits

In what follows we show that for dense graphs we can achieve slightly better space bounds for the problems mentioned above. Let T denote the DFS tree of G. Following Kammer et al. [99], we call a tree edge (u, v) of T with u being the parent of v *full marked* if there is a back edge from a descendant of v to a strict ancestor of u, *half marked* if it is not full marked and there exists a back edge from a descendant of v to u, and *unmarked*, otherwise. They use this definition to prove the following theorem:

Theorem 3.23 ([99]). *Let T denote a DFS tree of a graph G with root r, then the following holds:*

1. *every vertex u (except the root r) is a cut vertex exactly if at least one of the edges from u to one of its children is either an unmarked edge or a half marked edge, and*
2. *root r is a cut vertex exactly if it has at least two children in T.*

Now based on the above characterization and using the $O(m+n)$ time and $O(n \lg\lg n)$ space DFS algorithm of Theorem 3.10, they gave $O(m + n)$ time and $O(n \lg\lg n)$ bits algorithm to test/report if G has any cut vertex. Our main observation is that we can give a similar characterization for bridges in G, and essentially using the same implementation as theirs, we can also obtain $O(m + n)$ time and $O(n \lg\lg n)$ bits algorithms for testing 2-edge connectivity and reporting bridges of G. We start with the following lemma.

Lemma 3.24. *A tree edge $e = (u, v)$ in T is a bridge of G if and only if it is unmarked.*

Proof. If e is unmarked, then no descendants of v reaches u or any strict ancestor of u, so deleting e would result in disconnected graph, thus e has to be a bridge. On the other direction, it is easy to see that if e is a bridge, it has to be an unmarked edge. □

Now we state our theorem below.

Theorem 3.25. *Given an undirected graph G, in $O(m + n)$ time and $O(n \lg\lg n)$ bits of space we can determine whether G is 2-edge connected. If G is not 2-edge connected, then in the same amount of time and space, we can compute and output all the bridges of G.*

Proof. Using Lemma 3.24 and the exact implementation of using stack compression and other tools of the algorithm provided in Section 3.2 of Kammer et al. [99], we can prove the above theorem. □

Now combining the results of Theorem 3.10 [69], Theorem 3.25 and the result of [99], we obtain the following,

Theorem 3.26. *A DFS traversal of an undirected or directed graph G can be performed in $O(m + n)$ time using $O(n \lg\lg n)$ bits of space. In the same amount of time and space, given a connected undirected graph G, we can test if G is 2-vertex (and/or edge) connected. If not, in the same amount of time and space, we can compute and report all the bridges and cut vertices of G.*

Note that the space bound of Theorem 3.26 improves the results of Theorem 3.19 and Theorem 3.16 for sufficiently dense graphs (when $m = \omega(n \lg\lg n)$ and $m = \omega(n \lg^{O(1)} n)$ respectively) while keeping the same linear runtime. Thus combining all these results, we have the following unified result.

Theorem 3.27. *A DFS traversal of an undirected or directed graph G can be performed in $O(m + n)$ time using $O(n.\mathsf{min}\{\lg(m/n), \lg\lg n\})$ bits of space. In the same amount of time and space, given a connected undirected graph G, we can test if G is 2-vertex (and/or edge) connected. If not, in the same amount of time and space, we can compute and report all the bridges and cut vertices of G.*

3.9 Concluding remarks and open problems

We have provided several implementations of BFS focusing on optimizing space without much degradation in time. In particular with $2n + o(n)$ bits we get an optimal linear time algorithm whereas squeezing space further gives an algorithm with running time $O(mf(n) \lg n)$ where $f(n)$ can be any (extremely slow-growing) function of n. One can immediately obtain similar time-space tradeoffs for natural applications of BFS including testing whether a graph is bipartite or to obtain all connected components of a graph. To achieve this, we also provide a simple and space efficient implementation of the *findany* data structure. This data structure supports in constant time, apart from the standard insert, delete and membership queries, the operations findany and enumerate. Very recently, Poyias et al. [117] considered the problem of compactly representing a rewritable array of bit-strings, and for this, they used our findany data structure. It would be interesting to find other such applications for our data structure. Using the findany data structure, we also provide a space efficient implementation of the *decrement* data structure which supports in constant time the decrement and check if any element is zero operations. We use this decrement data structure to design space efficient algorithms for performing topological sort in a directed acyclic graph and computing degeneracy ordering of a given undirected graph. For the MST problem, we could reduce the space further to $n + o(n)$ bits. It is an interesting question whether we can perform BFS using $n + o(n)$ bits with a runtime of $O(m \lg^c n)$ for some constant c or even $O(mn)$.

For DFS, we provide an $O(m+n)$ time and $O(n.\mathsf{min}\{\lg(m/n), \lg\lg n\})$ bits of space algorithm. For a large class of graphs including planar, bounded degree and bounded treewidth graphs, this gives an $O(n)$ bits and $O(m+n)$ time DFS algorithm. Within the same time and space bound, we also show how to test biconnectivity and 2-edge connectivity, obtain cut vertices and bridges, and compute a chain decomposition of a given undirected graph G. It is a challenging and interesting open problem whether DFS and all these applications can be performed using $O(m+n)$ time and $O(n)$ bits for dense graphs as well. In the next chapter, we show that we can come pretty close. I.e., we can design $O(n)$ bits algorithms for most of the problems considered in this chapter and others, albeit with a slightly more than linear running time.

Chapter 4

Time Efficient Linear Bits Algorithms for DFS, MCS and Applications

4.1 Introduction

In the last chapter our goal was to obtain as far as possible best space bounds for DFS, BFS and many of their applications without compromising the linear running time. In this chapter, we take a look at those problems from different a angle i.e., here we focus on developing $O(n)$ bits algorithms for them. As we see later, we often pay some extra polylog multiplicative factor in the running time to achieve the goal of designing linear bits algorithms. As in the previous chapter, here also we work with the same *register input model*, and we assume that the input graphs $G = (V, E)$ are represented using *adjacency array*.

4.1.1 Our results and organization of the chapter

First, as a warm up, we start with some simple applications of the space efficient DFS to show the following.

- An $O(m \lg n \lg \lg n)$ time and $O(n)$ bits of space algorithm to compute the strongly connected components of a directed graph in Section 4.2.1.

In addition, we also give

- an algorithm to output the vertices of a directed acyclic graph in a topologically sorted order in Section 4.2.2, and
- an algorithm to find a sparse (with $O(n)$ edges) spanning biconnected subgraph of an undirected biconnected graph in Section 4.2.3

both using asymptotically the same time and space used for DFS, i.e., using $O(n)$ bits and $O(m \lg \lg n)$ time.

To develop fast and space efficient algorithms for other non-trivial graph problems which are also applications of DFS, in Section 4.3, we develop and describe in detail a space efficient tree covering technique, and use this in subsequent sections. This technique, roughly speaking, partitions the DFS tree into connected smaller sized subtrees which can be stored using less space. Finally we solve the corresponding graph problem on these smaller sized subtrees and merge the solutions across the subtrees to get an overall solution. All of these can be done using less space and not paying too much penalty in the running time. Some of these ideas are borrowed from succinct tree representation literature.

As the first application, we consider in Section 4.4.1, a space efficient implementation of chain decomposition of an undirected graph. As we have seen in the previous chapter, this is an important preprocessing routine for an algorithm to find cut vertices, biconnected components, cut edges, and also to test 3-connectivity [123] among others. We provide an algorithm that takes $O(m \lg^2 n \lg\lg n)$ time using $O(n)$ bits of space, improving on previous implementations that took $\Omega(n \lg n)$ bits [124] or $\Theta(m + n)$ bits [14] of space.

In Section 4.4.2, we give improved space efficient algorithms for testing whether a given undirected graph G is biconnected, and if G is not biconnected, we also show how one can find all the cut vertices of G. For this, we provide a space efficient implementation of Tarjan's classical lowpoint algorithm [128]. Our algorithms take $O(m \lg n \lg\lg n)$ time and $O(n)$ bits of space. In Section 4.4.3, we provide a space efficient implementation for testing 2-edge connectivity of a given undirected graph G, and producing cut edges of G using $O(m \lg n \lg\lg n)$ time and $O(n)$ bits of space.

Given a biconnected graph, and two distinguished vertices s and t, st-numbering is a numbering of the vertices of the graph so that s gets the smallest number, t gets the largest and every other vertex is adjacent both to a lower-numbered and to a higher-numbered vertex. Finding an st-numbering is an important preprocessing routine for a planarity testing algorithm [72, 73] among others. In Section 4.4.4, we give an algorithm to determine an st-numbering of a biconnected graph that takes $O(m \lg^2 n \lg\lg n)$ time using $O(n)$ bits. This improves the earlier implementations that take $\Omega(n \lg n)$ bits [31, 62, 72, 130]. Using this as a subroutine, in Section 4.5, we provide improved space effcient implementation for two well-known problems i.e., two-partitioning and two independent spanning tree problems.

Moving on from DFS and its applications, in Section 4.6 we introduce Maximum Cardinality Search (MCS) and provide its implementation as described in [131]. This is followed by various space efficient implementations of MCS in Section 4.7. Next we show, in Section 4.8, some applications of our MCS algorithm by providing space efficient procedures to recognize chordal graphs, find independent set and proper coloring in chordal graphs.

4.1.2 Preliminaries

Related work on Space-efficient DFS: Elmasry et al. [69] showed the following tradeoff result for DFS,

Theorem 4.1 ([69])**.** *For every function $t : \mathbb{N} \to \mathbb{N}$ such that $t(n)$ can be computed within the resource bound of this theorem (e.g., in $O(n)$ time using $O(n)$ bits), the vertices of a directed or undirected graph G can be visited in depth first order in $O((m+n)t(n))$ time with $O(n + n\frac{\lg\lg n}{t(n)})$ bits.*

In particular, fixing $t(n) = O(\lg\lg n)$, one can obtain a DFS implementation which runs in $O(m \lg\lg n)$ time using $O(n)$ bits. We build on top of this DFS algorithm to provide space efficient implementation for various applications of DFS in directed and undirected graphs in the rest of this chapter.

4.2 Some simple applications of DFS using $O(n)$ bits

Classical applications of DFS in directed graphs (see [51]) are to find strongly connected components of a directed graph, and to do a topological sort of a directed acyclic graph among many others. Also, given an undirected biconnected graph G, DFS is used as the main tool to produce a sparse spanning biconnected subgraph of G where the goal is to produce a sparse (with $O(n)$ edges) subgraph of G which contains all the vertices of G and still remains biconnected. We show here that while topological sort and producing a sparse spanning biconnected subgraph of an undirected biconnected graph can be solved using the same $O(n)$ bits and $O(m \lg\lg n)$ time (as for DFS), strongly connected components of a directed graph can be obtained using $O(n)$ bits and $O(m \lg n \lg\lg n)$ time.

4.2.1 Strongly Connected Components

There is a classical two pass algorithm (see [51] or [58]) for computing the Strongly Connected Components (SCC) of a given directed graph G which works as follows. In the first step, it runs a DFS on G^R, the reverse graph of G. In the second pass, it runs the connected component algorithm using DFS in G but it processes the vertices in the decreasing order of the finishing time from the first pass.

We can obtain G^R by switching the role of in and out adjacency arrays present in the input representation. As we can not remember the vertex ordering from the first pass due to space restriction, we process them in batches of size $n/\lg n$ in the reverse order i.e., we run a full DFS in G^R to obtain and store the last $n/\lg n$ vertices in an array A as they are the ones which have the highest set of finishing numbers in decreasing order. I.e., we maintain A as a queue of size $n/\lg n$ and as and when a new element is finished, it is added to the queue and the element with the earliest finish time at the other end of the queue is deleted. Now, we pick the vertices from A one by one in the order from the queue with the latest finish time and start a fresh DFS in G to compute the connected components and output all the vertices reachable as a SCC. The output vertices are marked in a bitmap so that we don't output them again. Once we are done with all the vertices in A, we restart the DFS from the beginning and produce the next chunk of $n/\lg n$ vertices by remembering the last vertex produced in the previous step and stop as soon as we hit that boundary vertex. Then we repeat the connected component algorithm from this chunk of vertices and continue this way. It is clear that the algorithm produces the SCCs correctly. As we are calling the DFS algorithm $O(\lg n)$ times, total time taken by this algorithm is $O(m \lg\lg n \lg n)$ with $O(n)$ bits of space. Hence, we have the following,

Theorem 4.2. *Given a directed graph G on n vertices and m edges, represented as in/out adjacency array, we can output the strongly connected components of G in $O(m \lg n \lg\lg n)$ time and $O(n)$ bits of space.*

4.2.2 Topological Sort

The standard algorithm for computing topological sort [51] outputs the vertices of a DFS in reverse order. If we can keep track of the DFS numbers, then reversing is an easy task. While working in space restricted setting (with $o(n \lg n)$ bits), this is a challenge as we don't have space to keep track of the DFS order. We can do as we did in the strongly connected components algorithm in the last section, by storing and outputting vertices in batches of $n/\lg n$ resulting in an $O(m \lg n \lg\lg n)$ time algorithm.

Elmasry et al. [69] showed that, the vertices of a DAG G can be output in the order of a topological sort within the time and space bounds of a DFS in G plus an additional $O(n \lg\lg n)$ bits. As they also showed how to perform DFS in $O(m + n)$ time and $O(n \lg\lg n)$ bits, overall their algorithm takes $O(m+n)$ time and $O(n \lg\lg n)$ bits to compute a topological sorting of G. Their main idea is to maintain enough information about a DFS to resume it in the middle and apply this repeatedly to reverse small chunks of its output, produced in reverse order, one by one.

We observe that, instead of storing information to restart DFS and produce the reverse order, we simply work with the reverse graph itself (which can be obtained from the input representation by switching the role of in and out adjacency arrays) and do a DFS in the reverse graph and output vertices as they are finished (or blackened) i.e., in the increasing order of finishing time. To see the correctness of this procedure, note that the reverse graph is also a DAG, and if (i, j) is an edge in the DAG G, then (j, i) is an edge in the reverse graph and i will become black before j while the algorithm performs DFS in the reverse graph. Hence, i will be placed before j in the correct topological sorted order. Thus we have the following,

Theorem 4.3. *Given a DAG G on n vertices and m edges, if the black vertices of the DFS of G can be output using $s(n)$ space and $t(n)$ time, then its vertices can be output in topologically sorted order using $O(s(n))$ space and $O(t(n))$ time assuming that the input* *representation has both the in and out adjacency array of the graph.*

From Theorem 4.1 (setting $t(n) = O(\lg\lg n)$) and Theorem 4.3, we have the following.

Corollary 4.4. *Given a DAG G on n vertices and m edges, its vertices can be output in topologically sorted order using $O(m\lg\lg n)$ time and $O(n)$ bits.*

Note that, we knew all along that DFS and topological sort take the same time, the main contribution of Theorem 4.3 is that it shows they take the same space (improving on the result of [69] where they showed that topological sort space = DFS space + $O(n\lg\lg n)$ bits under the same time) when both the in/out adjacency arrays are present in the input.

Now we proceed to produce an algorithm for topological sort that uses $o(n)$ bits of space, one of the very few such algorithms in this thesis.

4.2.3 Topological Sort in Sublinear Space

We note the following theorem of Asano et al. [9].

Theorem 4.5. *DFS on a DAG G can be performed in space $O(\frac{n}{2^{(\sqrt{\lg n})}})$ bits and in polynomial time.*

While it should immediately follow from Theorem 4.3 that topological sort can also be performed using similar space, there is one caveat. Asano et al.'s algorithm works assuming that the given DAG G has a single source vertex. In particular, they determine whether a vertex is black by checking whether it is reachable from *the* source without using the gray vertices (using the sublinear space reachability algorithm of [19]).

The algorithm can be easily extended to handle s many sources if we have some additional $s\lg n$ bits. We simply keep track of the indices of the sources from which DFS has been explored, and to determine whether a vertex is black, we ask if it is reachable from an earlier source or from the current source without using the gray vertices. Thus we have the following improved theorem.

Theorem 4.6. *DFS on DAG G with s sources can be performed using $s \lg n + o(n)$ bits and polynomial time. In particular, if s is $o(n/\lg n)$, the overall space used is $o(n)$ bits.*

Thus from Theorem 4.3 and Theorem 4.6 we obtain the following,

Theorem 4.7. *Topological Sort on a DAG G with s sinks can be performed using $s \lg n + o(n)$ bits and polynomial time. In particular if s is $o(n/\lg n)$, the overall space used is $o(n)$ bits.*

4.2.4 Finding a sparse biconnected subgraph of a biconnected graph

The problem of finding a k-connected spanning subgraph with the minimum number of edges of a k-connected graph is known to be NP-hard for any $k \geq 2$ [86]. But the complexity of the problem decreases drastically if all we want is to produce a "sparse" k-connected spanning subgraph, i.e., one with $O(n)$ edges. Nagamochi and Ibaraki [115] gave a linear time algorithm which produces a k-connected spanning subgraph with at most $kn - \frac{k(k+1)}{2}$ edges. Later, Cheriyan et al. [47] gave another linear time algorithm for $k = 2$ and 3 that produced a 2-connected spanning subgraph with at most $2n - 2$ edges, and a 3-connected subgraph with at most $3n - 3$ edges. Later, Elmasry [68] gave an alternate linear time algorithm for producing a sparse spanning biconnected subgraph of a given biconnected graph by performing a DFS with additional bookkeeping. In what follows, we provide a space efficient implementation for it. In order to do that, we start by briefly describing Elmasry's algorithm.

Let $DFI(v)$ denote the index (integer) that represents the time at which the vertex v is first discovered from the vertex u when performing a DFS i.e., u is the parent of v in the DFS tree. Let $low(v)$ be the smallest DFI value among the DFI values of vertices w such that (v, w) is a back edge. (Note that this quantity is different from the "lowpoint" value used in Tarjan's [128] classical biconnectivity algorithm.) Basically $low(v)$ captures the information regarding the deepest back edge going out of the vertex v. If v has no backedges, for convenience (the reason will become clear in the following lemma), we adopt the convention that $low(v) = DFI(parent(v))$. The edge $(v, low(v))$ is the deepest backedge out of v. Note that, it is actually the tree edge between v and its parent if v does not have a backedge. The algorithm maintains all the edges of the DFS tree. In addition, for every vertex in the graph, the algorithm maintains the DFI and the low values along with the back edge that realizes it. As the root of the DFS tree does not have any back edge and, as the underlying graph is 2-connected, the root has only one child v so that there is no back edge emanating from v as well. Thus we get at most $n-2$ back edges along with $n-1$ tree edges, giving a subgraph with at most $2n-3$ edges. Elmasry [68] proved that the resulting graph is indeed a spanning 2-connected subgraph of G. His algorithm takes $O(m+n)$ time and $O(n \lg n)$ bits of space. We improve the space bound, albeit with slight increase in time, by first proving a more general lemma as following,

Lemma 4.8. *Given any undirected graph G with n vertices and m edges, we can compute and report the $low(v)$ values i.e., deepest back edge going out of v in the DFS tree of G, for every vertex v, using $O(n)$ bits of space and $O(m \lg\lg n)$ time.*

Proof. The aim is to output all the deepest back edges out of every vertex v in G as we perform the DFS. As always, let $\{v_1, v_2, \cdots, v_n\}$ be the vertices of the graph. We perform a DFS with the usual color array and other relevant data structures (as required in Theorem 4.1 with $t(n) = \lg\lg n$) along with one more array of n bits, which we call DBE (for Deepest Back Edge) array, which is initialized to all zero. $DBE[i]$ is set to 1 if and only if the algorithm has found and output the deepest back edge emanating from vertex v_i. So whenever a white vertex v_i becomes grey (i.e., v_i is visited for the first

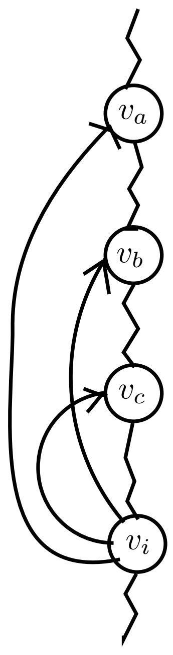

Figure 4.1: A part of the full DFS tree. The wiggling edges represent tree edges and the edges with arrow heads represent back edges. If $low(v_i) = v_a$, we would come across v_i in the adjacency array of v_a before encountering from the arrays of v_b and v_c. I.e., the back edge (v_a, v_i) will be processed before the other back edges (v_b, v_i) and (v_c, v_i) since we process the vertices (and the backedges incident to them) in their DFS order.

time), we scan v_i's adjacency array to mark, for every white neighbor v_j, $DBE[j]$ to 1 if and only if it was 0 before. The correctness of this step follows from the fact that as we are visiting vertices in DFS order, and if $DBE[j]$ is 0, then vertex v_j is not adjacent to any of the vertices we have visited so far, and as it is adjacent to v_i, the deepest back edge emanating from v_j is (v_i, v_j). Hence we output this edge and move on to the next neighbor and eventually with the next step of DFS until all the vertices are exhausted. This completes the description of the algorithm. See Figure 3.1 for an illustration. Now to see how this procedure produces all the deepest back edges out of every vertex, note that, at vertex v_i, our algorithm reports all the back edges $e = (v_i, v_j)$ where e is the deepest back edge from v_j, and also all tree edges (v_i, v_j) where v_j has no back edge. Observe that from our convention, in the second case, (v_i, v_j) is the deepest back edge out of v_j. This concludes the proof of the lemma. As we performed just one DFS to produce all such edges, using Theorem 4.1, the claimed running time and space bounds follow. □

The way we will actually use Lemma 4.8 in our algorithms, is for finding and storing the *low* values for at most $n/\lg n$ vertices. So we state a corollary for that.

Corollary 4.9. *Given any undirected graph G with n vertices and m edges and any set L of $O(n/\lg n)$ vertices as input, we can compute, report and store the $low(v)$ values for every vertex v in L in the DFS tree T of G using $O(n)$ bits of space and $O(m \lg\lg n)$ time.*

Note that, Lemma 4.8 holds true for any undirected connected graph G. In what follows, we use Lemma 4.8 to give a space efficient implementation of Elmasry's algorithm when the input graph G is an undirected biconnected graph. In particular, we show the following,

Theorem 4.10. *Given an undirected biconnected graph G with n vertices and m edges, we can output the edges of a sparse spanning biconnected subgraph of G using $O(n)$ bits of space and $O(m \lg\lg n)$ time.*

Proof. When the underlying graph G is undirected biconnected graph, we know that Elmasry's algorithm produces a sparse spanning subgraph which is also biconnected. In order to implement that, given an undirected biconnected graph G, we first run on G the algorithm of Lemma 4.8 which produces and reports all the deepest back edges out of all the vertices v in G. Out of all those deepest back edges, note that, some are actually tree edges from our convention. Hence, we don't want to report them multiple time. More specifically, if a vertex v_j has no back edge going out of it, Lemma 4.8 outputs the edge (v_i, v_j) as the deepest back edge out of v_j, which is actually a tree edge in the DFS tree T of G. In order to avoid reporting such edges more than once, we perform the following. During the scanning of v_i's adjacency array, we also check if any of its neighbor, other than its parent, is grey. If so, we report the edge from v_i to its parent.

Note that if v_i has a back edge to one of its ancestors (other than its parent), then this step reports the tree edge from v_i to its parent. Otherwise, v_i didn't have any back edge, and hence the tree edge to its parent would have been output while DFS was exploring and outputting deepest back edges from its parent; so we do not output the edge again. Note that, we can do this test along with the algorithm of Lemma 4.8 so that using just one DFS, we can produce all the tree edges and deepest back edges as required in Elmasry's algorithm. Thus using Theorem 4.1, we can output the edges of a sparse spanning biconnected subgraph of G using $O(n)$ bits of space and $O(m \lg \lg n)$ time. $\square$

4.3 Tree cover and space efficient construction

Before moving on to handle other complex applications of DFS in undirected graphs, namely biconnectivity, 2-edge connectivity and *st*-numbering, in the this section we discuss the common methodology to attack all of these problems. Once we set all our machinery here, in Section 4.3, we see afterwards how to use them almost in a similar fashion to several problems. Central to all of our algorithms following this section is a decomposition of the DFS tree. For this we use the well-known tree covering technique which was first proposed by Geary et al. [88] in the context of succinct representation of rooted ordered trees. The high level idea is to decompose the tree into subtrees called *minitrees*, and further decompose the mini-trees into yet smaller subtrees called *microtrees*. The microtrees are tiny enough to be stored in a compact table. The root of a minitree can be shared by several other minitrees. Thus to represent the tree, we only have to represent the connections between the subtrees. Later He et al. [95] extended this approach to produce a representation which supports several additional operations. Farzan and Munro [74] modified the tree covering algorithm of [88] so that each minitree has at most one node, other than the root of the minitree, that is connected to the root of another minitree. This simplifies the representation of the tree, and guarantees that in each minitree, there exists at most one non-root node which is connected to (the root of) another minitree. The tree decomposition method of Farzan and Munro [74] is summarized in the following theorem:

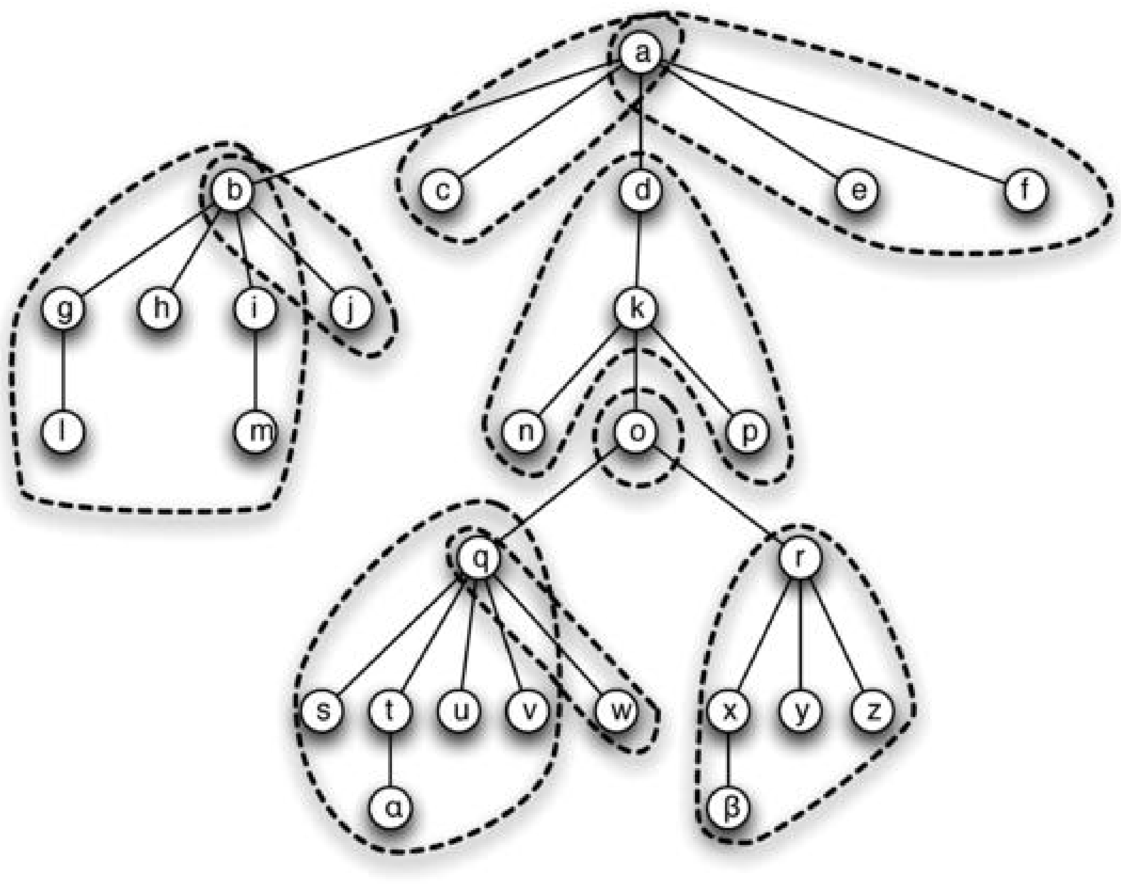

Figure 4.2: An illustration of Tree Covering technique with $L = 5$. The figure is reproduced from [74]. Each closed region formed by the dotted lines represents a minitree. Note that each minitree has at most one 'child' minitree (other than the minitrees that share its root) in this structure.

Theorem 4.11 ([74]). *For any parameter $L \geq 1$, a rooted ordered tree with n nodes can be decomposed into $\Theta(n/L)$ minitrees of size at most $2L$ which are pairwise disjoint aside from the minitree roots. Furthermore, aside from edges stemming from the minitree root, there is at most one edge leaving a node of a minitree to its child in another minitree. The decomposition can be performed in linear time.*

See Figure 3.2 for an illustration. In our algorithms, we apply Theorem 4.11 with $L = n/\lg n$. For this parameter L, since the number of minitrees is only $O(\lg n)$, we can represent the structure of the minitrees within the original tree (i.e., how the minitrees are connected with each other) using $O(\lg^2 n)$ bits. The decomposition algorithm of [74] ensures that each minitree has at most one 'child' minitree (other than the minitrees that share its root) in this structure. We use this property crucially in our algorithms. We refer to this as the *minitree-structure*. See Figure 3.3(a) for the minitree structure of the tree decomposition shown in Figure 3.2.

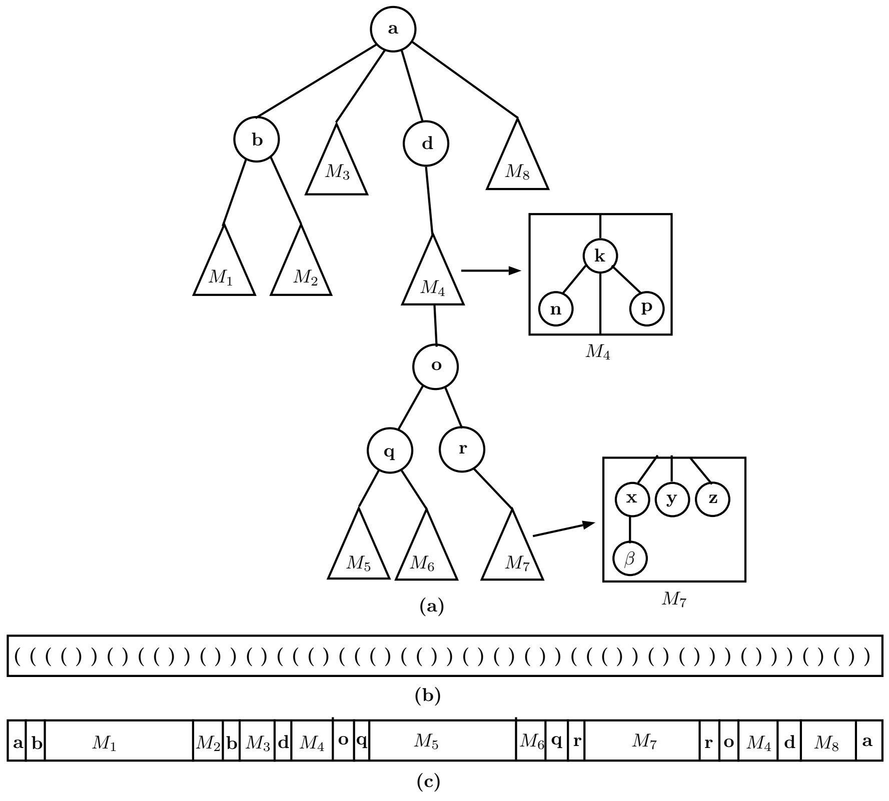

Figure 4.3: (a) The minitree structure of the tree decomposition shown in Figure 2. (b) This array encodes the entire DFS tree using the balanced parenthesis (BP) representation. (c) In this array, we demonstrate how the minitrees are split into a constant number of consecutive chunks in the BP representation. Note that the bottom array can actually be encoded using $O(\lg^2 n)$ bits, by storing, for each of the $O(\lg n)$ minitrees, pointers to all the chunks in BP sequence indicating the starting and ending positions of the chunks corresponding to the minitrees.

Explicitly storing all the minitrees (using pointers) requires $\omega(n)$ bits overall. One way to represent them efficiently using $O(n)$ bits is to store them using any linear-bit encoding of a tree. But these representations [95, 88] don't allow us to efficiently compute the preorder numbers of the nodes, for example, which is needed in our algorithms. Instead, we encode the entire tree structure using a linear-bit encoding, and store pointers into this encoding to represent the minitrees, as described below. We first encode the tree using the *balanced parenthesis* (BP) representation [105, 112], summarized in the following theorem.[1]

Theorem 4.12 ([105]). *Given a rooted ordered tree T on n nodes, it can be represented as a sequence of balanced parentheses of length $2n$. Given the preorder or postorder number of a node v in T, we can support subtree size, parent and i-th child on v in $O(1)$ time using an additional $o(n)$ bits.*

We now represent each minitree by storing pointers to the set of all *chunks* in the BP representation that together constitute the minitree. The following lemma by Farzan et al. [75, Lemma 2] (restated) shows that each minitree is split into a constant number of consecutive chunks in the BP sequence.

Lemma 4.13. *In the BP sequence of a tree, the bits corresponding to a mini-tree form a set of constant number of substrings. Furthermore, these substrings concatenated together in order, form the BP sequence of the mini-tree.*

Hence, one can store a representation of the minitrees by storing an $O(\lg^2 n)$-bit structure that stores pointers to the starting positions of the chunks corresponding to each minitree in the BP sequence We refer to the representation obtained using this tree covering (TC) approach as the TC representation of the tree. See Figure 3.3 for a complete example of a minitree structure along with the BP sequence of the tree of Figure 3.2. The following lemma shows that we can construct the TC representation of the DFS tree of a given graph, using $O(n)$ additional bits.

Lemma 4.14. *Given a graph G on n vertices and m edges, if there is an algorithm that*

[1]The representation of [112] does not support computing the i-th child of a node in constant time while the one in [105] can. When using these representations to produce a tree cover, the representation of [112] is sufficient as we just need to compute the 'next child' as we traverse the tree in post-order computing the subtree sizes of each subtree.

takes $t(n, m)$ time and $s(n, m)$ bits to perform DFS on G, then one can create the TC representation of the DFS tree in $t(n, m) + O(n)$ time, using $s(n, m) + O(n)$ bits.

Proof. We first construct the balanced parenthesis (BP) representation of the DFS tree as follows. We start with an empty sequence, BP, and append parentheses to it as we perform each step of the DFS algorithm. In particular, whenever the DFS visits a vertex v for the first time, we append an open parenthesis to BP. Similarly when DFS backtracks from v, we append a closing parenthesis. At the end of the DFS algorithm, as every vertex is assigned a pair of parenthesis, length of BP is $2n$ bits. We just need to run the DFS algorithm once to construct this array, hence the running time of this algorithm is asymptotically the same as the running time of the DFS algorithm.

We construct auxiliary structures to support navigational operations on the DFS tree using the BP sequence, as mentioned in Theorem 4.12. This takes $o(n)$ time and space using the algorithm of [87]. We then use the BP sequence along with the auxiliary structures to navigate the DFS tree in postorder, and simulate the tree decomposition algorithm of Farzan and Munro [74] for constructing the TC representation of the DFS tree. If we reconstruct the entire tree (with pointers), then the intermediate space would be $\Omega(n \lg n)$ bits. Instead, we observe that the tree decomposition algorithm of [74] never needs to keep more than $O(L)$ *temporary components* (see [74] for details) in addition to some of the *permanent components*. Each component (permanent or temporary) can be stored by storing the root of the component together with its subtree size. Since $L = n/\lg n$, and the number of permanent components is only $O(\lg n)$, the space required to store all the permanent and temporary components at any point of time is bounded by $O(n)$ bits. The construction algorithm takes $O(n)$ time. □

We use the following lemma in the description of our algorithms in the later sections.

Lemma 4.15. *Let G be a graph, and T be its DFS tree. If there is an algorithm that takes $t(n, m)$ time and $s(n, m)$ bits to perform DFS on G, then, using $s(n, m) + O(n)$*

bits, one can reconstruct any minitree given by its ranges in the BP sequence of the TC representation of T, along with the labels of the corresponding nodes in the graph in $O(t(n, m))$ time.

Proof. We first perform DFS to construct the BP representation of the DFS tree, T. We then construct the TC representation of T, as described in Lemma 4.14. We now perform DFS algorithm again, keeping track of the preorder number of the current node at each step. Whenever we visit a new node, we check its preorder number to see if it falls within the ranges of the minitree that we want to reconstruct. (Note that, as mentioned above, from [75, Lemma 2], the set of all preorder number of the nodes that belong to any minitree form a constant number of ranges, since these nodes belong to a constant number of chunks in the BP sequence.) If it is within one of the ranges corresponding to the minitree being constructed, then we add the node along with its label to the minitree. □

4.4 Applications of DFS using tree-covering technique

In this section, we provide $O(n)$ bit implementations of various algorithmic graph problems that use DFS, by using the tree covering technique developed in the previous section. At a higher level, we use the tree covering technique to generate the minitrees one by one, and then partially solve the corresponding graph problem inside that minitree before finally combining the solution across all the minitrees. The problems we consider include algorithms to test biconnectivity, 2-edge connectivity and to output cut vertices, edges, and to find a chain decomposition and an *st*-numbering among others. To test for biconnectivity and related problems, the classical algorithm due to Tarjan [128, 129] computes the so-called "low-point" values (which are defined in terms of a DFS-tree) for every vertex v, and checks some conditions based on these values. Brandes [31] and Gabow [85] gave considerably simpler algorithms for testing biconnectivity and computing biconnected components by using some path-generating rules instead of low-points; they call these algorithms path-based. An algorithm due to Schmidt [124] is based on chain decomposition of graphs to determine biconnectivity (and/or 2-edge connected). All these algorithms take $O(m+n)$ time and $O(n)$ words of space. Roughly these approaches compute DFS and process the DFS tree in specific order maintaining some auxiliary information of the nodes. Readers are referred to Section 3.5.2 for the full explanation of the chain decomposition and its application.

4.4.1 Chain decomposition

In what follows, we describe an implementation of Schmidt's chain decomposition algorithm using only $O(n)$ bits of space and in $O(m \lg^2 n \lg\lg n)$ time using our partition of the DFS tree of Section 4.3. In the following description, *processing a back edge* refers to the step of outputting the chain (directed path or cycle) containing that edge and marking all the encountered vertices as visited. Processing a node refers to processing all the back edges out of that node. The main idea of our implementation is to process all the back edges out of each node in their *preorder* (as in Schmidt's algorithm). To perform this efficiently (within the space limit of $O(n)$ bits), we process the nodes in *chunks* of size $n/\lg n$ each (i.e., the first chunk of $n/\lg n$ nodes in preorder are processed, followed by the next chunk of $n/\lg n$ nodes, and so on). But when processing the back edges out of a chunk C, we process all the back edges that go from C to all the minitrees in their *postorder*, processing all the edges from C to a minitree τ_1 before processing any other back edges going out of C to a different minitree. This requires us to go through all the edges out of each chunk at most $O(\lg n)$ times (once for each minitree). Thus the order in which we process the back edges is different from the order in which we process them in Schmidt's algorithm, but we argue that this does not affect the correctness of the algorithm. In particular, we observe the following easy to see fact:

- Schmidt's algorithm correctly produces a chain decomposition even if we process vertices to any order – for example, in level order instead of preorder, as long as we process a vertex v only after all its ancestors are also processed. This also implies that as long as we process the back edges coming to a vertex v (from any of its descendants) only after we process all the back edges going to any of its ancestors from any of v's descendants, we can produce a chain decomposition correctly.

To process a back edge (u, v) between a chunk C and a minitree τ, where u belongs to C, v belongs to τ, and u is an anscestor of v, we first output the edge (u, v), and then traverse the path from v to the root of τ, outputting all the traversed edges and marking the nodes as visited. We then start another DFS to produce the minitree τ_p containing the parent p of the root of τ, and output the path from p to the root of τ_p, and continue the process untill we reach a vertex that has been marked as visited. Note that this process will terminate since u is marked and is an ancestor of v. We maintain a bitvector of length n to keep track of the marked vertices to perform this efficiently. A crucial observation that we use in bounding the runtime is that once we produce a minitree τ_p for a particular pair (C, τ), we don't need to produce it again, as the root of τ will be marked after the first time we output it as part of a chain. Also, once we generate a chunk C and a minitree τ, we go through all the vertices of C in preorder, and process all the edges that go between C and τ. We provide the pseudocode (see Algorithm 1) below describing the high-level algorithm for outputting the chain decomposition.

The time taken for the initial part, where we construct the DFS tree, decompose it into minitrees, and construct the auxiliary structures, is $O(m \lg\lg n)$, using Theorem 4.1 with $t(n) = \lg\lg n$. The running time of the rest of the algorithm is dominated by the cost of processing the back edges. As outlined in Algorithm 1, we process the back edges between every pair (C_i, τ_j), where C_i is the i-th chunk of $n/\lg n$ nodes in preorder, and τ_j is the j-th minitree in postorder, for $1 \leq i \leq \lg n$ and $1 \leq j \leq l$. The outer loop of the

Algorithm 1 Chain Decomposition

```
   Let $\tau_1, \tau_2, \cdots, \tau_l$ be the minitrees in postorder and $C_1, C_2, \cdots, C_m$ be the chunks of
   vertices in preorder where $l = O(\lg n)$ and $m = \lg n$
1: for $i = 1$ to $m$ do
2:     for $j = 1$ to $l$ do
3:         for all back edges $(u, v)$ with $u \in C_i$ and $v \in \tau_j$ do
4:             output the chain containing the edge $(u, v)$
5:         end for
6:     end for
7: end for
```

algorithm generates each chunk in preorder, and thus requires a single DFS to produce all the chunks over the entire execution of the algorithm. The inner loop goes through all the minitrees for each chunk. Since there are $\lg n$ chunks and $O(\lg n)$ minitrees, and producing each minitree takes $O(m \lg\lg n)$ time, the generation of all the chunk-minitree pairs takes $O(m \lg\lg n \lg^2 n)$ time.

For a particular pair (C, τ), we may need to generate many ($O(\lg n)$, in the worst-case) minitrees. But we observe that, this happens for at most one back edge for a every pair (C, τ), since after processing the first such back edge, the root of the minitree τ is marked and hence any chain that is output afterwards will stop before the root of the minitree. Also, if a minitree τ_ℓ is generated when processing a pair (C, τ), then it will not be generated when processing any other pair (C', τ') which is different from (C, τ) (since each minitree has at most one child minitree). Thus the overall running time is dominated by generating all the pairs $C, \tau)$, which is $O(m \lg^2 n \lg\lg n)$. Thus, we obtain the following.

Theorem 4.16. *Given an undirected graph G on n vertices and m edges, we can output a chain decomposition of G in $O(m \lg^2 n \lg\lg n)$ time using $O(n)$ bits.*

4.4.2 Testing biconnectivity and finding cut vertices

A naïve algorithm to test for biconnectivity of a graph $G = (V, E)$ is to check if $(V \setminus \{v\}, E)$ is connected, for each $v \in V$. Using the $O(n)$ bits and $O(m + n)$ time BFS algorithm

[14] for checking connectivity, this gives a simple $O(n)$ bits algorithm running in time $O(mn)$. Another approach is to use Theorem 4.16 combining with criteria mentioned in Theorem 3.15 to test for biconnectivity and output cut vertices in $O(m \lg^2 n \lg\lg n)$ time using $O(n)$ bits.

Here we show that using $O(n)$ bits we can design an even faster algorithm running in $O(m \lg n \lg\lg n)$ time. If G is not biconnected, then we also show how one can find all the cut-vertices of G within the same time and space bounds. We implement the classical low-point algorithm of Tarjan [128]. Recall that, the algorithm performs a DFS and computes for every vertex v, a value lowpoint$[v]$ which is recursively defined as

$$\begin{aligned}\text{lowpoint}[v] = \min\{\ & DFI(v) \cup \{\text{lowpoint}[s] |\ s \text{ is a child of } v\} \\ & \cup \{DFI(w) | (v,w) \text{ is a back-edge}\}\ \}\end{aligned}$$

Tarjan proved that if a vertex v is not the root, then v is a cut vertex if and only if v has a child w such that lowpoint$[w] \geq DFI(v)$. The root of a DFS tree is a cut vertex if and only if the root has more than one child. Since the lowpoint values require $\Omega(n \lg n)$ bits in the worst case, this poses the challenge of efficiently testing the condition for biconnectivity with $O(n)$ bits. To deal with this, as in the case of the chain decomposition algorithm, we compute lowpoint values in $O(\lg n)$ batches using our tree covering algorithm. Cut vertices encountered in the process, if at all, are stored in a separate bitmap. We show that each batch can be processed in $O(m \lg\lg n)$ time using DFS, resulting in an overall runtime of $O(m \lg n \lg\lg n)$.

Computing lowpoint and reporting cut vertices: We first obtain a TC representation of the DFS tree using the decomposition algorithm of Theorem 4.11 with $L = n/\lg n$. We then *process* each minitree, in the postorder of the minitrees in the minitree structure. To process a minitree, we compute the lowpoint values of each of the nodes in the minitree (except possibly the root of the minitree) in overall $O(m)$ time. During the processing of any minitree, if we determine that a vertex is a cut vertex, we store this information by marking the corresponding node in a seperate bit vector. Each minitree can be reconstructed in $O(m \lg\lg n)$ time using Lemma 4.15. The lowpoint value of a node is a function of the lowpoints of all its children. However the root of a minitree may have children in other minitress. Hence for the root of the minitree, we store the partial lowpoint value (till that point) which will be used to update its value when all its subtrees have computed their lowpoint values (possibly in other minitrees).

Computing the lowpoint values in each of the minitrees is done in a two step process. In the first step, we compute and store the *low* values of each node (which is the DFI value of the deepest back edge emanating from that node) belonging to the minitree using Corollary 4.9. Note that the *low* values form one component of the values among which we find the minimum, in the definition of *lowpoint* above, with a slight change. I.e., if a vertex v has a backedge, then $low(v)$ is nothing but $\min\{DFI(w) : (v, w) \textit{ is a back edge}\}$. However, if v does not have a backedge, by our convention $low(v)$ has the DFI value of its parent which needs to be discounted in computing *lowpoint* value of v. This is easily done if we also remember the DFI value of the parent of every node in the minitree (using $O(n)$ bits).

Once these $low(v)$ values are computed and stored for all the vertices v belonging to a minitree, they are passed on to the next step for computing $lowpoint(v)$ values. More specifically, in the second step, we do another DFS starting at the root of this minitree and compute the lowpoint values for all the vertices v belonging to a minitree exactly in the same way as it is done in the classical Tarjan's [128] algorithm using the explicitly stored $low(v)$ values. We provide the code snippet which actually shows how to compute lowpoint values recursively for a minitree in Algorithm 2. Thus we obtain the following,

Lemma 4.17. *Computing and storing the $lowpoint(v)$ values for all the nodes v in a minitree can be performed in $O(m \lg\lg n)$ time, using $O(n)$ bits.*

Algorithm 2 DFS(v)

```
1:  if $low(v) = DFI(parent(v))$ then
2:  $lowpoint(v) = DFI(v)$
3:  else $lowpoint(v) = Min\{DFI(v), low(v)\}$
4:  for all $y \in adj(v)$ do
5:      if $y$ is white then
6:          $DFI(y) \leftarrow DFI(v) + 1$
7:          DFS(y)
8:          if $lowpoint(y) < lowpoint(v)$ then
9:              $lowpoint(v) = lowpoint(y)$
10:         end if
11:     end if
12: end for
```

To compute the effect of the roots of the minitrees on the *lowpoint* computation, we store various $\Theta(\lg n)$ bit information with each of the $\Theta(\lg n)$ minitree roots including their partial/full lowpoint values, the rank of its first/last child in its subtree. After we process one minitree, we generate the next minitree, in postorder, and process it in a similar fashion and continue until we exhaust all the minitrees. As we can store the cut vertices in a bitvector B of size n marking $B[i] = 1$ if and only if the vertex v_i is a cut vertex, reporting them at the end of the execution of the algorithm is a routine task. Clearly we have taken $O(n)$ bits of space and the total running time is $O(m \lg\lg n \lg n)$ as we run the DFS algorithm $O(\lg n)$ times overall. Thus we have the following

Theorem 4.18. *Given an undirected graph G with n vertices and m edges, in $O(m \lg n \lg\lg n)$ time and $O(n)$ bits of space we can determine whether G is 2-vertex connected. If not, in the same amount of time and space, we can compute and report all the cut vertices of G.*

4.4.3 Testing 2-edge connectivity and finding bridges

The classical algorithm of Tarjan [129] takes $O(m+n)$ time using $O(n)$ words to check if G is 2-edge connected. Schmidt's algorithm [124] which is based on chain decomposition can also be implemented in linear time but with $O(m)$ words. The purpose of this section is to improve the space bound to $O(n)$ bits, albeit with slightly increased running time.

For this, we use the following folklore characterization:

- A tree edge (v, w), where v is the parent of w, is a bridge if and only if lowpoint$[w]$ $> DFI(v)$.

That is, a tree edge (v, w) is a bridge if and only if the vertex w and any of its descedants in the DFS tree cannot reach to vertex v or any of its ancestors. Thus if the edge (v, w) is removed, the graph G becomes disconnected. Note that, since storing the lowpoint values requires $\Omega(n \lg n)$ bits, we cannot store all of them at once to check the criteria mentioned in the characterization, and this poses the challenge of efficiently testing the condition for 2-edge connectivity with only $O(n)$ bits. To perform this test in a space efficient manner, we extend ideas developed in the previous section.

Similar to the biconnectivity algorithm, here also we first construct a TC representation of the DFS tree using the decomposition algorithm of Theorem 4.11 with $L = n/\lg n$. We then *process* each minitree, in the postorder of the minitrees in the minitree structure. To process a minitree, we compute the lowpoint values of each of the nodes in the minitree (except possibly the root) in overall $O(m)$ time. While processing these minitrees, if we come across any bridge, we store it in a separate bitvector so that at the end of the execution of the algorithm we can report all of them. Using Lemma 4.15, we know that each minitree can be reconstructed in $O(m \lg\lg n)$ time, and also we store for the root the partially computed lowpoint (till the point we are done processing minitrees). Now we compute the lowpoint values for each of the vertices belonging to a minitree using Lemma 4.17.

Once we determine lowpoint values for all the vertices belonging to a minitree, we generate each minitree along with the node labels, and easily test whether any tree edge is a bridge using the characterization mentioned above. We also need to check this condition for edges that connect two minitrees; but this can also be done within the same time and space bounds. We store this information using a bit vector B of length $n-1$ such that

$B[i] = 1$ if and only if the i-th edge in pre-order, of the DFS tree, is a bridge. Thus, by running another DFS, we can report all the bridges of G. Clearly this procedure takes $O(n)$ bits of space and the total running time is $O(m \lg\lg n \lg n)$ as we run the DFS algorithm $O(\lg n)$ times overall. Hence we obtain the following.

Theorem 4.19. *Given an undirected graph G with n vertices and m edges, in $O(m \lg n \lg\lg n)$ time and $O(n)$ bits of space we can determine whether G is 2-edge connected. If G is not 2-edge connected, then in the same amount of time and space, we can compute and output all the bridges of G.*

4.4.4 *st*-numbering

The *st*-ordering of vertices of an undirected graph is a fundamental tool for many graph algorithms, e.g., in planarity testing and graph drawing. The first linear-time algorithm for *st*-ordering the vertices of a biconnected graph is due to Even and Tarjan [72], and is further simplified by Ebert [62], Tarjan [130] and Brandes [31]. All these algorithms, however, preprocess the graph using depth-first search, essentially to compute lowpoints which in turn determine an (implicit) open ear decomposition. A second traversal is required to compute the actual *st*-ordering. All of these algorithms take $O(n \lg n)$ bits of space. We give an $O(n)$ bits implementation of Tarjan's [130] algorithm.

We first describe the two pass classical algorithm of Tarjan without worrying about the space requirement. The algorithm assumes, without loss of generality, that there exists an edge between the vertices s and t, otherwise it adds the edge (s, t) before starting with the algorithm. Moreover, the algorithm starts a DFS from the vertex s and the edge (s, t) is the first edge traversed in the DFS of G. Let $p(v)$ be the parent of vertex v in the DFS tree. $DFI(v)$ and $lowpoint(v)$ have the usual meaning as defined previously. The first pass is a depth first search during which for every vertex v, $p(v)$, $DFI(v)$ and $lowpoint(v)$ are computed and stored. The second pass constructs a list L, which is initialized with

$[s, t]$, such that if the vertices are numbered in the order in which they occur in L, then we obtain an *st*-ordering. In addition, we also have a sign array of n bits, intialized with sign[s]=-. The second pass is a preorder traversal starting from the root s of the DFS tree and works as described in the following pseudocode (Algorithm 3) below.

Algorithm 3 st-numbering

```
1: DFS(s) starts with the edge $(s, t)$
2: for all vertices $v \neq s, t$ in preorder of DFS(s) do
3:     if sign(lowpoint($v$)) == + then
4:         Insert $v$ after $p(v)$ in $L$
5:         sign($p(v)$) = $-$
6:     end if
7:     if sign(lowpoint($v$))==- then
8:         Insert $v$ before $p(v)$ in $L$
9:         sign($p(v)$)=+
10:    end if
11: end for
```

It is clear from the above pseudocode that the procedure runs in linear time using $O(n \lg n)$ bits of space for storing elements in L. To make it space effcient, we use ideas similar to our biconnectivity algorithm. At a high level, we generate the lowpoint values of the first $n/\lg n$ vertices in depth first order and process them. Due to space restriction, we cannot store the list L as in Tarjan's algorithm; instead we use the BP sequence of the DFS tree and augment it with some extra information to 'encode' the final *st*-ordering, as described below.

In the first phase, to obtain the lowpoint values of the first $n/\lg n$ vertices in depth first order, we store explicitly for these vertices their lowpoint values in an array. Also during the execution of the biconnectivity algorithm, the BP sequence is generated and stored in the BP array. We create two more arrays, of size n bits, that have one to one correspondence with the open parentheses of the BP sequence. We can use rank/select operations (as defined Section 3.2) to map the position of a vertex in these two arrays to the corresponding open parenthesis in the BP sequence. The first array, called Sign, is for storing the sign for every vertex as in Tarjan's algorithm. To simulate the effect of the list L, we create the second array, called P, where we store the relative position, i.e., “before” or “after”, of every vertex with respect to its parent. Namely, if u is the parent of v, and v comes before (after, respectively) u in the list L in Algorithm 3, then we store $P[v] = b$ ($P[v] = a$, respectively). One crucial observation is that, even though the list L is dynamic, the relative position of the vertex v does not change with respect to the position of u, and is determined at the time of insertion of v into the list L (new vertices may be added between u and v later). In what follows, we show how to use the BP sequence, and the array P to emulate the effect of list L and produce the *st*-ordering.

We first describe how to reconstruct the list L using the BP sequence and the P array. Note that all the nodes in a subtree appear “together” (consecutively) in the list L. Moreover, all the children marked b appear in the increasing order of the DFI while all the children marked a appear in the decreasing order of the DFI. The main observation we use in the reconstruction of L is that a node v appears in L after all the nodes in its child subtrees whose roots are marked with b in P, and also before all the nodes in its child subtrees whose roots are marked with a in P. Thus by looking at the $P[v]$ values of all the children of a node u, and computing their subtree sizes, we can determine the position in L of u among all the nodes in its subtree. Let us call a child v of u as *after-child* if v is marked a in P. Similarly, if v is marked b in P, it is called *before-child*. Let $T(v)$ denote the subtree rooted at the vertex v in the DFS tree T of G and $|T(v)|$ denotes the size of $T(v)$. Let us also suppose that the vertex u has $k + \ell$ children, out of which k children $v_1, \cdots, v_k$ are before-children and the remaining ℓ children $w_1, \cdots, w_\ell$ are after-children, where $DFI(v_1) < DFI(v_2) < \cdots < DFI(v_k)$ and $DFI(w_1) < DFI(w_2) < \cdots < DFI(w_\ell)$. Then in L, all the vertices from $T(v_1)$, $T(v_2)$, followed by till $T(v_k)$ appear, followed by u and finally the vertices from $T(w_\ell)$, $T(w_{\ell-1})$ till $T(w_1)$ appear. More specifically, u appears at the $(S + 1)$-th location where $S = \sum_{i=1}^{k} |T(v_i)|$. With this approach, we can reconstruct the list L, and hence output the *st*-numbers of all the nodes in linear time, if L can be stored in memory - which requires $O(n \lg n)$ bits. Now to perform this step with $O(n)$ bits, we repeat the whole process of reconstruction $\lg n$ times, where in the i-th iteration, we reproduce sublist $L[(i-1)n/\lg n+1,\ldots,i.n/\lg n]$ – by ignoring any node that falls outside this range – and reporting all these nodes with *st*-numbers in the range $[(i-1)n/\lg n+1,\ldots,i.n/\lg n]$. As each of these reconstruction takes $O(m\lg n\lg\lg n)$ time, we obtain the following,

Theorem 4.20. *Given an undirected biconnected graph G on n vertices and m edges, and two distinct vertices s and t, we can output an st-numbering of all the vertices of G in $O(m\lg^2 n\lg\lg n)$ time, using $O(n)$ bits of space.*

4.5 Applications of *st*-numbering

In this section, we show that using the space efficient implementation of Theorem 4.20 for *st*-numbering, we immediately obtain similar results for a few applications of *st*-numbering.

4.5.1 Two-partitioning problem

In this problem, given vertices $a_1,\cdots,a_k$ of a graph G and natural numbers $c_1,\cdots,c_k$ with $c_1+\cdots+c_k=n$, we want to find a partition of V into sets $V_1,\cdots,V_k$ with $a_i\in V_i$ and $|V_i|=c_i$ for every i such that every set V_i induces a connected graph in G. This problem is called the *k-partitioning problem.* The problem is NP-hard even when $k=2$, G is bipartite and the condition $a_i\in V_i$ is relaxed [61]. But, Györi [93] and Lovasz [104] proved that such a partition always exists if the input graph is k-connected and can be found in polynomial time in such graphs. Let G be 2-connected. Then two-partitioning problem can be solved in the following manner [125]: Let $v_1:=a_1$ and $v_n:=a_2$, compute an v_1v_n-numbering $v_1,v_2,\cdots,v_n$ and note that, from the property of *st*-numbering, for any vertex v_i (in particular for $i=c_1$) the graphs induced by $v_1,\cdots,v_i$ and by $v_i,\ cdots,v_n$ are always connected subgraph of G. Thus applying Theorem 4.20, we obtain the following:

Theorem 4.21. *Given an undirected biconnected graph G, two distinct vertices a_1, a_2, and two natural numbers c_1, c_2 such that $c_1 + c_2 = n$, we can obtain a partition (V_1, V_2) of the vertex set V of G in $O(m \lg^2 n \lg\lg n)$ time, using $O(n)$ bits of space, such that $a_1 \in V_1$ and $a_2 \in V_2$, $|V_1| = c_1$, $|V_2| = c_2$, and both V_1 and V_2 induce connected subgraph on G.*

4.5.2 Vertex-subset-two-partition problem

Wada and Kawaguchi [134] defined the following problem which they call the *vertex-subset-k-partition* problem. This is actually an extension of the *k-partition* problem defined in Section 4.5.1. The problem is defined as follows:

Input:

1. An undirected graph $G = (V, E)$ with n vertices and m edges;
2. a vertex subset V' $(\subseteq V)$ with $n' = |V'| \geq k$;
3. k distinct vertices a_i $(1 \leq i \leq k) \in V', a_i \neq a_j$ $(1 \leq i < j \leq k)$; and
4. k natural numbers $n_1, n_2, \cdots, n_k$ such that $\sum_{i=1}^{k} n_i = n'$.

Output: a partition $V_1 \cup V_2 \cup \cdots \cup V_k$ of the vertex set V and a partition $V_1' \cup V_2' \cup \cdots \cup V_k'$ of vertex set V' such that for each $i (1 \leq i \leq k)$

1. $a_i \in V_i'$;
2. $|V_i'| = n_i$;
3. $V_i' \subseteq V_i$ and
4. each V_i induces a connected subgraph of G.

Note that this problem is an extension of the k-partition problem, since choosing $V' = V$ corresponds to the original k-partition problem. Wada and Kawaguchi [134] proved that vertex-subset-k-partition problem always admits a solution if the input graph G is k-connected (for $k \geq 2$). In particular, if G is 2-connected, using st-ordering, the vertex-subset-two-partitioning problem can be solved in the following manner [134]: suppose that $G, V' \ (\subseteq V), a_1, a_2, n_1$ and n_2 $(n_1 + n_2 = n' = |V'|)$ are the inputs. Let $s = v_1 := a_1$ and $t = v_n := a_2$, compute an st-numbering $v_1, v_2, \cdots, v_n$. From this st-numbering, note that, V now can be partitioned in two sets V_1 and V_2 such that $|V_1 \cap V'| = n_1$ and $|V_2 \cap V'| = n_2$. From the property of st-numbering, we know that both V_1 and V_2 induce a connected subgraph of G. Moreover, $a_1 \in V_1$ and $a_2 \in V_2$. Using Theorem 4.20 as a subroutine to compute such an st-numbering of G, we obtain the following result.

Theorem 4.22. *Given an undirected biconnected graph G, we can solve the vertex-subset-two-partitioning problem in $O(m \lg^2 n \lg\lg n)$ time, using $O(n)$ bits of space.*

4.5.3 Two independent spanning trees

Recall that k spanning trees of a graph G are independent if they all have the same root vertex r, and for every vertex $v \neq r$, the paths from v to r in the k spanning trees are vertex-disjoint (except for their endpoints). Itai and Rodeh [98] conjectured that every k-connected graph contains k independent spanning trees. Even though the most general version of this conjecture has not been proved yet, this conjecture is shown to be true for $k \leq 4$ [48, 55, 98, 135], and also for planar graphs [97]. In particular, if the given graph G is biconnected, we can generate two independent spanning trees (let us call them S and T) in the following manner [98].

Choose an arbitrary edge, say (s, t) in G. Let f be an st-numbering of G. To construct S, choose for every vertex $v \neq s$ an edge (u, v) such that $f(u) < f(v)$, and for t choose an edge other than (s, t). To construct T, choose the edge (s, t) and for every vertex

$v \notin \{s, t\}$ an edge (v, w), $f(v) < f(w)$. We will make s as the root of both S and T. Also S and T are independent spanning trees as, for every vertex v, the path from the root s to v in S consists of vertices u with $f(u) < f(v)$ but except the edge (s, t), whereas in T, along with the edge (s, t), it consists of vertices w with $f(v) < f(w)$. Using Theorem 4.20 to compute such an st-numbering of G, we can produce S and T in $O(m \lg^2 n \lg\lg n)$ time. Thus we obtain the following,

Theorem 4.23. *Given an undirected biconnected graph G, we can report two independent spanning trees S and T in $O(m \lg^2 n \lg\lg n)$ time, using $O(n)$ bits.*

This concludes our space efficient algorithms for DFS based applications. From the next section, we start discussing Maximum cardinality search (MCS) and its applications.

4.6 Maximum Cardinality Search (MCS)

A widely used graph search method which is a restriction of breadth-first search (BFS) is lexicographic BFS (Lex-BFS), introduced by Rose et al. [122] under the name Lex-P. They used Lex-BFS to find a perfect elimination ordering (PEO) of the vertices of a graph G if G is chordal. A perfect elimination ordering in G is an ordering of the vertices such that, for each vertex v, v and the neighbors of v that occur after v in the order form a clique. Fulkerson et al. [84] showed that a graph is chordal if and only if it has a perfect elimination ordering. Thus to recognize a chordal graph, we can run the Lex-BFS algorithm, and test whether the resulting order is a perfect elimination order. Rose et al. [122] showed both the tasks of performing Lex-BFS of G and testing if the resulting order is PEO can be done in $O(m+n)$ time. Even though the Lex-BFS runs in linear time, its implementation is a bit involved, and it takes $O(m + n)$ words of space [122]. Later Tarjan, in an unpublished note [127], derived another simpler and alternate graph search method for finding a PEO of chordal graphs, known as Maximum Cardinality Search (MCS). Tarjan and Yannakakis [131] presented MCS and its applications to recognize chordal graphs and test acyclicity of hypergraphs. We provide space efficient algorithms for MCS and its many applications in chordal graphs.

4.6.1 The MCS algorithm and its implementation

We start by briefly describing the MCS algorithm and its implementation as provided in [131]. The output of the MCS algorithm is a numbering of the vertices from 1 to n. During the execution of the algorithm, vertices are chosen by choosing an unnumbered vertex that is adjacent to the maximum numbered vertices. We provide the pseudocode below for completeness along with an illustration of the MCS algorithm in a given graph.

Algorithm 4 Maximum Cardinality Search (MCS)

```
   Input: a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$.
   Output: an ordering $\sigma$ of $V$.
1: assign label 0 to all the vertices
2: for $i \leftarrow 1$ to $n$ do
3:     pick an unnumbered vertex $v$ with maximum label
4:     $\sigma(i) \leftarrow v$                    ▷ This assigns to $v \in V$ the number $i$
5:     for each unnumbered vertex $w$ adjacent to $v$ do
6:         $label(w) \leftarrow label(w) + 1$
7:     end for
8: end for
```

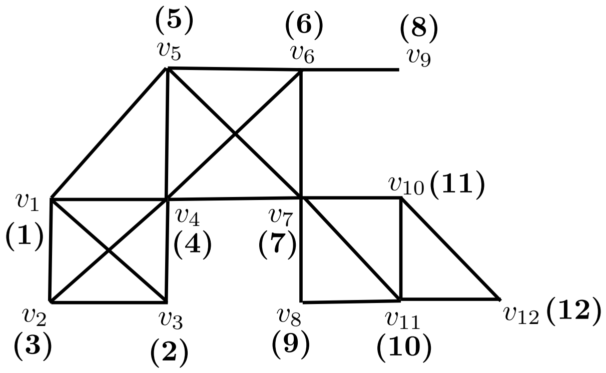

Figure 4.4: An input graph G with n vertices from v_1 to v_n. The numbers in the brackets show one possible output of the MCS algorithm on G when it starts with the vertex v_1. The resulting vertex ordering obtained is $v_1, v_3, v_2, v_4, v_5, v_6, v_7, v_9, v_8, v_{11}, v_{10}, v_{12}$.

Tarjan et al. [131] gave a $O(m + n)$ time implementation of MCS. Even though they did not explicitly analyse the space requirement of their algorithm, we show that it takes $O(n)$ words of space. This exact space bound is particularly interesting and worth mentioning since many of the subsequent papers actually cite this version of MCS result saying, the implementation of Tarjan et al. [131] takes $O(m+n)$ time and words of space. For example see Theorem 7.1 of [102] and Theorem 5.2 of [22]. In the rest of this section, we briefly describe the original algorithm of Tarjan et al. [131] and its time and space complexities.

The MCS algorithm of Tarjan et al. maintains an array of sets $set[i]$ for $0 \leq i \leq n-1$ where $set[i]$ stores all unnumbered vertices adjacent to exactly i numbered vertices. So, at the beginning all the vertices belong to $set[0]$. They also maintain the largest index j such that $set[j]$ is non-empty. To implement an iteration of the MCS algorithm, they remove a vertex v from $set[j]$ and number it. For each unnumbered vertex w adjacent to v, w is moved from the set containing it, say $set[i]$, to $set[i+1]$. If there is a new entry in $(j+1)$, we move to $set[j+1]$ and repeat the same. Otherwise when $set[j]$ becomes empty, they repeatedly decrement j till a non-empty set is found and in this set we repeat the same procedure. In order to delete easily, they implement each set as a doubly-linked list. In addition, for every vertex, they store the index of the set containing it. This completes the description of the implementation level details of MCS as provided by Tarjan et al. [131].

Now, observe that in the above implementation, when the vertex w needs to be moved from $set[i]$ to $set[i+1]$, we just know the index of $set[i]$ that w belongs to, but not w's location inside $set[i]$. To get overall linear time, we cannot afford to perform a linear search for w in $set[i]$ as this might be a costly operation. A simple way to fix this is to, instead of storing for every vertex v the set index where it belongs to, store a pair (i, j) if a vertex v belongs to the list $set[i]$ and j is the pointer to v's location inside $set[i]$. Then we can directly access v and move it to $set[i+1]$ from $set[i]$ in constant time. This concludes the description of our modified implementation. It is easy to see that every vertex appears only once in any of these sets, so array *set* takes at most $2n$ words in worst case. Clearly the running time is $O(m+n)$ and space required is $O(n)$ words. In the next section we provide space efficient implementations for MCS.

4.7 Space efficient implementations of MCS

Algorithm 1: Using $n+O(\lg n)$ bits. Here we show using just $n+O(\lg n)$ bits, albeit with increased time, we can perform MCS. Towards that we maintain a bit vector B (initialized with all 0 entries) of size n bits where the $B[i]$-th entry is 1 if and only if the vertex v_i has already been numbered. The algorithm works as follows: at each step it scans the whole B array to find all the zero entries and for each one of them, it goes over the adjacency array to find out how many of its neighbors are already marked '1'. Then the vertex v which has the maximum number of numbered neighbors (ties are broken arbitrarily) is marked '1' in B. We repeat this step until all the vertices are marked. This procedure uses $n+O(\lg n)$ bits. At each step, the algorithm spends $O(m)$ time to find a vertex to number, and this happens exactly n times. Hence, the running time is $O(mn)$.

Algorithm 2: Using $O(n)$ bits. By increasing the space bound by a constant factor, we can design a faster algorithm, by using similar ideas as in Tarjan et al.'s [131] algorithm with a few changes. We define the *label* of an unnumbered vertex (at any instance of the algorithm) as its number of numbered neighbors. The main idea is to maintain a doubly linked list, call it L, of size at most $n/\lg n$ at any point during the execution of the algorithm. Each element in L stores a label k and a pointer to a *sublist*. The sublist labeled k stores a set of vertices with label k, and is itself maintained as a doubly linked list. The sublists in L are stored in the increasing order of their labels. Moreover, the sum of the sizes of all the sublists, at any time, is at most $n/\lg n$. Also, we maintain all the vertices that are currently stored in L in a balanced binary search tree, T, and store

pointers from the nodes of T to the corresponding vertices in L.

At the beginning of the algorithm, we add $n/\lg n$ vertices into the tree T and the same vertices also into the list L in a sublist labeled 0 (with pointers to the corresponding vertices from T to L). As before, we maintain a bit vector B of length n to keep track of the numbered vertices. The i-th step of the algorithm, for $1 \leq i \leq n$, proceeds as follows. We select the first (arbitrary) element of the sublist with the largest label in L. Let v be the vertex stored in this element. Then, we first number the vertex v with i and delete v from both L and T. We then go through each unnumbered neighbor w of v and compute the label k of w. If the new label k is greater than the label of the sublist with the smallest label, then we add w to L in the sublist labeled k; delete the first (arbitrary) element from the sublist with the smallest label, if necessary (to maintain the invariant that L has at most $n/\lg n$ elements); and also add (and delete) the corresponding vertices to the tree T. (Note that we can also add w to L if k is equal to the label of the sublist with the smallest label as long as the number of elements in L is at most $n/\lg n$; but this does not change the worst-case running time of the algorithm.) Also, if w is already stored in L (in the sublist labeled $k-1$), then we move w from the sublist labeled $k-1$ to the sublist labeled k.

Note that, after a while, due to deletion of vertices, the list L may become empty. At that time, we refill the list L with $n/\lg n$ unnumbered vertices with the highest labels (or refill with all the remaining vertices if there are fewer than $n/\lg n$ unnumbered vertices). This is done by scanning the bit vector B from left to right, and for each unnumbered vertex v, computing its label. The first $n/\lg n$ unnumbered vertices are simply inserted into the appropriate sublists in L. For the remaining unnumbered vertices, if the label is greater than the label of the smallest labeled sublist currently in L, then we insert the new vertex into the appropriate sublist, and delete the first vertex from the sublist with the smallest label. The cost of this refilling is dominated by the cost of computing the labels of the vertices which is $O(m)$. After constructing the list L, we insert each of the vertices in L into an initially empty binary search tree T and add pointers from the nodes in T to the corresponding elements in L, which takes $O(n)$ time. The refilling of the list L happens at most $O(\lg n)$ times since at least $n/\lg n$ vertices will be numbered after each refilling, and hence over the full execution of the algorithm it takes $O(m \lg n)$ time. Computing the label of a vertex v takes $O(d_v)$ time, where d_v is the degree of v. During the execution of the algorithm, we need to compute the label of a vertex v at most $O(d_v)$ times (every time one of its neighbors is numbered). Thus, running time of computing the labels of vertices is bounded by, $O(\sum_{v \in V} d_v^2) = O(m^2/n)$. Finally, moving an element from one sublist to another takes $O(\lg n)$ time since we need to search for it in the binary search tree first, before moving it. Since a vertex v is moved at most d_v times, this step contributes $O(\sum_{v \in V} d_v \lg n) = O(m \lg n)$ time to the total running time. Thus overall the algorithm takes $O(m^2/n + m \lg n)$ time.

Algorithm 3: Using $O(n \lg \frac{m}{n})$ bits. We show that using $O(n \lg \frac{m}{n})$ bits we can design a significantly faster algorithm. Note that for sparse graphs ($m = O(n)$), this space is only $O(n)$ bits. We first scan the adjacency list of each vertex and construct a bitvector D as follows (as we have done in Section 3.6): starting with an empty bitvector D, for $1 \leq i \leq n$, if d_i is the degree of the vertex v_i, then we append the string $0^{\lceil \lg d_i \rceil - 1}1$ to D. The length of D is $\sum_{i=1}^{n} \lceil \lg d_i \rceil$, which is bounded by $O(n \lg(m/n))$. We also construct auxiliary structures to support *select* queries on D in constant time [109]. Like the previous algorithm, we also maintain the current top $O(n/\lg n)$ values in the list of doubly linked list L along with all other auxiliary information. Finally, we maintain in a bitmap B marking all the already numbered and output vertices. Overall space usage is $O(n \lg \frac{m}{n})$ bits.

The algorithm is essentially same as earlier. The only difference is that, using the structure D, we can compute and update (by doing word level read and write) the labels of vertices in $O(1)$ time (instead of $O(d_v)$ time as in the earlier algorithm). Thus the running time of computing the labels of the vertices is now bounded by $O(m)$, and the rest of the computations is same as earlier. Hence the overall running time is $O(m \lg n)$. The three algorithms described above can be summarized as follows.

Theorem 4.24. *Given a graph G, we can obtain an MCS ordering of G in (a) $O(mn)$ time using $n+O(\lg n)$ bits, or (b) $O(m^2/n+m\lg n)$ time using $O(n)$ bits, or (c) $O(m\lg n)$ time using $O(n\lg(m/n))$ bits.*

4.8 Applications of MCS

In this section, we provide several applications of our MCS algorithms presented in the previous section. We start with some surprising connection of our MCS algorithm with a few other totally unrelated graph problems and their algorithms with respect to designing space efficient algorithms. Next we show how to use MCS to provide space efficient solutions for solving some combinatorial problems in chordal graphs, and also recognition of chordal graphs. We start by describing the connection with MCS first.

4.8.1 Connection with other problems

In this section, we discuss two other seemingly different problems and their algorithms which will be similar to the space efficient MCS algorithms. First is the problem of topologically sorting the vertices of a directed acyclic graph, and the second is that of finding the degeneracy of an undirected graph. The similarity of these two problems with MCS comes from the fact that the linear time algorithms for all three of these problems have a very natural greedy strategy. We start by briefly explaining the linear time greedy algorithms for these two problems.

One of the algorithms for topological sort works by maintaining the in-degree count of every vertex, and at each step, it deletes a vertex of in-degree zero and all of its outgoing edges. The order in which the vertices are deleted gives the topological sorted order. To efficiently implement the algorithm, all the vertices currently having in-degree zero are stored in a queue. This algorithm can also test if the given graph is acyclic or not at the same time. If it is acyclic, it produces a topological sort. Note the similairy of this algorithm with that of MCS. It is not hard to see that each of the solutions for MCS explained before could be made to work for topological sort with similar resource bounds.

The degeneracy of a graph G is the smallest value d such that every nonempty subgraph of G contains a vertex of degree at most d. Such graphs have a degeneracy ordering, i.e., an ordering in which each vertex has d or fewer neighbors that come later in the ordering. Degeneracy and degeneracy ordering, can be computed by a simple greedy strategy of repeatedly removing a vertex with smallest degree (and its incident edges) from the graph until it is empty. The degeneracy is the maximum of the degrees of the vertices at the time they are removed from the graph, and the degeneracy ordering is the order in which vertices are removed from the graph. The linear time implementation of this works almost in the same way as the MCS algorithm [20]. One can implement this algorithm space efficiently using similar ideas as that of MCS. We omit the relatively easy details. It would be intersting to find other problems with similar flavour. We want to conclude this section by remarking that, for any greedy algorithm of this flavour we can use similar technique to design space efficient implementation. We summarize our results in the theorem below.

Theorem 4.25. *Given a directed graph G, we can report whether G is acyclic or not, and if so, we can produce a topological sort ordering in (a) $O(mn)$ time using $n+O(\lg n)$ bits, or (b) $O(m^2/n + m \lg n)$ time using $O(n)$ bits, or (c) $O(m \lg n)$ time using $O(n \lg(m/n))$ bits. Using the same running time and space bounds, we can also test if an undirected graph G is d-degenerate, and if so, we can output a d-degenerate ordering of G.*

It is worth mentioning that in Theorem 4.4, we showed that using just $O(n)$ bits, topological sort of a directed acyclic graph G can be performed in $O(m \lg\lg n \lg n)$ time.

This contrasts with the above theorem in the fact that the algorithm of Theorem 4.4 heavily relies on that fact that in the input representation both the in as well as out adjacency lists are provided for every vertex v, and this facilitates the improved running time of their algorithm. Here, instead, we work with more traditional and commonly used out-adjacency list in our $O(n)$ bits algorithm. We don't know if we can achieve the bounds of Theorem 4.4 without the in and out adjacency list assumption.

4.8.2 Finding Independent set, Vertex cover and Proper coloring

In this section, we show that using Theorem 4.24 how one can solve some combinatorial problems on chordal graphs space efficiently. Recall that a perfect elimination ordering (PEO) in G is an ordering of the vertices such that, for each vertex v, v and the neighbors of v that occur after v in the order form a clique. Tarjan et al. [131] showed that, if G is a chordal graph and σ is an MCS ordering of G, then the reverse of σ is a PEO of G. More specifically, given the graph G, and a vertex ordering σ of G, we define the following:

- The edge directions implied by σ is obtained by directing (v_i, v_j) as $v_i \rightarrow v_j$ if $i < j$ and $v_j \rightarrow v_i$ if $i > j$.
- If $v_i \rightarrow v_j$ is an edge implied by the order, then v_i is the predecessor of v_j and v_j is a successor of v_i.

Thus the theorem of Tarjan et al. [131] says that, the predecessor set of every vertex (i.e., set containing all the predecessors) in σ forms a clique, or equivalently in the reverse of σ, for each vertex v, v and its successor set form a clique if and only if G is chordal. To solve some combinatorial problems in chordal graphs, we need the reverse PEO whereas for some applications we need the PEO. For all applications where we need a PEO, we spend extra time for reversing the list produced by our MCS algorithm as, due to space restriction, we cannot store the PEO and reverse it in linear time. This seems to be a fundamental bottleneck while designing space efficient algorithms.

Maximum Independent Set: We start with the problem of finding a maximum independent set (MIS), and for this, a simple greedy strategy works [89]. Given a reverse PEO of the input chordal graph G, the algorithm scans the vertices in order, and for every vertex v_i, it adds v_i to the solution set I if none of its predecessors has been added to I already. Note that using a bitmap S of size n bits, where $S[i]$ is set if the vertex v_i belongs to I, on top of the structures of Theorem 4.24, we can easily implement this to find an MIS with no extra time. More specifically, at any iteration of the algorithm, while we arrive at a vertex v_i i.e., at that instance vertex v_i is generated in the reverse PEO, we scan all the neighbors of v_i and add v_i to the solution set I if none of its predecessors has been added to I already. Also the complement of the set S gives us a minimum vertex cover for G. Thus,

Theorem 4.26. *Given a chordal graph G, we can output a maximum independent set and/or a minimum vertex cover of G in (a) $O(mn)$ time using $2n + O(\lg n)$ bits, or (b) $O(m^2/n + m \lg n)$ time using $O(n)$ bits, or (c) $O(m + n \lg n)$ time using $O(n \lg(m/n))$ bits.*

Proper coloring: In what follows, we discuss a space efficient implementation of finding a proper coloring of a chordal graph G. It is known that the natural greedy algorithm [89] for coloring yields the optimal number of colors if andonly if the vertex order is a PEO. The algorithm works as follows: given a PEO, it scans the vertices in order, and colors each vertex with the smallest color not used among its predecessors. Note that, we need a PEO here, but MCS produces a reverse one, thus we first need to reverse the list produced by the MCS algorithm. Also, it is easy to see that this coloring scheme assigns, for any vertex v, the color at most $max_i\{indeg(v_i) + 1\}$ where $indeg(v_i)$ refers to the neighbors of v_i which appeared before in PEO and have already been colored.

Suppose we store the explicit colors for each vertex in an array B, then the length of B is $\sum_{i=1}^{n} \lceil \lg d_i + 1 \rceil = O(n \lg(m/n))$. We construct auxiliary structures to support *select* queries on B in constant time [109]. Suppose we have $O(n \lg(m/n))$ bits at our disposal, then we run our MCS algorithm and store the last chunk of $O(n/ \lg_{m/n} n)$ vertices in a queue Q. Now dequeue vertices from Q one by one and run the greedy coloring algorithm while storing the colors in B array explicitly, and continue. Once Q becomes empty, run the MCS algorithm to generate the previous chunk and store them in Q, and repeat the greedy coloring algorithm afterwards. This process is repeated at most $O(\lg_{m/n} n)$ times, and each time we run the MCS algorithm followed by the greedy coloring scheme, hence total running time is $O((m + n \lg n) \lg_{m/n} n)$. We summarize our result below.

Theorem 4.27. *Given a chordal graph G, we can output a proper coloring of G in $O((m + n \lg n) \lg_{m/n} n)$ time using $O(n \lg(m/n))$ bits.*

Note that with $O(n)$ bits, we cannot store the colors of all the vertices simultaneosuly, and this poses a challenge for the greedy algorithm. We leave open the problem to find a proper coloring of chordal graphs using $O(n)$ bits.

4.8.3 Recognition of chordal graphs

In what follows, we present a space efficient implementation for the recognition of chordal graphs. The idea is to apply MCS on G first to generate a vertex ordering, and then check whether the resulting vertex ordering is indeed a PEO. Let $v_1, v_2, \ldots, v_n$ be the ordering of the vertices reported by the MCS algorithm.

Chordal graph recongnition with $O(n \lg(m/n))$ bits: We first observe that one can compute the predecessor/successor of any vertex v_i during the MCS algorithm. Since we have $O(n \lg(m/n))$ bits, we can store one pointer into its adjacency list with each vertex. We use this space to store a pointer to the last predecessor for each vertex. To test whether G is chordal, we need to test, for each vertex v_i, whether the neighbors of v_i numbered less than i form a clique. To perform this test, it is enough to test whether the predecessor set of v_i is a subset of the predecessor set of v_j where v_j is the last predecessor of v_i [89].

Now we run the MCS algorithm once again, and whenever we number a vertex v_j, we also generate its predecessor set P_j. For this purpose, we maintain a bit vector B to mark all the vertices that are numbered during the current MCS algorithm. Using this (dynamic) bit vector B, one can easily generate the set P_j by simply scanning the adjacency list of v_j and checking whether they are marked in B. This set P_j is stored as a bit vector Q of length n such that $Q[\ell] = 1$ iff $v_\ell \in P_j$. This helps us in checking the membership of a vertex in the set P_j in $O(1)$ time. The bit vector Q is initialized in $O(n)$ time at the beginning of the algorithm, and is used to store the predecessor set of the currently numbered vertex v_j, for $1 \leq j \leq n$. After generating the predecessor set P_j of v_j, we scan the adjacency list of v_j to check for any vertex v_i, where $i > j$, if v_j is the last predecessor of v_i (given v_i and v_j, we can check whether v_j is the last predecessor of v_i in constant time using the predecessor pointer of v_i). If v_j is the last predecessor of v_j, then we test whether the predecessor set of v_i, excluding v_j, is a subset of P_j. Note that since v_j is the last predecessor of v_i, all the other predecessors of v_i must have already been numbered when v_j is numbered, and hence are stored in the set represented by the bit vector B. Thus we can test whether the predecessor set of v_i is a subset of P_j in time proportional to the degree of v_i (with the aid of the bit vector Q). This completes the description of our algorithm for chordal graph recongnition.

The overall runtime of the algorithm is dominated by the runtime of the MCS algorithm. Thus if we can perform MCS on a graph G in $t(n, m)$ time using $s(m, n)$ bits, then we can also check whether G is chordal in $O(t(n, m))$ time using $s(m, n) + O(n \lg(m/n))$ bits.

Chordal graph recongnition with $O(n)$ bits: Since we cannot store the predecessors of all the n vertices of G (in the MCS order), we generate these vertices in $\lg n$ phases, where in the ℓ-th phase, we generate the vertices $V_\ell = \{v_{\ell k+1}, v_{\ell k+2}, \ldots v_{\ell(k+1)}\}$, for $1 \leq \ell \leq \lg$ and $k = n/\lg n$. In each phase, for each vertex v_i generated, we test the condition whether predecessor set of v_i is a subset of the predecessor set of v_j where v_j is the last predecessor v_i. To do this, we first perform another MCS, to compute the last predecessor of each vertex in V_ℓ (in fact, this step can be combined with the step where we generate the set V_ℓ). We maintain a bit vector to mark the set of last predecessors of elements in V_ℓ to check their membership in $O(1)$ time, and another bit vector B to mark the numbered vertices. Now we start another MCS, and whenever a node v_j is numbered, we check to see if it is the last predecessor of some node v_i in V_ℓ. If so, we generate the predecessor sets of v_j and v_i with the aid of B, and check the required condition.

Thus if we can perform MCS on a graph G in $t(n, m)$ time using $s(m, n)$ bits, then we can also check whether G is chardal in $O(t(n, m) \cdot \lg n)$ time using $s(m, n) + O(n)$ bits. Thus we obtain the following,

Theorem 4.28. *Given an undirected graph G, if MCS on G can be performed in $t(n, m)$ time using $s(m, n)$ bits, then chordality of G can be tested in (a) $O(t(n, m))$ time using $s(m, n) + O(n \lg(m/n))$ bits, or (b) $O(t(n, m) \cdot \lg n)$ time using $s(m, n) + O(n)$ bits.*

Corollary: Given an undirected graph G with n vertices and m edges, we can test if G is chordal using $O(m \lg n)$ time and $O(n \lg(m/n))$ bits of space or $O((m^2 \lg n)/n + m \lg^2 n)$ time using $O(n)$ bits of space.

4.9 Concluding remarks and open problems

We have presented space efficient algorithms for a number of important applications of DFS. Obtaining linear time algorithms for them while maintaining $O(n)$ bits of space usage is both interesting and challenging open problem. One of the main bottlenecks (with this approach) towards this is finding an $O(n)$-bit, $O(m + n)$-time algorithm for DFS, which is also open, even though for BFS we know such implementations [14, 69]. Another challenging open problem is to remove the additional poly-log terms in the running times of the algorithms described (e.g., the $\lg n$ term in the running time of 2-vertex and 2-edge connectivity algorithm, and the $\lg^2 n$ term in the running time of two independent spanning trees algorithm). These terms seem inherent in our tree covering approach. It would be interesting to find other applications of our tree covering approach for space efficient algorithms. Planarity is an example where DFS has been used very crucially. So it is a natural question that, can we test planarity of a given graph using $O(n)$ bits? Recently Kammer [99] et al. provided an algorithm to recognize outerplanar graphs using $O(n)$ bits.

We also showed several time-space tradeoffs for performing MCS and provided space efficient implementation for its applications including testing if a given undirected graph is chordal, reporting an maximum independent set, and a proper coloring of a given chordal graph. One very challenging open problem would be to obtain a sublinear space implementation for any of these methods (DFS and MCS). Note that DFS is known to be P-complete, hence we don't expect to have a poly-log space algorithm for DFS as that would collapse the class L (deterministic logspace) and P (polynomial time). But can we even design $O(\sqrt{n})$ space and polynomial time algorithm for DFS? Even though no such complexity theoretic results are known for MCS, it still seems hard to obtain sublinear space algorithm for MCS as well. We leave that also as another challenging open problem.

Chapter 5

Space Efficient Algorithms for Optimization Problems in Bounded Treewidth Graphs

5.1 Introduction

So far we have seen a variety of space efficient graph algorithms but for none of them we could achieve sublinear space algorithms. In this chapter, we show that for a large class of optimization problems in bounded treewidth graphs, we could actually design such sublinear space algorithms without loosing too much on time. In fact our result is more general i.e., we provide a smooth time/space tradeoff result for these problems.

More specifically, it is well-known by the famous Courcelle's theorem that many graph properties (those that are expressible in monadic second order logic) can be solved in linear time on graphs of bounded treewidth. Logspace versions of this using automata theoretic framework are also known. In this chapter, we develop an alternate methodology using the standard table-based dynamic programming approach to give a space efficient version of Courcelle's theorem. We assume that the given graph and its tree decomposition are given in a read-only memory. Our algorithms use the recently developed stack-compression machinery of Barba et al. [18] and the classical framework of Borie et al. [28] to develop time-space tradeoffs for dynamic programming algorithms that use $O(p \lg_p n)$ variables where $2 \leq p \leq n$ is a parameter. We believe that our approach is more natural and simpler. En route we also slightly generalize the stack compression framework to a broader class of algorithms, which we believe can be of independent interest.

5.1.1 Motivation

The aim of this chapter is to demonstrate the power of two class of algorithms – one classical [28] and one recent [18] – and to show that a combination of them can provide extremely space efficient algorithms for graphs of bounded treewidth even when the tree of the decomposition is given in a read-only memory.

It is well-known [26] that many problems that are NP-hard on general graphs can be solved in linear time, using dynamic programming, on trees and graphs of bounded treewidth. This is captured by the most general theorem due to Courcelle [53] which states that properties that can be expressed in Counting Monadic Second Order Logic (CMSO) can be tested in linear time on graphs of bounded treewidth. Aspvall et al. [13] considers a closely related problem. Given the tree T of tree decomposition, they look at the problem of finding a root and traversal order to minimize the space requirement i.e. the minimum number of tables that need to stored simultaneously in the main memory. They showed how to find such a root, and also proved that dfs order traversal of T actually minimizes the space requirement. More specifically, they showed that, the space requirement is lower bounded by the pathwidth of the tree of the tree decomposition T plus one, and upper bounded by twice the pathwidth of T. In particular, as the pathwidth of a tree on n nodes is $O(\lg n)$, this implies that $O(\lg n)$ tables are sufficient. Not just the optimum value, Bodlaender and Telle [27] later show that the sets that realize the optimum solution can be determined in $O(n \lg_s n)$ time using $O(s + w)$ words of space where $s \leq \sqrt{cn/2}$ is a parameter, w is the pathwidth of T and c is the maximum number of children of any node in the tree of the tree decomposition.

Our main objective here is to implement these standard dynamic programming algorithms in the read-only memory model where the input tree is given in a read-only input. Apart from adjacency of the graph, we can only access the leftmost child, right sibling and the bags of each node in constant time (in particular, we do not assume parent pointers which poses a challenge when doing a bottom-up traversal). We show that the above bounds (that of Aspvall et al. [13]) can be achieved if we use an additional stack that could grow to linear size. We then use the recent stack compresion method of Barba et al. [18] to reduce the space to $O(\lg n)$ without too much degradation in time. Towards this end, we first generalize the stack-compression method to a broader class of stack algorithms.

5.1.2 Related Work

Elberfeld et al. [64] showed that Bodlaender's [24] linear time algorithm (to determine whether a given graph has treewidth at most k for a fixed k) and Courcelle's [53] linear time algorithm (to determine the satisfiability of a CMSO expressible property on a bounded treewidth graph) can be implemented in $O(1)$ words, i.e., using $O(\lg n)$ bits of space. However, the algorithm of Elberfeld et al. [64] is reasonably complicated, and is based on the automata theoretic framework of Courcelle's theorem, while we use the natural table based approach of the dynamic programming algorithms. Elberfeld et.al [64] does not focus on time needed to implement their algorithm and, similarly, Courcelle's [53] linear time algorithm does not consider space issues. We, in this chapter, give a smooth tradeoff between time and space for such dynamic programming algorithms.

The Maximum Independent Set problem covered by our main result in this chapter, Bhattacharya et. al [23] gave an $O(n)$ algorithm using $O(h)$ word space; here h is the height of the tree that could be as high as n. In contrast, our algorithm gives the optimum value in $O(n)$ time and $O(n^{\epsilon})$ words of space even for weighted trees and bounded treewidth graphs. It can also output the set realizing the optimum value in the same time and space for unweighted trees. But for weighted trees and bounded treewidth graphs, outputting the solution set takes $O(n^2)$ time using the same space. We also give algorithms that take $O(\lg n)$ words of space and $O(n^{1+\epsilon})$ time to output the optimum value. Here ϵ is any fixed positive constant less than 1.

Our results, apart from adding to the growing body of literature on space efficient graph algorithms, bring to light two important results for space efficient implementation of dynamic programming algorithms:

- the classical observation (doesn't seem to be well-known) due to Aspvall et.al. [13] that a careful post order traversal of the tree decomposition requires only $O(\lg n)$ open tables for dynamic programming based algorithms for bounded tree-width graphs, and
- the recent technique to reduce the space requirement of algorithms that use only a stack (that could grow to linear size) and a constant number of auxiliary words.

Like previous chapters, here also we work with the *register input model.* Our algorithms work with the standard left-most child, right-sibling based representation for trees [51]. I.e., we assume that the tree (of the input or of the tree-decomposition) is given in a representation where the children of a node are organized in a linked list. I.e. given a node label, we can find its left-most child and its right sibling in constant time. In particular, we do not have parent pointers associated with the nodes, unless stated otherwise. For the weighted versions of the algorithms and for graphs of bounded treewidth, we assume that weights or the bag sets of the vertices are given in a separate array indexed by the labels of the vertices of the tree (which we assume are in $\{1, 2, \ldots n\}$). For bounded treewidth graphs, apart from the tree decomposition, we also need the graph in read-only memory, represented in a way that adjacency can be checked in constant time.

5.1.3 Problem Definitions

In this subsection, we define some of the optimization problems we solve space efficiently on trees and bounded treewidth graphs. Given a graph G, a subset $S \subseteq V$ is called a *vertex cover* if for all $e = (i, j) \in E$, S contains i or j or both. Equivalently $G[V - S]$ is an independent set i.e., a set of vertices in a graph, no two of which are adjacent.

MINIMUM VERTEX COVER

Input: A graph $G = (V, E)$.

Question: Find the cardinality of a minimum sized vertex cover and a set that realizes this value.

Similarly, we define MAXIMUM INDEPENDENT SET.

MAXIMUM INDEPENDENT SET

Input: A graph $G = (V, E)$.

Question: Find the cardinality of a maximum sized independent set and a set that realizes this value.

Given a graph $G = (V, E)$, a matching M in G is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex.

MAXIMUM MATCHING

Input: A graph $G = (V, E)$.

Question: Find the cardinality of a maximum sized matching and a set that realizes this value.

Given a graph $G = (V, E)$, a subset $S \subseteq V$ is called a *dominating set* if every vertex in the subset $V \setminus S$ is adjacent to at least one vertex in S.

Minimum Dominating Set
Input: A graph $G = (V, E)$.
Question: Find the cardinality of a minimum sized dominating set and a set that realizes this value.

For the weighted versions of these problems (except Maximum Matching), we are given a non-negative integer weight on each vertex of the input graph, and the objective is to find the weight of a minimum/maximum weighted set and a set that realizes this minimum/maximum weight.

5.1.4 Organization of the chapter

In Section 5.2, we first explain the general working scheme of stack based algorithms and also state the main lemma from [18]. Then we provide our generalization of the stack based algorithms, and prove our stack compression lemma for the generalization. We begin the development of our dynamic programming based algorithms on the simple problem of performing a depth-first search (DFS) of a tree in Section 5.3. The algorithm contains the main ideas that are used later to develop dynamic programming algorithms on bounded treewidth graphs. Then in Section 5.4 we provide the dynamic programming algorithms for the specific optimization problems on trees using the stack compression machinery, as a warm up for our general result. This section also considers different variants of these problems on trees. In Section 5.5 we develop our main space efficient algorithm for problems expressible in counting monadic second order logic (CMSO). This section also considers the weighted versions and the modifications necessary to output the solution set. Section 5.6 contains some concluding remarks.

5.2 Generalized Stack Framework

In what follows, we explain the general scheme of stack based algorithms defined in [18]. As we generalize the scheme slightly, we find it useful for the readers to reproduce the framework and the main lemma with proof sketches which we use to argue about our generalized framework.

Let $\mathcal{A}$ be a class of deterministic algorithms which uses a stack and optionally other auxiliary data structures of constant size. The operations that can be supported are push, pop and accessing the k top most elements of the stack for a constant k.

Let $X \in A$ be any algorithm and $a_1, a_2, \cdots, a_n \in I$ be the values of the input, in the order in which they are treated by A. We only assume that given a_i, we can access the next input element in constant time. We will call X a stack based algorithm if, given a set of ordered input I, X processes $a_i \in I$ one by one in order and based on a_i, the top k elements (for some constant k) of the stack and the auxiliary data structures' current configuration, it decides to pop some elements off the stack first, then push a_i (or some function of a_i) along with some constant words of information into the stack. Then the computation moves forward to the next element, until all the elements are exhausted. The final result which is stored in the stack, is then output by popping them one by one until the stack is empty. Any algorithm following this structure is called a *stack based algorithm.* We provide the pseudocode in the next page for completeness [18].

Algorithm 5 Basic scheme of a stack based algorithm

```
1: Initialize stack and auxiliary data structure DS with $O(1)$ elements from $I$
2: **for all** input $a \in I$ in order **do**
3:     **while** some-condition(a, DS, STACK.TOP(1),$\cdots$, STACK.TOP($k$)) **do**
4:         STACK.POP
5:     **end while**
6:     **if** another-condition(a, DS, STACK.TOP(1),$\cdots$, STACK.TOP($k$)) **then**
7:         STACK.PUSH(a or some function of a)
8:     **end if**
9: **end for**
10: Report(STACK)
```

For such stack based algorithms, in [18] the authors prove the following result. We give the proof for completeness as it is helpful for understanding our generalized framework.

Theorem 5.1 ([18])**.** *Any stack algorithm which takes $O(n)$ time and $\Theta(n)$ space can be adapted so that, for any parameter $2 \leq p \leq n$, it solves the problem in $O(n^{1+(1/\lg p)})$ time using $O(p \lg_p n)$ variables.*

Proof. We partition the input into p blocks for a given $2 \leq p \leq n$ (assume for ease of presentation that n is divisible by p). Thus each block has a size of n/p. We then compress each block except the top two non-empty blocks in the stack. Compression involves storing only the first and last elements of the blocks which are in the stack and are denoted by a_b and a_t respectively for each block. Along with a_b we store an auxiliary data structure of size $O(1)$ to help reconstruct the block elements when required. The top two non empty blocks are partially compressed level by level. This means that we recursively subdivide them into p sub-blocks till level $h = \lg_p n - 1$ when we explicitly store the blocks. Note that at level h the explicitly stored blocks have p elements stored. Let us denote by F_i and S_i the first two blocks at level i. We also say that F_i may be empty but S_i is always non-empty.

Lemma 5.2. *[18] The compressed stack structure uses $O(p \lg_p n)$ words space.*

Proof. At each level both the top two partially compressed blocks and the totally compressed blocks below them take $O(p)$ space each. As there are $h = \lg_p n$ levels the space bound follows. We can look at it as storing h different stacks each of $O(p)$ space. □

Let us now see how to implement the *Pop* and *Push* operations such that the compressed stack structure is maintained.

Push Operation : We push an element at level h. If the block F_h has space left, then we add the element to the block and *update* all the levels below it. By *update*, we refer to changing the a_t values of the blocks F_i at each level i to the newly pushed element. It may also happen that F_h or some lower level F_i has no space left and we need to create a new block F_i for the pushed element. In this case the old F_i becomes the new S_i and we compress the old S_i noting that the compressed S_i is already available at a level above. So we need to just query level $i - 1$ for the compressed S_i. In this case we need to also push in the element along with the auxiliary data structure. Thus all push operations can be performed in constant time per level.

Pop Operation : We pop an element from level h. If the block F_h or S_h has elements left (note that F_h may be empty), in that case we again change the a_t values of all the levels below for the corresponding F_i or S_i. Now if the block F_h becomes empty then we propagate down the levels until we find a non-empty F_i (or S_i if F_i is empty). For this non empty F_i or S_i both a_b and a_t are known and we need to reconstruct this block.

Reconstruction : For reconstructing a block, say B, we have the a_b value stored along with the context. The reconstruction of the block mimics algorithm $\mathcal{A}$ i.e. it starts with a_b and then follows algorithm $\mathcal{A}$ pushing and popping in elements until it reaches a_t when it stops. The procedure is handled by another auxiliary stack, say $S_{\mathcal{A}}$. We initially feed it a_b along with the context. It then generates either partially compressed blocks or a full block stored explicitly (at level h). The auxiliary stack is also compressed similar to the original stack. After the procedure we can return the reconstructed block and continue with the algorithm $\mathcal{A}$. As the procedure exactly follows algorithm $\mathcal{A}$ but for a smaller input (for block B), the elements in $S_{\mathcal{A}}$ will be the same as B. Note here that as the number of levels for $\mathcal{A}$ is bounded, the *Reconstruction* procedure will also terminate.

Lemma 5.3. *[18] The space used by the Reconstruction procedure is $O(p \lg_p m)$ words, where m is the number of elements between a_b and a_t.*

Proof. : *Reconstructing* a block B can trigger multiple nested reconstructions if, say, a block in $S_{\mathcal{A}}$ becomes empty. But the crucial fact here is that $S_{\mathcal{A}}$ mimics $\mathcal{A}$ but on a smaller input. Hence, for each nested reconstruction the level of a block to reconstruct increases. Thus there can be at most h many reconstructions possible. We can use auxiliary stacks for each level. By Lemma 5.2, each reconstruction procedure takes $O(p \lg_p m)$ space. We observe that the reconstruction procedure at level i can utilize space unused by the original stack itself as then the space allocated in the original stack for levels greater than i remains unused. So the space used by $S_{\mathcal{A}}$ and the other auxiliary stacks can be charged to the space unused in the original stack. Thus the reconstruction operation does not need any extra space other than the space allocated to the original stack. □

Lemma 5.4. *[18] The time taken by the stack algorithm $\mathcal{A}$ is $O(n^{1+1/\lg p})$.*

Proof. The time taken by the *Push* and *Pop* operations are constant time per level. So the total time taken is $O(nh)$. We now bound the time taken by *Reconstruction* using a charging scheme. For a block that is reconstructed, we charge the time taken for that block to the block that, on being empty, had caused the *reconstruction.* We know that a block once popped cannot be popped again, hence we can bound the number of reconstructions by the number of blocks. We should also take into consideration that we are working with a compressed stack. So a block on being reconstructed by $\mathcal{A}$ may have to be compressed again, to be reconstructed again by some smaller block S_i getting empty. Thus each block can be charged twice for a reconstruction. Also the time taken for reconstruction of a block of size p is $O(p)$. If $T(m)$ is the time taken to reconstruct a block of size m, then $T(p) = O(p)$.

Thus we have the recurrence, $T(n) = 2pT(n/p) + O(p)$, where the $O(p)$ denotes the constant time spend to read the context of each a_b to reconstruct. This recurrence solves to $O(n^{1+1/\lg p})$. Thus the total running time is $O(nh + n^{1+1/\lg p}) = O(n^{1+1/\lg p})$, when $h = \lg_p n - 1$. □

Hence, from Lemmas 5.3 and 5.4, Theorem 5.1 follows. □

In [18], the authors have shown that a large number of algorithms in computational geometry literature fits into the stack framework and for those applications they obtained time-space tradeoff using Theorem 5.1. Even though the stack framework is general enough to capture those algorithms, we observe that even for performing a depth first traversal (DFS) on trees, this is not sufficient. In particular, in [18], the input elements are maintained in a list structure so that given access to an input element a_i, the algorithm can access a_{i+1} in constant worst case time. However to perform DFS, the next element of the DFS order is to be determined by navigating the tree, which may involve multiple backtracking, resulting in a non-constant time. Towards capturing such problems, we first generalize the above stack based algorithms to a broader class of algorithms and still prove a version of Lemma 5.1. More specifically,

- We allow the number of auxiliary variables to be a parameter t, which need not be a constant, and any of the auxiliary variables can be accessed in a constant time. This is possible if we simply assume that the auxiliary variables form an array of t elements. We then capture the time and space also as a function of the number of auxiliary variables.
- We assume that the next element to be processed could depend on the pushes and pops done and hence could take more than a fixed constant time to determine. In fact, as the input is in read-only memory, the algorithm may even make several scans over the input (along with the stack and auxiliary storage) and take, say some $g(n, t)$ time (as k is a constant, we ignore its dependence on g) to find the next element to process.
- We also assume that each element is processed at most once.
- Finally, we allow the condition that decides what to push into the stack or whether to pop the stack element to take time an arbitrary function $h(n, t)$ of n and t and

space $s(n)$ words. Hence each push and pop can take $h(n,t)$ time, and hence the entire algorithm takes $O(nh(n,t) + ng(n,t))$ time using $s(n) + t$ words apart from a stack.

We also allow the algorithms to output elements to a write-once output array along the way.

We call such a stack algorithm, a generalized stack based algorithms. We provide the pseudocode below.

Algorithm 6 Basic scheme of a generalized stack based algorithm

```
1: Initialize the stack and auxiliary data structure DS of size $t$
2: Initialize the first input element $a$ to be processed and push it into the stack.
3: repeat
4:     while some-condition(a, input, DS, STACK.TOP(1),..., STACK.TOP($k$)) do
5:         STACK.POP
6:     end while
7:     $a \leftarrow next(a)$ ($next(a)$ is computed based on $a$, all the pops and push done in this
       step as well as the input set)
8:     if some-condition(a, input, DS, STACK.TOP(1),..., STACK.TOP($k$)) then
9:         STACK.PUSH(a or some function of a)
10:    end if
11: until $a$ is NULL
```

For such generalized stack based algorithms, we can prove the following theorem:

Theorem 5.5. *Any generalized stack algorithm that takes $O(n(h(n,t) + g(n,t)))$ time and $O(n + t + s(n))$ space, where $O(n)$ space is for the explicit stack and $O(t)$ space is for auxiliary data structures, and $s(n)$ space is to check the conditions for pushes and pops, can be adapted so that, for any parameter $2 \le p \le n$, it solves the problem in $O((n(h(n,t) + g(n,t)))^{1+(1/\lg p)})$ time using $O(pt \lg_p n + s(n))$ variables.*

Proof. The space for auxiliary data structures play a role in Lemma 5.1 when we compress a block in the stack, and store the 'context' of the auxiliary data structures with the first element of the block. In our case, we can store the values of the t variables into the stack, thereby increasing the storage space by a factor of t. Also the reconstruction steps in

Lemma 5.1 simulate the original stack algorithm repeatedly (sometimes with recursive calls), and hence the ability to compute the next element of an element after processing that element (which could take more than a constant time) does not affect the analysis in any way. At the base case of the recursive step of reconstruction, a block of size p is reconstructed with full stack space of size p which takes $O(p(g(n,t)+h(n,t)))$ time (as only the elements of that block are pushed and popped), so $T(p)=O(p(g(n,t)+h(n,t)))$. The general step remains as before resulting in the earlier recurrence for the total running time as $T(n)=2p(h(n,t)+g(n,t))T(\frac{n}{p})+O(p(h(n,t)+g(n,t))$ which solves to the claimed runtime bound. □

For the generalized stack algorithms we deal with in the chapter, the parameters $s(n)$ and $h(n,t)$ are constants while $g(n,t)$ is an amortized constant. By fixing these parameters in Theorem 5.5, we get the following theorem,

Theorem 5.6. *Any generalized stack algorithm which takes $O(n)$ time and $O(n+t)$ space (where $O(n)$ is for the stack, and t is for the auxiliary variables) can be adapted so that, for any parameter $2 \le p \le n$, it solves the problem in $O(n^{1+(1/\lg p)})$ time using $O(pt \lg_p n)$ variables.*

To start with we use Theorem 5.6, in the next section, to provide a space efficient implementation of some optimization problems on trees and later generalize to bounded treewidth graphs.

5.3 DFS on trees

Asano *et al.* [9] in a recent paper gave an algorithm to perform the DFS traversal of an undirected tree T using $O(\lg n)$ bits of space and linear time. It actually produces rather the Eulerian tour of T. Specifically, their algorithm traverses T in a DFS fashion and outputs every edge of T twice, first time while going down and, secondly, while moving up the tree. The algorithm produces the list of edges in Eulerian tour order and it is not clear how to extract out the vertices in DFS order from this edge list in $O(\lg n)$ space and linear time (actually the number of times a vertex appears in the list is its degree in this list). We show that we can output the vertices in DFS traversal order albeit with a little more time and space. Note that, we assume that the input tree is given in leftmost child and right sibling representation which is strictly weaker than their input representation. We formalise our theorem below.

Theorem 5.7. *Given a rooted undirected tree on n vertices in left-most child and right-sibling representation, we can output the vertices in DFS order in $O(n^{1+(1/\lg p)})$ time using $O(p \lg_p n)$ variables of extra space where $2 \leq p \leq n$.*

Proof. We implement depth first order using a stack which is initialized to contain the root vertex. We output a node as we discover it, and add it to the stack (and so the root is output first). At each step, we add the leftmost child of the node at the top of the stack unless the node has no children. When the stack top node has no children, the node has been fully explored; we pop the node from the stack and add (after outputting) its right sibling (which is the next child of its parent to be explored) to the stack as long as the node has a right sibling. When the stack top node has no children or right sibling, then it is the last child of its parent and hence its parent has been fully explored. So we pop the node off the stack and pop also its parent (which will be at the top of the stack after popping the node) and continue the process until the stack becomes empty. A psuedocode description of the algorithm is given below.

Notice that this algorithm as we described fits into our generalized stack framework (with t and $s(n)$ constants while $g(n,t)$ and $h(n,t)$ are non-constants in some steps though they are constants when amortized over the processing steps of all the elements; hence they do not fit into the original generalized stack framework). Hence Theorem 5.5 can be applied to obtain the desired bounds. □

Algorithm 7 DFS(x)

```
1: Initialize stack $S$ to $NULL$
2: while $x \neq NULL$ do
3:     $Output(x), Push(x, S)$
4:     $x \leftarrow next(x)$
5: end while
6:
7: $next(x)$
8: if $x$ has leftmost child $y$ then
9:     $next(x) \leftarrow y$
10: else
11:     $z \leftarrow$ StackTop (note that $z = x$ initially)
12:     Pop($z, S$)
13:     while $z$ has no right sibling and stack is not empty do
14:         $z \leftarrow$ StackTop
15:         Pop($z, S$)
16:     end while
17:     if stack is empty then
18:         $next(x) \leftarrow NULL$
19:     else
20:         $next(x) \leftarrow right\ sibling(z)$
21:     end if
22: end if
```

Setting p to a small fixed constant or n^{ϵ} for a positive constant $\epsilon < 1$, we obtain

Corollary 5.8. *We can output the vertices of a rooted undirected tree T in DFS order using $O(\lg n)$ variables and $O(n^{1+\epsilon})$ time or using $O(n^{\epsilon})$ variables and $O(n)$ time.*

5.4 Optimization problems

In the next section we give a generic algorithm for problems expressible in CMSO over graphs of bounded treewidth. In this section as a warm up, we explain our algorithms for specific problems on trees. The main reason for doing this is to show that to obtain a space efficient algorithm, all we need to do is to turn natural dynamic programming based algorithms into a stack based algorithm and then use Theorem 5.5 to obtain the desired trade-off between space and time. Thus, these algorithms are useful in practice.

5.4.1 Unweighted Trees

We exemplify our approach by giving time-space tradeoffs for MINIMUM VERTEX COVER, MAXIMUM INDEPENDENT SET, MAXIMUM MATCHING, and MINIMUM WEIGHTED DOMINATING SET (MWDS).

Theorem 5.9. MINIMUM VERTEX COVER *and* MAXIMUM INDEPENDENT SET*,* MAXIMUM MATCHING *and* MINIMUM DOMINATING SET *can be solved on a rooted unweighted tree on n vertices in $O(n^{1+(1/\lg p)})$ time using $O(p \lg_p n)$ variables of extra space where $2 \leq p \leq n$. In fact, we can also output the corresponding set and its cardinality for each of the above problems in the same time and space. The output is generated through an output array (which can not be seen/used by the algorithm once the output has been generated) which will list out the vertices.*

Proof. The standard dynamic programming algorithms for these problems on trees compute these values in a bottom up fashion. To implement those algorithms, we proceed as in the case of the DFS traversal algorithm in Section 5.3 except when we push elements onto the stack, we also push some (tentative) values along with them. And as we compute these values bottom up, the values from the children are passed on to the parents through the stack. Thus we also get around the lack of parent pointer by pushing elements into the stack as we traverse (and so the parent is at the top of the stack when an element is popped), but then this lets the stack to grow to size n. Then we apply the stack compression of Theorem 5.5 to reduce the space. This completes a high level description of our algorithm starting from the root. Below we give detailed specifics of the algorithm for the specific problems claimed in the theorem.

MINIMUM VERTEX COVER: A standard algorithm to find a minimum vertex cover in a tree is based on the following simple observation:

Observation If v is a vertex with degree 1, then there exists a minimum vertex cover that contains v's unique neighbor u.

Hence the algorithm repeatedly includes the neighbor of leaf nodes into the solution and deletes them (and their neighbors) from the tree, until the tree becomes empty. This algorithm can be implemented using a stack as follows: we do a depth first traversal of the tree starting from the root, including every vertex into the stack along with a bit 0 associated with the vertex (to indicate our initial guess to exclude that vertex from the solution) as we visit. The first popping happens when we visit a leaf. While popping out a vertex (after visiting its entire subtree), if its value remains as 0, we make its parent (which would be at the top of the stack) 1. Once a node becomes 1, it stays as 1 during its lifetime in the stack. While popping a node if its associated bit is 1, we output it in the output array. Thus in the third line of Algorithm 7, replace Push(X) by Push($X, 0$) and Pop(X) by the following code snippet,

Pop(X, b);
If $b = 1$, output X;
If $b = 0$; $Z \leftarrow StackTop$;
Pop(Z, b);
Push($Z, 1$).

We also maintain a variable which is incremented everytime we output a vertex so that, along with the set, we can also report the cardinality of the optimum solution. This completes the description of the algorithm, and it is easy to see that this algorithm essentially implements the algorithm outlined above using a stack and a constant number of variables. When a vertex assigned 0 is popped it implies that it is a leaf and hence its parent is assigned 1 which is correct by the observation above.

Maximum Independent Set: This can be solved similarly. We provide the details below for completeness. Basically we use the observation that the complement of minimum vertex cover is a maximum independent set. So, the algorithm is exactly similar to the minimum vertex cover algorithm with the switching of the roles of 1 and 0. i.e., we set the bit of the vertex to 1 as we push into the stack, and whenever a vertex is popped, if its value is 1, we set its parent's value to 0. And a node getting 0 will remain as 0 during its stay in the stack. And when we pop a vertex whose value is set to 1, we output it. We can report the cardinality of the solution in similar way as before.

MAXIMUM MATCHING: We modify the minimum vertex cover algorithm as follows. When we pop out a vertex with value 0, if its parent (which would be at the top of the stack) has value 0, we set it to 1, and output the edge joining it and its parent as part of the maximum matching. This ensures that the set of edges in the output form a matching, and every time when a vertex becomes 1 the first time, an edge (joining it to the vertex which made it 1) is taken into the matching, ensuring that the size of the matching output is the same as the size of the (minimum) vertex cover. This proves that what we output is a maximum matching (by Konig's theorem, as a tree is a bipartite graph).

MINIMUM DOMINATING SET: Similar to the algorithm for MINIMUM VERTEX COVER, this algorithm also uses the observation that there exists a minimum dominating set that contains the unique neighbor of any leaf node. However, unlike minimum vertex cover algorithm where a vertex along with its incident edges can be deleted once included in a solution (and the problem can be solved recursively), before we delete a vertex that has been included in the dominating set, we mark its parent as one that has been already dominated. Furthermore, we include the parent of a leaf node in the dominating set only if the leaf node has not been dominated already. In particular, if all the children (leaves) of a node are already dominated, then the parent is *not* included in the solution unless it is a root. Detailed implementation of the algorithm with a stack is given below.

As before, we do a depth first traversal and assign the number 0 to the vertex as it is pushed into the stack. At any point of time, each vertex (in the stack) is in one of the three states indicated by its assigned bit. A 1 bit indicates that the vertex is in the dominating set (constructed so far), while 2 indicates that the vertex, though not in the dominating set, has already been dominated. An assigned bit of 0 for a vertex indicates that the vertex has not been dominated and we are yet to decide about the vertex's presence in the dominating set.

While popping out a vertex v (after visiting its entire subtree), if it has value 0 and has a parent, then we assign the parent (which is at the top of the stack) value 1; if that vertex is the root, we assign it value 1; if the vertex that was popped out has value 1, then we assign 2 to the parent if the parent was not already assigned 1. In all other cases (if the parent of the popped vertex with value 1, was already 1 or if the popped vertex has value 2), we do nothing. The vertices which are assigned 1 while popping out form the solution set and hence are output as we pop them out. As before, the assignment to each vertex is forced in a bottom-up fashion. Therefore when a vertex assigned 0 is popped it implies that none of its children has dominated the vertex nor does the vertex belong to the dominating set. So if the vertex has no parent we take it into the solution, assigning value 1. If it has a parent we assign the parent as 1 as the parent can dominate more vertices than the vertex itself. This proves the correctness of the algorithm.

It is also clear that the algorithms satisfy the generalized stack framework and hence the claimed bounds follow from Theorem 5.5. □

5.4.2 Weighted Trees

The algorithms we described for Minimum Vertex Cover or Maximum Independent Set use the observation that for any leaf, its unique neighbor can be picked up in any maximum independent set or a minimum vertex cover. This is no longer true in the weighted variants of these problems. Thus, in this case we resort to a modification of standard dynamic programming algorithm. However, this comes at a cost: now we cannot output the desired set in the same running time, but with a linear factor increase in the runtime. We exemplify the approach via Minimum Weighted Dominating Set (MWDS), Minimum Weighted Vertex Cover and Maximum Weighted Independent Set.

Theorem 5.10. *Given a rooted node-weighted tree on n vertices where every vertex has a positive weight, given in the form of a read-only memory pointer representation, we can determine*

- *the weight of a maximum weight independent set, a minimum weight vertex cover and a minimum weight dominating set in the tree in $O(n^{1+(1/\lg p)})$ time using $O(p \lg_p n)$ variables of extra space where $2 \leq p \leq n$; furthermore, using the same time and space,*
  - *we can determine whether or not the root is in the optimum solution of a maximum weight independent set or a minimum weight vertex cover.*
  - *we can determine whether or not the root is in the optimum solution for a minimum dominating set. If the root is not in the minimum weight dominating set whose weight is $D(root)$, we can also find the child that dominates the root. Furthermore, we can also find in the same time and space,*
    - *$E(root)$ = The weight of a minimum dominating set that does not contain the root.*
    - *$DE(root)$ = The weight of a minimum dominating set that dominates all vertices of the subtree except possibly the root; also whether or not the root is in such a dominating set.*
    - *$I(root)$ = The weight of a minimum dominating set that contains the root.*

Proof. We describe the essence of the algorithms first and suggest modifications needed for them to use just a stack. The standard dynamic programming algorithm to compute weighted Maximum Independent Set on trees computes the following values for every node v in bottom up fashion:

$M^+[v]$ = weight of the maximum weight independent set of the subtree rooted at v containing v.

$M^-[v]$ = weight of the maximum weight independent set of the subtree rooted at v not containing v.

Once we have computed these quantities for all the children $v_i \cdots v_d$ of an internal node v, then we can compute it for v as follows:

$$M^+[v] = w[v] + \sum_{i=1}^{d} M^-[v_i]; M^-[v] = \sum_{i=1}^{d} \max\{M^+[v_i], M^-[v_i]\}$$

As before, our algorithms proceed in a depth first fashion inserting the nodes, and some initial values with them as we push them into the stack. The initial values for any node as they are inserted into the stack are $w(v)$ for $M^+[v]$ and 0 for $M^-[v]$. Observe that for a vertex v, what we actually push into a stack is the tuple, Push$(v, M^+[v], M^-[v])$ in the third line of Algorithm 7. It is easy to compute these values for leaf nodes. Once we pop a vertex u, we would have visited its entire subtree, and hence computed these values for that vertex. We, then update its parent v (which is at the top of the stack) with its contribution to both these quantities of its parent. More precisely, we update $M^+[v]$ by $M^+[v] + M^-[u]$, and $M^-[v]$ by $M^-[v] + \max\{M^+[u], M^-[u]\}$. It is easy to see that the above expression correctly computes the two quantities for every node in a postorder traversal using just a stack and the final answer is given by $\max\{M^+[root], M^-[root]\}$. Furthermore we see that the root will be in the solution if $M^+[root] \geq M^-[root]$.

The weight of the Minimum Vertex Cover can be computed by the straightforward modification of the $M^+[v]$ and $M^-[v]$ stated above. And then similar depth first traversal of the tree along with value updation in stack and final application of our generalized stack compression theorem yields same running time and space bounds for Minimum Vertex Cover.

The standard dynamic programming algorithm for MWDS needs the values at each child of a node before computing the value of the node, which may result in a storage more than a stack. To recall, a dynamic programming algorithm works as follows. It computes four quantities $I(v), D(v), DE(v), E(v)$ for each vertex v, which are defined as below.

1. $E(v)$ = The weight of a minimum dominating set of the subtree rooted at v, that excludes vertex v.
2. $DE(v)$ = The weight of a minimum dominating set of the subtree rooted at v that dominates all vertices of the subtree except possibly v.
3. $I(v)$ = The weight of a minimum dominating set of the subtree rooted at v that contains vertex v.
4. $D(v)$ = The weight of a minimum dominating set of the subtree rooted at v.

Clearly $D(v) = \min\{I(v), E(v)\}$. It is easy to find these values for leaf nodes, and once we have computed these quantities for all the children of a node v, they can be computed for v as follows: let $u_1, u_2, \ldots u_d$ be the children of v. Then
$I(v) = w(v) + \sum_{i=1}^{d} DE(u_i)$,
$DE(v) = \min\{I(v), \sum_{i=1}^{d} D(u_i)\}$,
$E(v) = \min_{1\leq i\leq d}\{I(u_i) + \sum_{k=1,k\neq i}^{d} D(u_k)\}$

Now, all these quantities can be computed in a depth first order as follows. We initialize $DE(u)$ and $E(u)$ to 0, and $I(u)$ to $w(u)$, as we push a vertex u into the stack. Once we pop a vertex u, we would have visited its entire subtree, and hence computed these values for that vertex. We can then easily compute $D(u)$ for that vertex. Then we update its parent (which is at the top of the stack) with its contribution to all of these quantities of its parent. More precisely, for the parent vertex v, we make $I(v) \leftarrow I(v) + DE(u); DE(v) \leftarrow DE(v) + D(u)$.

Updating $E(v)$ is a bit tricky, but it becomes easier, if we rewrite the quantity $E(v)$ in the above equation as,

$$E(v) = \min_{1\leq i\leq d}\left\{\sum_{i=1}^{d} D(u_i) + (I(u_i) - D(u_i))\right\} = DE(v) + \min_{1\leq i\leq d}(I(u_i) - D(u_i)).$$

To compute $D(v)$, we simply keep track of $\min\{I(u)-D(u)\}$ over its children u of v and also the vertex u that realizes the min. When we pop a vertex u, we update the DE value of its parent v as $DE(v) = DE(v) + D(u)$ and update the $\min\{I(u)-D(u)\}$ at its parent by $(I(u) - D(u))$ if the $(I(u) - D(u))$ quantity is smaller. When all the children are popped, $E(v)$ is set to $E(v) = DE(v) + (I(u) - D(u))$. It is easy to see that the above expression correctly computes the four quantities in a postorder traversal using just a stack and the final answer is given by $D(root)$.

If $D(root) = I(root)$ then we conclude that the root in the solution. If not, then $D[root] = E[root] = DE[root] + \min\{I(u) - D(u)\}$ where the *min* is taken over the children u of *root*. Hence if the root is not in the solution then the child u realizing the min is the vertex that dominates the root. It is easy to see that we can implement the above described procedure in the template of Algorithm 7 and, hence, using Theorem 5.5 we get the claimed running time and space bounds. □

5.4.3 Reporting Solution Set in the Weighted Case

Theorem 5.10 gives only the output weight of the solution. But to output a set that actually realizes the optimal solution is a bit tricky. In particular this can not be computed in a bottom-up fashion, as the presence of a vertex in the optimum solution depends both on its parent as well as its children. One approach to solve this is to perform the "forward" direction as in Theorem 5.10 and determine whether the root node is in the optimum solution or not. Then trace back to the leaves back through the entries in the earlier nodes that gave rise to the optimum solution. This results in a linear factor increase in the runtime. More specifically, we do a depth first traversal of the tree, starting from root, calling Theorem 5.10 at each node to determine appropriate values at the node, including its presence or absence in the optimum solution in the subtree rooted at the vertex, and store them in a stack. Its actual value (whether or not it is in the optimum solution of the entire tree) is computed based also on the appropriate values of its parent, that is stored in the stack. We let this stack grow to linear size, and use the stack compression lemma to compress it to the required space. We formalize our theorem below.

Theorem 5.11. *Given a node-weighted tree on n vertices where every vertex has a positive weight, given in the form of a read-only memory representation, we can determine the solution set that realizes a* MAXIMUM WEIGHTED INDEPENDENT SET *or a* MINIMUM WEIGHTED VERTEX COVER *or a* MINIMUM WEIGHTED DOMINATING SET *in the tree in $O(n^{(2+\frac{1}{\lg p})})$ time and $O(p \lg_p n)$ space where $2 \leq p \leq n$.*

Proof. We examplify our theorem with MAXIMUM WEIGHTED INDEPENDENT SET and a straightforward modification can output the solution set for MINIMUM WEIGHTED VERTEX COVER. Finally we show how to produce the solution set realizing an optimal value for MINIMUM WEIGHTED DOMINATING SET.

MAXIMUM WEIGHTED INDEPENDENT SET: We perform a depth first order calling Theorem 5.10 at each node to determine whether the node (the root of the subtree rooted at the node) is in the maximum independent set. Let $b[v] = 1$ if v is in the maximum weighted independent set of the subtree rooted at v (as obtained from Theorem 5.10), and 0 otherwise. As we visit a node v in depth first order, we push the node label along with its b value in the stack. Let v be an internal node (encountered in the DFS order) and let u be v's parent. If $b[u]$, which is available in the stack top, is 1, then we simply set $b[v]$ to 0 (in which case we need not even call Theorem 5.10 at v) and move on. If $b[u]$ is 0, then we call theorem 5.10 with the subtree rooted at u, and set its b value appropriately and insert it into the stack. Whenever we set the b value of a node to 1, we output it. Compression, Pop, Push are implemented exactly as in the proof of Theorem 5.9. Furthermore, as we run Theorem 5.10 for every vertex of the tree, our run time is n times the runtime of Theorem 5.10, i.e. $O(n^{2+(1/\lg p)})$. Hence the claimed bounds for time and space.

MINIMUM WEIGHTED DOMINATING SET: This proceeds in a similar fashion to the above. We run Theorem 5.10 as a black box on every vertex of the tree in depth first traversal order, and go on inserting them in a stack along with its associated bit value b determining whether vertex v is in the solution or not. If the b value of a node is set to 0, we also store the label of a node (a child or its parent) value x that dominates the node (obtained from Theorem 5.10), in the stack. Initially for the root, $b[root]$ is set to 1 if root is in the minimum dominating set, and is set to 0 otherwise. If $b[root] = 0$, we find the child x that dominates it from Theorem 5.10 and store it in the stack.

For a node u with parent v (on the top of the stack), if $b[v]$ is to 1, then we call Theorem 5.10 to determine $DE(u)$ and set $b[u]$ to 1 if u is in such a dominating set (determined by Theorem 5.10) and 0 otherwise. If $b[u]$ happens to be 0, we simply set the node (that dominates it) value to the label of the parent v.

For a node u with parent v whose b value is 0, if the label of u is stored in the parent (as the node that dominates it), then we call Theorem 5.10 to find $I(u)$, and set $b[u]$ to 1. Otherwise, we simply call Theorem 5.10 to compute $D(u)$ and set $b[u]$ and the child that dominates it (if appropriate) and push them into the stack.

Once we set a b value of a node in the stack to 1, we can output it. It is easy to see that this correctly outputs the set realizing the optimal value and we invoke Theorem 5.10 total n times i.e., for every vertex of the tree, hence the bounds for time and space follow. □

5.5 Algorithms for graphs of bounded treewidth

In this section we design the most general result, namely the space efficient version of optimization (which actually works for larger class of graphs than bounded treewidth graphs and larger class of properties than CMSO expressible ones) variant of Courcelle's Theorem. We follow the proof of Borie *et al.* [28] and use the machinery of stack compression to obtain the desired theorem. We start with the notations and set up the language in which we will be working with.

5.5.1 Treewidth

For a rooted tree T and a non-root node $t \in V(T)$, by parent(t) we denote the parent of t in the tree T. For two nodes $u, t \in T$, we say that u is a *descendant* of t, denoted $u \preceq t$, if t lies on the unique path connecting u to the root. Note that every node is thus its own descendant.

Definition 5.12 (tree decomposition). *A* tree decomposition *of a graph G is a pair (T, β), where T is a rooted tree and $\beta : V(T) \to 2^{V(G)}$ is a mapping such that:*

- *for each node $v \in V(G)$ the set $\{t \in V(G) | v \in \beta(t)\}$ induces a nonempty and connected subtree of T,*
- *for each edge $e \in E(G)$ there exists $t \in V(T)$ such that $e \subseteq \beta(t)$.*

The set $\beta(t)$ is called the *bag at t*, while sets $\beta(u) \cap \beta(v)$ for $uv \in E(T)$ are called *adhesions*. Following the notation from [90], for a tree decomposition (T, β) of a graph G we define auxiliary mappings $\sigma, \gamma : V(T) \to 2^{V(G)}$ as

$$\sigma(t) = \begin{cases} \emptyset & \text{if t is the root of } T \\ \beta(t) \cap \beta(\text{parent}(t)) & \text{otherwise} \end{cases}$$

$$\gamma(t) = \bigcup_{u \preceq t} \beta(u)$$

We now define a class of graph optimization problems, called MIN/MAX-CMSO$[\psi]$, with one problem for each CMSO sentence ψ on graphs, where ψ has a free vertex (edge) set variable S.

5.5.2 Counting Monadic Second Order Logic.

We use Counting Monadic Second Order Logic (CMSO), an extension of MSO, as a basic tool to express properties of vertex/edge sets in graphs. The syntax of Monadic Second Order Logic (MSO) of graphs includes the logical connectives $\vee$, $\wedge$, $\neg$, $\Leftrightarrow$, $\Rightarrow$, variables for vertices, edges, sets of vertices, and sets of edges, the quantifiers $\forall$, $\exists$ that can be applied to these variables, and the following five binary relations:

1. $u \in U$ where u is a vertex variable and U is a vertex set variable;
2. $d \in D$ where d is an edge variable and D is an edge set variable;
3. $\mathbf{inc}(d, u)$, where d is an edge variable, u is a vertex variable, and the interpretation is that the edge d is incident with the vertex u;
4. $\mathbf{adj}(u, v)$, where u and v are vertex variables and the interpretation is that u and v are adjacent;
5. equality of variables representing vertices, edges, sets of vertices, and sets of edges.

In addition to the usual features of monadic second-order logic, if we have atomic sentences testing whether the cardinality of a set is equal to q modulo r, where q and r are integers such that $0 \leq q < r$ and $r \geq 2$, then this extension of the MSO is called the *counting monadic second-order logic.* So essentially CMSO is MSO with the following atomic sentence for a set S:

$$\mathbf{card}_{q,r}(S) = \mathbf{true} \text{ if and only if } |S| \equiv q \pmod{r}.$$

We refer to [6, 53, 52] and the book of Courcelle and Engelfriet [54] for a detailed introduction to CMSO. In [54], CMSO is referred to as CMS_2.

The MIN-CMSO problem defined by ψ is denoted by MIN-CMSO[ψ] and defined as follows.

MIN-CMSO[ψ]
Input: A graph $G = (V, E)$.
Question: Find the cardinality of a minimum sized subset $S \subseteq V$ ($S \subseteq E$) (if exists) such that $(G, S) \models \psi$.

The definition of MAX-CMSO[ψ] problem is analogous to the MIN-CMSO[ψ] problem. The only difference is that now we try to find a maximum sized subset $S \subseteq V$. Here, we only give an algorithm for MIN/MAX-CMSO[ψ] problems when S is a vertex subset. All of our results can be extended to edge setting in a straightforward way. In particular, an edge set problem on graph $G = (V, E)$ can be transformed to a vertex subset problem on the edge-vertex incidence graph $I(G)$ of G, which is a bipartite graph with vertex bipartitions V and E with edges between vertices $v \in V$ and $e \in E$ if and only if v is incident with e in G. Observe that the treewidth of G and $I(G)$ only differ by a factor of 2 [56]. To make the translation work in the proof, it is sufficient to use the fact that the property of being an incidence graph of a graph G is expressible in CMSO. To avoid complications in our proof we omit the details for this. From now on *we only concentrate on* MIN/MAX-CMSO[ψ] *problems defined over vertex subsets.*

Now we give a couple of examples of problems, that can be encoded using MIN/MAX-CMSO[ψ]. To express MAXIMUM INDEPENDENT SET we do as follows. Given a graph G and a vertex subset X, a simple constant length formula $\mathbf{indp}(X)$ that verifies that X is an independent set of G is: $\forall_{x \in X} \forall_{y \in X} \neg \mathbf{adj}(x, y)$. Thus we can express MAXIMUM INDEPENDENT SET using the above logical sentence. Let us consider another example, namely, MINIMUM DOMINATING SET. Given a graph G and a vertex subset X, a simple constant length formula $\mathbf{dom}(X)$ that verifies that X is a dominating set of G is: $\forall_{x \in V(G)} [x \in X \vee \exists_{y \in X} \mathbf{adj}(x, y)]$.

Dynamic programming algorithms over tree decompositions. The standard dynamic programming algorithms on bounded treewidth graphs for standard optimization problems, see [6, 25, 26] proceed by doing a bottom-up traversal of the tree computing tables at every node starting from the leaves. Aspvall et al, in [13] identify the space requirement of the algorithms and argue that $O(\lg n)$ 'open tables' are sufficient in the bottom-up implementation. Bodlaender and Telle [27], building on the work of Aspvall et al, claim without proof that any property expressible in monadic second order logic can be implemented using $O(\lg n)$ tables in the bottom-up traversal.

However, implementing this in the read-only memory model without parent pointers, provide challenges in terms of space, and that is our task in this section.

In particular, we provide an alternate $O(\lg n)$ word space algorithm that precludes the need for parent pointers in the input representation, and uses the stack compression machinery recently developed by Barba et al [18] to prove the following theorem.

Theorem 5.13. *Let G be a graph given with a tree decomposition $(T = (V_T, E_T), \beta)$ of width k. Then* MIN/MAX-CMSO[ψ] *can be solved in time $O(\tau(k) \cdot n^{1+(1/\lg p)})$ time and $O(\tau(k) \cdot p \lg_p n)$ space algorithm, for any parameter $2 \leq p \leq n$. Here, $|V| = n$ and τ is a function of k alone.*

In fact we prove a weighted variant of Theorem 5.13. We also obtain an algorithm that not only outputs the weight of a maximum weighted subset (or a minimum weighted subset) S such that $(G, S) \models \psi$, but also the set S. We call this version of the problem CONSTRUCTIVE-WEIGHTED-MIN/MAX-CMSO[ψ].

Theorem 5.14. *Let G be a graph given with a tree decomposition $(T = (V_T, E_T),$ $\beta)$ of width k. Then* CONSTRUCTIVE-WEIGHTED-MIN/MAX-CMSO[ψ] *can be solved in time $O(\tau(k) \cdot n^{2+(2/\lg p)})$ time and $O(\tau(k) \cdot p \lg_p n)$ space algorithm, for any parameter $2 \leq p \leq n$. Here, $|V| = n$ and τ is a function of k alone.*

In what follows we prove Theorems 5.13 and 5.14. Towards this we first give definitions of recursive graphs and regular properties.

5.5.3 k-terminal recursive graphs

In this section we use the following alternative definition of treewidth, based on *terminal graphs*. A *k-terminal graph* $G = (V, T, E)$ is a graph with an ordered set $T \subseteq V$ of at most k distinguished vertices, called *terminals*. Denote by $\tau(G)$ the number of terminals of graph G.

A k-terminal graph (V, T, E) is a *base graph* if $V = T$. We define *composition operations* over the set of k-terminal graphs. A composition operation f is of arity 1 or 2. When f is of arity 2, it acts on two k-terminal graphs G_1, G_2 and produces a k-terminal graph $G = f(G_1, G_2)$ as follows. It first makes disjoint copies of the two graphs, then "glues" some terminals of graphs G_1 and G_2. Operation f is represented by a matrix $m(f)$. The matrix has 2 columns and $\tau(G) \leq k$ lines, its values are integers between 0 and k. At row i of the matrix, element $m_{ij}(f)$ indicates which terminal of graph G_j is identified to terminal number i of G. If $m_{ij}(f) = 0$ it means that no terminal of G_j was identified to terminal number i of G. A terminal of G_j can be identified to at most one terminal of G (a column j cannot contain two equal non-zero values). Note that if $m_{i1}(f) = 0$ and $m_{i2}(f) = 0$ it means that terminal i of G is a new vertex.

When f is of arity 1, its matrix $m(f)$ has only one column. The k-terminal graph $G = f(G_1)$ is obtained from graph G_1 and matrix $m(f)$ as above, by identifying terminal $m_{i1}(f)$ to terminal number i in G.

Observe that the number of possible composition operations over k-terminal graphs is bounded by some function of k. We say that a k-terminal graph G is *k-terminal recursive* if it can be obtained from k-terminal base graphs through a sequence of composition operations. This sequence is called the *k-expression* of graph G.

Lemma 5.15 ([25]). *For any tree decomposition of width k of graph $G = (V, E)$ and any bag W of the decomposition, (V, W, E) is a $(k + 1)$-terminal recursive graph.*

Let G be a graph and (T, β), be a tree-decomposition. For any node $v \in V(T)$, we think of $G[\gamma(v)]$ as $(k+1)$ terminal graphs with vertices in $\beta(v)$ as terminals.

5.5.4 Regular properties

To use the framework of Borie et al. [28], we need a notion of *regular properties.* Let $\mathcal{G}$ and $\mathcal{G}_k$ denote the family of graphs and the family of all k-terminal graphs, respectively. By Γ_{versub} ($\Gamma^k_{\mathsf{versub}}$), we denote the set of (graph $\times$ vertex subsets) pairs (G, S) such that $G = (V, E) \in \mathcal{G}$ ($G = (V, E) \in \mathcal{G}_k$) and $S \subseteq V$. A *vertex property $\mathcal{P}$* is a function from Γ_{versub} to $\{0, 1\}$. We can similarly define a notion of *edge property $\mathcal{P}$* over $\Gamma_{\mathsf{edgesub}}$. By *property $\mathcal{P}$* we mean either a vertex property or an edge property.

We say that a CMSO-sentence φ expresses a graph property $\mathcal{P}$ if $\mathcal{P}(G, X)$ is true if and only if $(G, X) \models \varphi$ (i.e., the sentence φ is true exactly on graphs G and vertex/edge subsets S such that $\mathcal{P}(G, X)$ is true). Borie et al. [28] defined *regular properties*, whose definition we give soon. For all our applications, we need only the fact from Borie et al. [28] that every property $\mathcal{P}$ expressible by a CMSO-formula is regular.

Let $G = (V, T, E)$ be a $(k+1)$-terminal recursive graph. For a composition operation f, let $\circ_f$ denote the composition operation over pairs (G, X), where f extends in a natural way over the values of vertex sets. If $G = f(G_1)$ then $\circ_f((G_1, X)) = (G, X)$. If $G = f(G_1, G_2)$ then $\circ_f((G_1, X_1), (G_2, X_2)) = (G, X)$, the operation being valid only if, for each pair of terminals that are merged together to form a new terminal vertex "i" of G, either all these terminals are members of their respective X's or all of them are vertices of the set X which contains exactly those terminals that meet the former requirements.

Definition 5.16 (Regular Property)**.** *Consider a property $\mathcal{P}$. A property $\mathcal{P}$ is called* regular *if, for every k, there exists a finite set $\mathcal{C}$, a homomorphism h associating to each k-terminal recursive graph $G = (V, E)$ and every $X \subseteq V$ ($X \subseteq E$) a class $h(G, X) \in \mathcal{C}$ ($h :\ \Gamma^k_{versub}(\Gamma^k_{edgesub}) \to \mathcal{C}$), and an* update function *$\odot_f : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ for each composition operation f of arity 2 (resp. $\odot_f : \mathcal{C} \to \mathcal{C}$ for each composition operation f of arity 1),* *satisfying:*

- *($\mathcal{P}$ is preserved) If $h(G_1, X_1) = h(G_2, X_2)$ then $\mathcal{P}(G_1, X_1) = \mathcal{P}(G_2, X_2)$.*
- *(integrity of operations) For any composition operation f, we have that*

  $$h(\circ_f((G_1, X_1), (G_2, X_2))) = \odot_f(h(G_1, X_1), h(G_2, X_2))$$

  if f is of arity 2, and

  $$h(\circ_f(G_1, X_1)) = \odot_f(h(G_1, X_1))$$

  if f is of arity 1.

We point out that the homomorphism class $h(G, X)$ depends on G and on the value of X. Typically the class of $h(G, X)$ encodes, among other information, the intersection of X with the set of terminals. For example, if the composition operation $\circ_f((G_1, X_1), (G_2, X_2))$ is not valid, then $\odot_f(c_1, c_2)$, where c_1 and c_2 are the respective homomorphism classes of (G_1, X_1) and of (G_2, X_2), is also undefined.

Typical algorithms over recursively constructed graphs proceed by dynamic programming. When browsing the $(k + 1)$-expression of G, the algorithm stores in each node a table of *classes* (sometimes called *characteristics*) depending on the branch of the current sub-expression and the partial solutions (i.e., possible subsets of X) encountered so far. Let G_1 be such a sub-expression and let X_1 be a subset of vertices that we aim to extend into the solution X. The intuition is that if the class of (G_1, X_1) is the same as the class of some other pair (G_2, X_2), then we can replace the branch of G_1 by an expression of G_2, and the new graph G' is such that X_1 extends into a solution $X_1 \cup Y$ of G if and only if X_2 extends into a solution $X_2 \cup Y$ of G'.

In order to efficiently solve our problem, we need an efficient computation of classes for base graphs, as well as an efficient computation of the classes for compositions of graphs and partial solutions. Note that for any fixed k and any regular property $\mathcal{P}$, the number of classes is a constant, the size of $\mathcal{C}$. Nevertheless, this constant depends on k and on the property $\mathcal{P}$. For algorithmic purposes, given k and $\mathcal{P}$, we need an explicit algorithm computing the homomorphism class of a given base graph, and an algorithm computing the update functions $\odot_f$. I.e., we need an algorithm that takes as input a composition operation f and one or two classes $c_1, c_2 \in \mathcal{C}$ and computes the class $\odot_f(c_1, c_2)$ if f is of arity 2 (resp. $\odot_f(c_1)$ if f is of arity 1). Eventually, we must know the set of *accepting classes*, that is the set of classes c such that $h(G, X) = c$ implies that $\mathcal{P}(G, X)$.

For our proof we will need following relation between regular properties and properties expressible using a CMSO formula.

Lemma 5.17 (Borie *et al.* [28])**.** *Any property $\mathcal{P}(G, X)$ expressible by a CMSO-formula is regular.*

Moreover, the result of Borie *et al.* [28] is constructive in the sense that, given a CMSO-formula, it provides in linear time the homomorphism classes $\mathcal{C}$, the subset of accepting classes and the algorithms computing the classes of base graphs as well as the update functions for the regular property $\mathcal{P}$ on $(t + 1)$-terminal recursive graphs. The regularity is actually proven in [28] for all properties expressible by CMSO-formulae, which allows an arbitrary number of free variables over vertices, edges, vertex sets and and edge sets. For our proof, it is sufficient to consider properties over graphs and one vertex set, corresponding to formulae with a unique free variable, which is a set of vertices.

Finally, we are ready to give proofs of Theorems 5.13 and 5.14.

Theorem 5.13 (Restated). *Let G be a graph given with a tree decomposition $(T = (V_T, E_T), \beta)$ of width k. Then* MIN/MAX-CMSO$[\psi]$ *can be solved in time $O(\tau(k) \cdot n^{1+(1/\lg p)})$ time and $O(\tau(k) \cdot p \lg_p n)$ space algorithm, for any parameter $2 \leq p \leq n$. Here, $|V| = n$ and τ is a function of k alone.*

Proof. First we show how to transform the proof of Borie *et al.* [28] into the stack framework. Then we get the desired result by applying Theorem 5.5.

The property ψ is regular and hence there exists a finite set $\mathcal{C}$ and a homomorphism function $h : \Gamma_S \to \mathcal{C}$. A class $c \in \mathcal{C}$ is said to be accepting if and only if $h(G, X) = c$ implies that $(G, S) \models \psi$. Recall that the size of $\mathcal{C}$ only depends on k and ψ. Since ψ is fixed, we can assume that $|\mathcal{C}| \leq \eta(k)$, for some function η that only depends on k. We will obtain our result by doing a bottom-up dynamic programming over T starting from leaves. For every $v \in V_T$, by $G_v = (\gamma(v), E_v)$, we denote the graph $G[\gamma(v)]$. We will view G_v as a $k+1$-terminal graph $(\gamma(v), \beta(v), E_v)$. For the algorithm at each node $v \in V_T$ we have a table stored as an array $\mathbb{T}_v$ indexed by $c \in \mathcal{C}$ and $Z \subseteq \beta(v)$ and stores the following. Let

$$\mathcal{F}(v, c, Z) = \{X \subseteq \gamma(v) \mid X \cap \beta(v) = Z,\ h((G_v, X)) = c\}.$$

If $\mathcal{F}(v, c, Z)$ is non-empty then by $\mathsf{value}(\mathbb{T}_v, c, Z)$ we denote the size of a largest (or a smallest depending on whether we are solving a maximization or minimization problem) set in $\mathcal{F}(v, c, Z)$ else we set it to 0. Thus, for every $c \in \mathcal{C}$, $\mathbb{T}_v[v, c, Z] = \mathsf{value}(\mathbb{T}_v, c, Z)$. Observe that the size of the table is upper bounded by $O(2^k \eta(k))$. The next question is how we update the table of a node when we know the tables for its children.

Towards this we use the proof of Borie *et al.* [28].. Recall that, an update function $\odot_f : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ for each composition operation f of arity 2 (resp. $\odot_f : \mathcal{C} \to \mathcal{C}$ for each composition operation f of arity 1), satisfying:

- (integrity of operations) For any composition operation f, we have that

$$h(\circ_f((G_1, X_1), (G_2, X_2))) = \odot_f(h(G_1, X_1), h(G_2, X_2))$$

if f is of arity 2, and

$$h(\circ_f(G_1, X_1)) = \odot_f(h(G_1, X_1))$$

if f is of arity 1.

We use the update function to obtain a table for a node in V_T, given tables for its children.

Now we give details for how we obtain a table at node v given tables for its children $v_1, \dots, v_p$. We view the graph G_v as follows.

$$G_v = f(f(\cdots f(f(f(\beta(v), G_{v_1}), G_{v_2}), G_{v_3}), \dots), G_{v_p}). \tag{5.1}$$

Here, the composition operation $f(G_1, G_2)$ takes two $(k+1)$-terminal graphs, with terminal sets W and W_1 respectively, and composes them into a new $(t+1)$-terminal graph having W as set of terminals. In the gluing operation, terminal number j of W_i is glued on terminal number ℓ of W if and only if they correspond to the same vertex of G. Hence, this composition operation $f(G_1, G_2)$ only depends on W and W_1. In particular, the terminals for G_v is $\beta(v)$. While designing our algorithm we construct the graph G_v using f applied from left to right in Expression (5.1). Let $G_1 = f(G[\beta(v)], G_{v_1})$. Then, for $i \in \{2, \dots, p\}$, we define $G_i = f(G_{i-1}, G_{v_i})$. Observe that $G_v = G_p$. Let $\mathbb{T}_i$ correspond to the table for G_i. Observe that f is an arity 2 composition function. We will first compute the table $\mathbb{T}_0^v$ for $G[\beta(v)]$. Observe that $G[\beta(v)]$ is a base graph and thus we can use Borie *et al.* [28] to obtain the homomorphism classes $\mathcal{C}$, the subset of accepting classes. Now we go over every subset $X \subseteq \beta(v)$ and compute $h(G[\beta(v)], X)$. Since the size of $\beta(v)$ is at most $k+1$, we can do this in $\tau(k)$ time. Given $h(G[\beta(v)], X)$, we can easily obtain $\mathbb{T}_0^v$. Observe that when v does not have a child, that is, when v is a leaf node, then $\mathbb{T}_v = \mathbb{T}_0^v$. Now we show how to compute the table $\mathbb{T}_i$ for, say, $G_i = f(G_{i-1}, G_{v_i})$, $i \in \{1, \dots, p\}$, given the table $\mathbb{T}_{i-1}$ for G_{i-1} (or $\mathbb{T}_0 = \mathbb{T}_0^v$) and the table $\mathbb{T}_{v_i}$ for G_{v_i}. For this we will use the composition operator $\odot_f : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$. For every $c \in \mathcal{C}$ define a family of pair of classes as follows: $\mathsf{Classes}(c) = \{(c_1, c_2) \mid c_1, c_2 \in \mathcal{C},\ \odot_f (c_1, c_2) = c\}$. Now given a set

$Z \subseteq \beta(v)$ and $c \in \mathcal{C}$, we obtain $\mathbb{T}_i[v, c, Z]$ as follows. Define,

$$\begin{aligned} \mathcal{V}(v, c, Z) \quad &= \quad \Big\{ \mathsf{value}(\mathbb{T}_{i-1}, c_1, Z) + \mathsf{value}(\mathbb{T}_{v_i}, c_2, Z_1) - |Z \cap Z_1| \;\Big|\; \\ &\qquad Z_1 \cap \beta(v) \subseteq Z, (c_1, c_2) \in \mathsf{Classes}(c) \Big\}. \end{aligned} \tag{5.2}$$

We set $\mathbb{T}_i[v, c, Z]$ with the largest (or the smallest depending on the problem type) integer in $\mathcal{V}(v, c, Z)$. The solution to the optimization problem is found by taking the minimum (or maximum) over those entries in $\mathbb{T}_r$ such that in its index we have $c \in \mathcal{C}$, that is an accepting class. Observe that the running time to compute a table is a function of the maximum size of a table and thus a function that only depends on k. Let $\tau(k)$ denote the maximum of a table size and the running time to compute a table.

Now we show how we can implement the above algorithm in the stack framework. Let r denote the root of the tree T. We do a depth first traversal of the given tree decomposition T starting with root r. We start from the root, keep following the leftmost child until we reach a node without a leftmost child (or a leaf node). However, during this process if v is not a leaf node (that is, it has a leftmost child) we push a tuple $(v, \mathbb{T}_0^v)$, else we push $(v, \mathbb{T}_v)$ (that is when v is a leaf node). If we can't push something to the stack, we pop the stack and we know that its parent is now at the top of the stack. Let v' be the vertex we have popped and the tuple currently with respect to this be $(v', \mathbb{T}')$. We pop the stack and say the current tuple is $(v'', \mathbb{T}'')$. We know that v'' is the parent of v'. Now we compute a new table at v'' by using tables $\mathbb{T}'$ and $\mathbb{T}''$. Let the newly constructed table be $\mathbb{T}$. Then we push back $(v'', \mathbb{T})$. Then, we push the right sibling w of v' with $(w, \mathbb{T}_0^w)$. Finally, when the stack is empty then the last tuple we have popped corresponds to $(r, \mathbb{T}_r)$. Using this we can find the size of a largest (a smallest) subset of vertices satisfying the CMSO predicate ψ. To get the desired running time we apply Theorem 5.5 with $t = O(\tau(k))$, $g(n, t) = O(1)$, $h(n, t) = O(\tau(k))$ and $s(n) = O(1)$. $\square$

We can also prove a weighted version of MIN/MAX-CMSO$[\psi]$ problem, namely WEIGHTED-

MIN/MAX-CMSO[ψ]. In this problem, apart from a graph $G = (V, E)$ we are also given a weight function $w : V \to \mathbb{N}$ ($w : E \to \mathbb{N}$), and our objective is to find the value of a maximum weighted subset (or a minimum weighted subset) S such that $(G, S) \models \psi$. Here, the weight of a subset S, denoted by $w(S)$, is $\sum_{s \in S} w(s)$. To solve the WEIGHTED-MIN/MAX-CMSO[ψ] problem, the only changes we need to make in the proof of Theorem 5.13 are as follows. For, every $v \in V(T)$, $Z \subseteq \beta(v)$ and $c \in \mathcal{C}$ we had defined $\mathcal{F}(v, c, Z)$ and $\mathsf{value}(\mathbb{T}_v, c, Z)$. However, for WEIGHTED-MIN/MAX-CMSO[ψ] we will store the weight of a maximum weighted set (or a minimum weighted set) in $\mathcal{F}(v, c, Z)$. Similarly, we will define

$$\begin{aligned}\mathcal{V}(v,c,Z) \;=\; & \Big\{\mathsf{value}(\mathbb{T}_{i-1}, c_1, Z) + \mathsf{value}(\mathbb{T}_{v_i}, c_2, Z_1) - w(Z \cap Z_1) \;\Big| \\ & Z_1 \cap \beta(v) \subseteq Z, (c_1, c_2) \in \mathsf{Classes}(c)\Big\}.\end{aligned}$$

These two changes together imply the following theorem.

Theorem 5.18. *Let G be a graph given with a tree decomposition $(T = (V_T, E_T), \beta)$ of width k. Then* WEIGHTED-MIN/MAX-CMSO[ψ] *can be solved in time $O(\tau(k) \cdot n^{1+(1/\lg p)})$ time and $O(\tau(k) \cdot p \lg_p n)$ space algorithm, for any parameter $2 \leq p \leq n$. Here, $|V| = n$ and τ is a function of k alone.*

Next we obtain a theorem that not only outputs the weight of a value of a maximum weighted subset (or a minimum weighted subset) S such that $(G, S) \models \psi$, but also the set S. We call this version of the problem: CONSTRUCTIVE-WEIGHTED-MIN/MAX-CMSO[ψ].

Theorem 5.14 (Restated). *Let G be a graph given with a tree decomposition $(T = (V_T, E_T), \beta)$ of width k. Then* CONSTRUCTIVE-WEIGHTED-MIN/MAX-CMSO[ψ] *can be solved in time $O(\tau(k) \cdot n^{2+(2/\lg p)})$ time and $O(\tau(k) \cdot p \lg_p n)$ space algorithm, for any parameter $2 \leq p \leq n$. Here, $|V| = n$ and τ is a function of k alone.*

Proof. As before let $\mathbb{T}_i$ correspond to the table for G_i. Our algorithm is a standard recursive algorithm based on tracing pointers from root to a specific node of T. We call the algorithm to be Set-Output. It takes as input a node v, a table entry $\mathbb{T}_v[v, c, Z]$ (of course we also know the arguments c and Z required in the definition of $\mathbb{T}_v[v, c, Z]$) and a set $\sigma(v)$ and outputs a set of vertices $S \setminus \sigma(v)$ such that $S \in \mathcal{F}(v, c, Z)$ and $w(S) = \mathbb{T}_v[v, c, Z]$. We start the algorithm Set-Output on an appropriate arguments related to the root. The algorithm first outputs $Z \setminus \sigma(v)$ and then checks whether the given node v is a leaf node or not. If v is not a leaf node (that is, it has a left child) we call the algorithm Set-Output with appropriate instantiations. To find the desired arguments for the algorithm we do as follows. We assume that we are currently working on a node v of T and w is one of its children (for the case of root we explain later).

1. From Equation 5.2, we know that each table entry of $\mathbb{T}_v$ is a function of table entries stored at its children. In particular it is a function which takes exactly one table entry from each of its children. For the node v, given a table entry $\mathbb{T}_v[v, c, Z]$, stored at v, we say that $\mathbb{T}_w[w, c', Z']$ is an *accompanying entry* if this is the entry stored at $\mathbb{T}_w$ that has been used to compute $\mathbb{T}_v[v, c, Z]$.

2. We assume that the table entry $\mathbb{T}_v[v, c, Z]$ is at the top of the stack (We ensure this when we explain the details for the root as we start from the roort.). Given the table entry $\mathbb{T}_v[v, c, Z]$, we apply the algorithm described in Theorem 5.13 and for w we compute the accompanying table entry $\mathbb{T}_w[w, c', Z']$.

3. The set $\sigma(w)$ is computed by taking intersection of $\beta(v)$ and $\beta(w)$.

The label w along with $\sigma(w)$ is pushed on to the stack (which is needed when we need to compute the set for its children).

For the root r, using Theorem 5.13 we compute the table $\mathbb{T}_r$ and as a table entry we take the one that corresponds to the value of a maximum weighted (a minimum weighted) subset of vertices satisfying the CMSO predicate ψ. Finally, we call Set-Output on r using these arguments, and store the output along with r in a stack. Our algorithm is a stack algorithm where we need $O(\tau(k) \cdot n^{1+(1/\lg p)})$ time (since we use Theorem 5.13) for finding what we need to push and thus the running time follows by applying Theorem 5.5 with $t = O(\tau(k))$, $g(n,t) = O(1)$, $h(n,t) = O(\tau(k) \cdot n^{1+(1/\lg p)})$ and $s(n) = O(\tau(k) p \lg_p n)$. This concludes the proof. $\square$

5.6 Concluding remarks and open problems

We have shown that several optimization problems can be solved on trees and bounded treewidth graphs using logarithmic number of extra variables, in linear (and sometimes quadratic) time even when the input tree is given in a read-only memory. We achieve this by modifying the standard dynamic programming algorithms to use only a stack and using the recent stack compression routine to reduce space. Barba et. al. [18] also provide a stack compression scheme that can be used to reduce work space to $O(1)$ words provided the (full stack) algorithm satisfies what they called a "green" property. The standard dynamic programming algorithms we use are not "green". It would be interesting to see whether our approach can be extended to obtain algorithms using only $O(1)$ or even $o(\lg n)$ words. This would give an alternate $O(1)$ (words of) space version of Courcelle's theorem. Another open problem is whether this approach helps to give an alternate logarithmic space version of Bodlaender's theorem [24].

Bodlaender and Telle [27] give a divide and conquer strategy to find the optimum set in $O(n \lg n)$ time (against a naive $O(n^2)$ time) using $O(\lg n)$ words of extra space, when the tree of the tree-decomposition has a constant number of children for each node. Extending this (in the read-only memory model) to the case when each node has an arbitrary number of children, or to obtain a 'nice-tree decomposition' from a general tree-decomposition and implementing their approach in logarithmic (words of) space in read-only memory model, are challenging open problems. It would also be interesting to find other applications of the (generalized) stack compression framework.

Chapter 6

Two Frameworks for Designing In-place Graph Algorithms

6.1 Introduction

Read-only memory (ROM) model is one of the classical models of computation to study time-space tradeoffs of algorithms along with a few others that we have mentioned in the beginning of the thesis. One of the early classical results on the ROM model is that any sorting algorithm that uses $O(s)$ words of extra space requires $\Omega(n^2/s)$ comparisons for $\lg n \leq s \leq n/\lg n$ and the bound has also been recently matched by an algorithm. However, if we relax the model (from ROM), we do have sorting algorithms (say Heapsort) that can sort using $O(n \lg n)$ comparisons using $O(\lg n)$ bits of extra space, even keeping a permutation of the given input sequence at any point of time of the algorithm.

We address similar questions for graph algorithms in this chapter. We show that a simple natural relaxation of ROM model allows us to implement fundamental graph search methods like BFS and DFS more efficiently than in ROM. By simply allowing elements in the adjacency list of a vertex to be permuted, we show that, on an undirected connected graph G having n vertices and m edges, the vertices of G can be output in a

- DFS order using $O(\lg n)$ bits of extra space and $O(m^2/n)$ time if the graph is given in an adjacency list, and in $O(m^2 \lg n/n)$ time if the graph is given in an adjacency array;
- BFS order using $O(\lg n)$ bits of extra space and $O(m)$ time if all vertices have degree at least $2\lg n + 3$, in $O(n^2)$ time if there are no degree 2 vertices, and in $O(n^3)$ time otherwise.

Most of these results carry over to directed graphs too, with a slight degradation in running time. Thus we obtain similar bounds for *reachability* and *shortest path distance* (both for undirected and directed graphs). With a little more (but still polynomial) time, we can also output vertices in the *lex-DFS* order. As reachability in directed graphs (even in DAGs) and shortest path distance (even in undirected graphs) are NL-complete problems, and lex-DFS is P-complete, our results show that our model is probably more powerful than ROM.

En route, we introduce and develop algorithms for another relaxation of ROM where the adjacency lists of the vertices are circular lists and we can only modify the heads of the lists. Here we first show a linear time DFS implementation using $n + O(\lg n)$ bits. Improving the space further to only $O(\lg n)$ bits, we also obtain BFS and DFS albeit with a slightly slower running time. Some of these algorithms also translate to improved algorithms for DFS and its applications in ROM. Both the models we propose maintain the graph structure throughout the algorithm, only the order of vertices in the adjacency list changes.

In sharp contrast, for BFS and DFS, to the best of our knowledge, there are no algorithms in ROM that use even $O(n^{1-\epsilon})$ bits of space; in fact, implementing DFS using cn bits for $c < 1$ has been mentioned as an open problem. Furthermore, DFS (BFS) algorithms using $n + o(n)$ ($o(n)$) bits use Reingold's or Barnes et al's reachability algorithm and hence have high runtime.

All our algorithms are simple but quite subtle, and we believe that these models are practical enough to spur interest for other graph problems in these models.

6.1.1 In-place model for graph algorithms

Our main objective of this chapter is to initiate a systematic study of efficient (i.e., low degree polynomial running time) in-place (i.e., using $O(\lg n)$ bits of extra space) algorithms for graph problems. To the best of our knowledge, this has not been done in the literature before. Our first goal is to properly define models for the in-place graph algorithms. As in the case of standard in-place model, we need to ensure that the graph (adjacency) structure remains intact throughout the algorithm. Let $G = (V, E)$ be the input graph with $n = |V|$, $m = |E|$, and assume that the vertex set V of G is the set $V = \{1, 2, \cdots, n\}$. To describe these models, we assume that the input graph representation consists of two parts: (i) an array V of length n, where $V[i]$ stores a pointer to the adjacency list of vertex i, and (ii) a list of singly linked lists, where the i-th list consists of a singly linked list containing all the neighbors of vertex i with $V[i]$ pointing to the head of the list. In ROM model, we assume that both these components cannot be modified. In our relaxed models, we assume that one of these components can be modified in a limited way.

The most natural analogue of in-place model allows any two elements in the adjacency list of a vertex to be swapped (in constant time assuming that we have access to the nodes storing those elements in the singly linked list). The adjacency "structure" of the representation does not change; only the values stored can be swapped. (One may restrict this further to allow only elements in adjacent nodes to be swapped. Most of our algorithms work with this restriction.) We call it the implicit model inspired by the notion of *implicit data structures* [113]. We introduce and develop algorithms for another relaxed model which we call `rotate` model. In this model, we assume that only the pointers stored in the array V can be modified, that too in a limited way - to point to any node in the adjacency list, instead of always pointing to the first node. In space-efficient setting, since we do not have additional space to store a pointer to the beginning of the adjacency list explicitly, we assume that the second component of the graph representation consists of a list of circular linked lists (instead of singly linked lists) – i.e., the last node in the adjacency list of each vertex points to the first node (instead of storing a null pointer). See the figure below to get a better visual description. We call the element pointed to by the pointer as the front of the list, and a unit cost rotate operation changes the element pointed to by the pointer to the next element in the list.

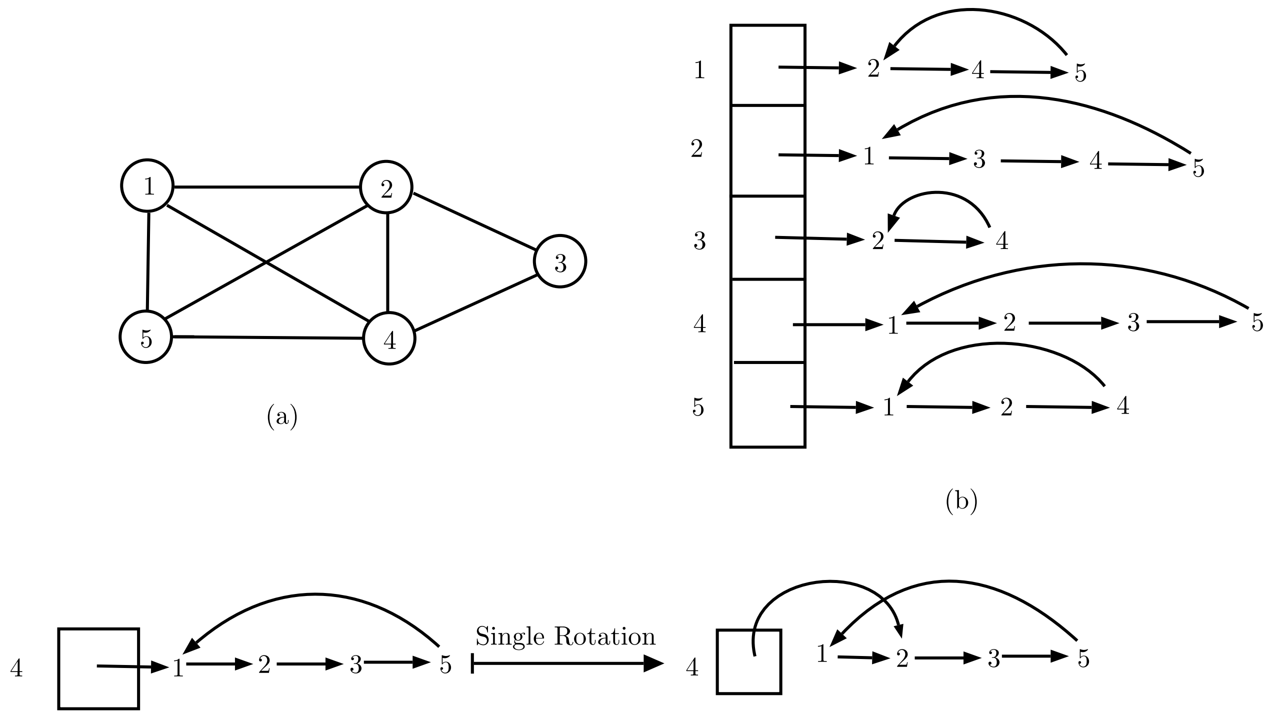

Figure 6.1: (a) An undirected graph G with 5 vertices and 8 edges. (b) A circular list representation of G. To avoid cluttering the picture, we draw the vertices and the pointers to the next node separately as opposed to a single node having two different fields in the circular list. (c) An illustration of a single clockwise rotation in the circular list of vertex 4.

Thus the `rotate` model corresponds to keeping the adjacency lists in read-only memory and allowing (limited) updates on the pointer array that points to these lists. And, the implicit model corresponds to the reverse case, where we keep the pointer array in read-only memory and allow swaps on the adjacency lists/arrays. A third alternative especially for the implicit model is to assume that the input graph is represented as an adjacency array, i.e., adjacency lists are stored as arrays instead of singly linked lists (see [41, 69, 99] for some results using this model); and we allow here that any two elements in the adjacency array can be swapped. In this model, some of our algorithms have improved performance in time.

As is standard in the design of space-efficient algorithms, while working with directed graphs, we assume that the graphs are given as in/out (circular) adjacency lists i.e., for a vertex v, we have the (circular) lists of both in-neighbors and out-neighbors of v. For the workspace, we assume the standard word RAM model of computation where the machine consists of words of size w in $\Omega(\lg n)$ bits and any logical, arithmetic and bitwise operation involving a constant number of words takes $O(1)$ time. We count space in terms of number of bits used by the algorithm other than the input. Before getting into the technical part of the chapter, we set up a few notations. In what follows, by a path of length d, we refer to a simple path on d edges. By $deg(x)$ we refer to the degree of the vertex x. In directed graphs, it should be clear from the context whether that denotes out-degree or in-degree. By a BFS or DFS traversal of the input graph G, as in the previous chapters, we refer to outputting the vertices of G in the BFS or DFS ordering, i.e., in the order in which the vertices are visited for the first time.

6.1.2 The complexity of BFS and DFS

In this chapter, we mainly focus on designing in-place algorithms for two of the most basic and fundamental graph search methods. As they are also backbone to so many other graph algorithms, many corollaries follow. For the DFS problem, there have been two versions studied in the literature. In the *lexicographically smallest DFS* or *lex-DFS* problem, when DFS looks for an unvisited vertex to visit in an adjacency list, it picks the "first" unvisited vertex where the "first" is with respect to the appearance order in the adjacency list. The resulting DFS tree will be unique. In contrast to lex-DFS, an algorithm that outputs *some* DFS numbering of a given graph, treats an adjacency list as a set, ignoring the order of appearance of vertices in it, and outputs a vertex ordering T such that there exists *some* adjacency ordering R such that T is the DFS numbering with respect to R. We say that such a DFS algorithm performs *general-DFS*. Reif [120] has shown that lex-DFS is P-complete (with respect to log-space reductions) implying that a logspace algorithm for lex-DFS results in the collapse of complexity classes P and L. Anderson et al. [5] have shown that even computing the leftmost root-to-leaf path of the lex-DFS tree is P-complete. For many years, these results seemed to imply that the general-DFS problem, that is, the computation of any DFS tree is also inherently sequential. However, Aggarwal et al. [1, 2] proved that the general-DFS problem can be solved much more efficiently, and it is in RNC. Whether the general-DFS problem is in NC is still open.

6.1.3 Our main results and the organization of this chapter

Rotate Model: For DFS, in the rotate model, we show the following in Sections 6.2.1, 6.2.3 and 6.2.5.

Theorem 6.1. *Let G be a directed or an undirected graph, and $\ell \leq n$ be the maximum depth of the DFS tree starting at a source vertex s. Then in the* rotate *model, the vertices of G can be output in*

(a) *the lex-DFS order in $O(m + n)$ time using $n \lg 3 + O(\lg^2 n)$ bits,*

(b) *a general-DFS order in $O(m + n)$ time using $n + O(\lg n)$ bits, and*

(c) *a general-DFS order in $O(m^2/n+m\ell)$ time for an undirected graph and in $O(m(n+$*

ℓ^2)) time for directed graphs using $O(\lg n)$ bits. For this algorithm, we assume that s can reach all other vertices.

This is followed by the BFS algorithms where, in the rotate model, we show the following in Sections 6.3.1 and 6.3.2.

Theorem 6.2. *Let G be a directed or an undirected graph, and ℓ be the depth of the BFS tree starting at the source vertex s. Then in the* rotate *model, the vertices of G can be output in a BFS order in*

1. *$O(m + n\ell^2)$ time using $n + O(\lg n)$ bits, and*
2. *$O(m\ell + n\ell^2)$ time using $O(\lg n)$ bits. Here we assume that the source vertex s can reach all other vertices.*

Implicit Model: In the implicit model, we obtain polynomial time implementations for lex-DFS and general-DFS using $O(\lg n)$ bits. For lex-DFS, this is conjectured to be unlikely in ROM as the problem is P-complete [120]. In particular, we show the following in Section 6.5.1.

Theorem 6.3. *Let G be a directed or an undirected graph with a source vertex s and $\ell \leq n$ be the maximum depth of the DFS tree starting at s that can reach all other vertices. Then in the* implicit *model, using $O(\lg n)$ bits the vertices of G can be output in*

(a) the lex-DFS order in $O(m^3/n^2 + \ell m^2/n)$ time if G is given in adjacency list and in $O(m^2 \lg n/n)$ time if G is given in adjacency array for undirected graphs. For directed graphs our algorithm takes $O(m^2(n+\ell^2)/n)$ time if G is given in adjacency list and $O(m \lg n(n + \ell^2))$ time if G is given in adjacency array;

(b) a general-DFS traversal order in $O(m^2/n)$ time if the input graph G is given in an adjacency list and in $O(m^2(\lg n)/n + m\ell \lg n))$ time if it is given in an adjacency array.

In the implicit model, we can match the runtime of BFS from rotate model, and do better in some special classes of graphs. In particular, we show the following in Section 6.6
.

Theorem 6.4. *Let G be a directed or an undirected graph with a source vertex that can reach all other vertices by a distance of at most ℓ. Then in the* implicit *model, using $O(\lg n)$ bits the vertices of G can be output in a BFS order in*

1. *$O(m + n\ell^2)$ time;*
2. *the runtime can be improved to $O(m + n\ell)$ time if there are no degree 2 vertices;*
3. *the runtime can be improved to $O(m)$ if the degree of every vertex is at least $2 \lg n+3$.*

In sharp contrast, for space efficient algorithms for DFS in ROM, the landscape looks markedly different. To the best of our knowledge, there are no DFS algorithms in general graphs in ROM that use $O(n^{1-\epsilon})$ bits. In fact, an implementation of DFS taking cn bits for $c < 1$ has been proposed as an open problem by Asano et al. [9]. Similar to DFS, to the best of our knowledge, there are no polynomial time BFS algorithms in ROM that use even $O(n^{1-\epsilon})$ bits. On the other hand, we don't hope to have a BFS algorithm (for both undirected and directed graphs) using $O(\lg n)$ bits in ROM as the problem is NL-complete [7].

Moving on from DFS and BFS, we also study the problem of reporting a minimum spanning tree (MST) of a given undirected connected graph G. We show the following result in Section 6.7.

Theorem 6.5. *A minimum spanning forest of a given undirected weighted graph G can be found using $O(\lg n)$ bits and in*

1. *$O(mn)$ time in the* rotate *model,*
2. *$O(mn^2)$ time in the* implicit *model if G is given in an adjacency list, and*

3. $O(mn \lg n)$ time in the implicit *model when G is represented in an adjacency array.*

Note that by the results of [121, 119], we already know logspace algorithms for MST in ROM but again the drawback of those algorithms is high time complexity. On the other hand, our algorithms have small polynomial running time, simplicity and we believe it would be easy to use in practice.

6.1.4 Techniques

Our implementations follow (variations of) the classical algorithms for BFS and DFS that use three colors (white, gray and black), but avoid the use of stack (for DFS) and queue (for BFS). In the rotate model, when a node is visited for the first time, we use the rotate operation to move the parent or a (typically the currently explored) child to the beginning of the list to help navigate through the tree during the forward or the backtracking step. This avoids the need for stack or queue. Algorithms using $O(\lg n)$ bits (which don't even have space to store the color array) use the rotate operation in a non-trivial way to move elements within the lists to determine the color of the vertices as well.

In the implicit model, we use the classical *bit encoding* trick used in the development of implicit data structures [113]. We encode one (or two) bit(s) using a sequence of two (or three respectively) distinct numbers. To encode a single bit b using two distinct values x and y with $x < y$, we store the sequence x, y if $b = 0$, and y, x otherwise. Similarly, permuting three distinct values x, y, z with $x < y < z$, we can represent six combinations. We can choose any of the four combinations to represent up to 4 colors (i.e. two bits). Generalizing this further, we can encode a pointer taking $\lg n$ bits using $2 \lg n$ distinct elements where reading or updating a bit takes constant time, and reading or updating a pointer takes $O(\lg n)$ time. This also is the reason for the requirement of vertices with (high) degree at least 3 or $2 \lg n + 3$ for faster algorithms, which will become clear in the description of the algorithms.

6.1.5 Simulations across models

In general, an algorithm implemented in the rotate model can be implemented

- in the implicit model with a slow-down of a factor of at most n in the runtime, as a rotate operation can be implemented in time proportional to the (maximum) degree of a vertex;
- in read-only memory using an extra $O(n \lg(m/n))$ bits by storing a pointer in each of the arrays in the adjacency array representation, which can be updated in constant time.

These are discussed in detail in Section 6.4 and in Section 6.8. Similarly any algorithm in ROM can be implemented in the rotate and implicit models without any additional time or space. The implicit model is the most general model and not surprisingly it is not easy to simulate algorithms of that model, in general, in ROM or in rotate models without substantial loss of time and/or space.

6.1.6 Consequences of our BFS and DFS results

There are many interesting and surprising consequences of our results for BFS and DFS in both the rotate and implicit model. In what follows, we mention a few of them.

- For *directed st-reachability*, as mentioned previously, the most space efficient polynomial time algorithm [19] uses $n/2^{\Theta(\sqrt{\lg n})}$ bits. In sharp contrast, we obtain efficient (timewise) logspace algorithms for this problem in both the rotate and implicit models (as a corollary of our directed graph DFS/BFS results). In terms of workspace this is exponentially better than the best known polynomial time algorithm [19] for this problem in ROM. For us, this provides one of the main motivations to study this model. A somewhat incomparable result obtained recently by Buhrman et al. [35, 101] where they designed an algorithm for *directed st-reachability* on catalytic Turing machines in space $O(\lg n)$ with catalytic space $O(n^2 \lg n)$ and time $O(n^9)$.

- Problems like *directed st-reachability* [7], *distance* [126] which asks whether a given G (directed, undirected or even directed acyclic) contains a path of length atmost k from s to t, are NL-complete i.e., no deterministic logspace algorithm is known for these problems. But in our (both rotate and $\mathsf{implicit}$) models, we design logspace algorithms for them. These result show that probably both our models with logspace are stronger than NL.

- The lex-DFS problem (both in undirected and directed graphs) is P-complete [120], and thus polylogarithmic space algorithms are unlikely to exist. But we show an $O(\lg n)$ space algorithm in the $\mathsf{implicit}$ model for lex-DFS. This implies that, probably $\mathsf{implicit}$ model is even more powerful than rotate model. It could even be possible that every problem in P can be computed using logspace in $\mathsf{implicit}$ model. A result of somewhat similar flavor is obtained recently Buhrman et al. [35, 101] where they showed that any function in TC^1 can be computed using catalytic logspace, i.e., $\mathsf{TC}^1 \subseteq \mathsf{CSPACE}(\lg \mathsf{n})$. Note that TC^1 contains L, NL and even other classes that are conjectured to be different from L.

- Our bounds for BFS and DFS in rotate and $\mathsf{implicit}$ model immediately imply (with some care) similar bounds, that are improvement over the best space bounds known so far in ROM, for many applications of DFS/BFS. Moreover, as described before, any algorithm in rotate model can be implemented in ROM using extra $O(n \lg(m/n))$ bits. Thus, our linear time DFS algorithm in rotate model can be implemented in ROM using $O(n \lg(m/n))$ bits, matching the bound of [41] for DFS. Using this DFS implementation, we can obtain improved space efficient algorithms for various applications of DFS in ROM. This is discussed in Section 6.8.

- For a large number of NP-hard graph problems, the best algorithms in ROM run in exponential time and polynomial space. We show that using just logarithmic amount of space, albeit using exponential time, we can design algorithms for those NP-hard problems in both of our models under some restrictions. This gives an exponential improvement over the ROM space bounds for these problems. In constrast, note that, no NP-hard problem can be solved in the ROM model using $O(\lg n)$ bits unless $\mathsf{P}{=}\mathsf{NP}$. This is described in Section 6.9.

6.2 DFS Algorithms in the rotate model

In this section, we describe our space-efficient algorithms for DFS in the rotate model proving Theorem 6.1. We restate the theorem statement here again for reader's convenience.

Theorem 6.1 (Restated) Let G be a directed or an undirected graph, and $\ell \leq n$ be the maximum depth of the DFS tree starting at a source vertex s. Then in the rotate model, the vertices of G can be output in

(a) the lex-DFS order in $O(m+n)$ time using $n \lg 3 + O(\lg^2 n)$ bits,

(b) a general-DFS order in $O(m+n)$ time using $n + O(\lg n)$ bits, and

(c) a general-DFS order in $O(m^2/n + m\ell)$ time for an undirected graph and in $O(m(n+\ell^2))$ time for directed graphs using $O(\lg n)$ bits. For this algorithm, we assume that s can reach all other vertices.

6.2.1 Lex-DFS using $n \lg 3 + O(\lg^2 n)$ bits for undirected graphs

We begin by describing our algorithm for undirected graphs, and later mention the changes required for directed graphs. Let us recollect how DFS works briefly again before we start with the technical details of the algorithms. In the normal exploration of DFS (see for example, Cormen et al. [51]) we use three colors. Every vertex v is white initially meaning that it has not been discovered yet, becomes gray when DFS discovers v for the first time, and is colored black when it is finished i.e., all its neighbors have been explored completely.

We maintain a color array C of length n that stores the color of each vertex at any point in the algorithm. We start DFS at the starting vertex, say s, changing its color from white to gray in the color array C. Then we scan the adjacency list of s to find the first white neighbor, say w. We keep rotating the list to bring w to the front of s's adjacency list (as the one pointed to by the head $V[s]$), color w gray in the color array C and proceed to the next step (i.e. to explore w's adjacency list). This is the first *forward* step of the algorithm. In general, at any step during the execution of the algorithm, whenever we arrive at a gray vertex u (including the case when u's color is changed from white to gray in the current step), we scan u's adjacency list to find the first white vertex. (i) If we find such a vertex, say v, then we rotate u's list to make v as the first element, and change the color of v to gray. (ii) If we do not find any white vertex, then we change the color of u to black, and *backtrack* to its parent. To identify u's parent, we use the following lemma.

Lemma 6.6. *Suppose w is a node that just became* black*. Then its parent p is the unique vertex in w's adjacency list which is (a)* gray *and (b) whose current adjacency list has w in the first position.*

Proof. Among all the neighbors of w, some vertices are w's children in the DFS tree, and the rest of them are w's ancestors, and among the ancestors, exactly one vertex is w's parent in the DFS tree. All the ancestors should have their currently explored (gray) child at the first position in their adjacency list; and this current child would be different from w for all the ancestors except p (as w was discovered from p). So, the second condition is violated for them. All of w's children have been fully processed earlier and have been colored black, and hence the first condition is violated for them. Observe that, if w has a child, say k, which is a leaf in the DFS tree, it might happen that k also has w at the first position in its current adjacency list, but, fortunately, k is black while scanning w's list. So for such vertices, the first condition gets violated. Only for w's parent, which is p here, both the conditions are satisfied. □

So, the parent can be found by by scanning the w's list, to find a neighbor p that is colored gray such that the first element in p's list is u. This completes the description of the backtracking step. Once we backtrack to p, we find the next white vertex (as in the forward step) and continue until all the vertices of G are explored. Other than some constant number of variables, clearly the space usage is only for storing the color array C. Since C is of length n where each element has 3 possible values, C can be encoded using $n \lg 3 + O(\lg^2 n)$ bits, so that the i-th element in C can be read and updated in $O(1)$ time [60]. So overall space required is $n \lg 3 + O(\lg^2 n)$ bits. As the algorithm systematically brings a white vertex to the front, makes it gray, and moves it to the end after it becomes black, at most two full rotations of each of the list may happen (the second one to determine that there are no more white vertices) resulting in a linear time lex-DFS algorithm.

6.2.2 Lex-DFS using $n \lg 3 + O(\lg^2 n)$ bits for directed graphs

Recall that we have access to both the in-adjacency and the out-adjacency lists for each vertex w in a directed graph G, hence we can use these lists separately for performing two steps of DFS. I.e., out-adjacency list is used for the exploration of DFS in the forward direction and the in-adjacency list is used for finding parent of a node during the backtracking step. We provide the details below. Similar to our algorithm for undirected graphs, in the forward direction, we scan the out-neighbor list of w to find the next white neighbor and proceed. Once the out-neighbor list of w is fully processed, we need to backtrack from w. Towards that we first have to identify w's parent. In order to do so we use the following lemma whose proof follows along the same lines as the Lemma 6.6 above. Hence we omit the proof.

Lemma 6.7. *Suppose w is a node that just became* black*. Then its parent p is the unique vertex in w's in-adjacency list which is (a) gray and (b) whose current out-adjacency list has w in the first position.*

Once we figure out w's parent p, DFS backtracks to p, finds the next white neighbor (as done in the forward step) and continues until all the vertices are exhausted. It is clear that this procedure performs lex-DFS on a directed graph G correctly in linear time, and this completes the proof of the first part of Theorem 6.1.

6.2.3 General-DFS using $n + O(\lg n)$ bits for undirected graphs

To improve the space further, we replace the color array C with a bit array $visited[1, \ldots, n]$ which stores a 0 for an unvisited vertex (white), and a 1 for a visited vertex (gray or black).

First we need a test like that in the statement of Lemma 6.6 without the distinction of gray and black vertices to find the parent of a node. Due to the invariant we have maintained, every internal vertex of the DFS tree will point to (i.e. have as first element in its list) its last child. So the nodes that could potentially have a node w in its first position are its parent, and *any* leaf vertex. Hence we modify the forward step in the following way.

Whenever we visit an unvisited vertex v for the first time from another vertex u (hence, u is the parent of v in the DFS tree and u's list has v in the first position), we, as before, mark v as visited and in addition to that, we also rotate v's list to bring u, to the front (during this rotation, we do not mark any intermediate nodes as visited). Then we continue as before (by finding the first unvisited vertex and bringing it to the front) in the forward step. Now the following invariants are easy to see and are useful.

Invariants: During the exploration of DFS, in the (partial) DFS tree

1. any internal vertex has the first element in its list as its current last child; and
2. for any leaf vertex of the DFS tree, the first element in its list is its parent.

The first invariant is easy to see as we always keep the current explored vertex (child) as the first element in the list. For leaves, the first time we encounter them, we make its parent as the first element in the forward direction. Then we discover that it has no unvisited vertices in its list, and so we make a full rotation and bring the parent to the front again. The following lemma provides a test to find the parent of a node.

Lemma 6.8. *Suppose w is a node that has just become* black*. Then its parent p is the* **first** *vertex x in w's adjacency list which is marked* 1 *in the visited array, and whose current adjacency list has w in the first position.*

Proof. From the invariants we observed, the nodes that can potentially have w in the first position of their lists are its parent and its children that happen to be leaves. But in w's list, as we began the exploration of its neighbors starting from its parent, its parent will appear first before its children. Hence the first node in w's list which has w in the first position must be its parent. □

Once we backtrack to p, we find the next white vertex, and continue until all the vertices of G are explored. Overall this procedure takes linear time. As we rotate the list to bring the parent of a node, before exploring its white neighbors, we are not guaranteed to explore the first white vertex in its original list, and hence we loose the lexicographic property.

6.2.4 General-DFS using $n + O(\lg n)$ bits for directed graphs

For performing DFS in directed graphs using $n + O(\lg n)$ bits, we don't even need to apply the modifications as we did for the undirected graphs during the forward direction, and we can essentially use the same forward step idea as used for lex-DFS in undirected graphs of Section 6.2.1. We provide the details below. When we arrive at a previously unvisited vertex v from the vertex u (hence u is the parent of v in the DFS tree), we rotate the in-neighbor list of v to bring u to the front and u stays there during the entire course of the exploration. Thus we maintain the invariant that for any visited node v, the first element in its in-neighbor list is its parent in the DFS tree. Now the algorithm scans v's adjacency list to find its unvisited neighbor. (i) If we find such a vertex, say w, then we rotate v's list to make w as the first element, and mark w visited. (ii) If we do not find any such unvisited neighbor of v, then DFS needs to *backtrack* to its parent. From the invariant we maintain in the in-neighbor list of every visited vertex, this is easy. All we need to do is to see the first entry in v's in-neighbor list to retrieve its parent u and then continue from u. Overall this procedure takes linear time. Upon closer inspection, it can be seen that, as we are not modifying the ordering of the vertices in the out-neighbor lists in the forward direction (in contrast with the undirected graph algorithm of Section 6.2.3), this procedure actually traverses the directed graph G in lex-DFS ordering. This completes the proof of the second part of Theorem 6.1.

6.2.5 General-DFS using $O(\lg n)$ bits for undirected graphs

Now to decrease the space to $O(\lg n)$, we dispense with the color/visited array, and give tests to determine white, gray and black vertices. For now, assume that we can determine the color of a vertex. The forward step is almost the same as before except performing the update in the color array. I.e., whenever we visit a white vertex v for the first time from another vertex u (hence u is the parent of v), we rotate vs list to bring u to the front.

Then we continue to find the first white vertex to explore. We maintain the following invariants. (i) any gray vertex has the first element in its list as its last child in the (partial) DFS tree; (ii) any black vertex has its parent as the first element in its list. We also store the depth of the current node in a variable d, which is incremented by 1 every time we discover a white vertex and decremented by 1 whenever we backtrack. We maintain the maximum depth the DFS has attained using a variable max. At a generic step during the execution of the algorithm, assume that we are at a vertex x's list, let p be x's parent and let y be a vertex in x's list. We need to determine the color of y and continue the DFS based on the color of y. We use the following characterization.

Lemma 6.9. *Suppose the DFS has explored starting from a source vertex s, up to a vertex x at level d. Let p be x's parent. Note that both s and x are* gray *in the normal coloring procedure. Let max be the maximum level of any vertex in the partial DFS exploration. Let y be a vertex in x's list. Then,*

1. *y is* gray *(i.e. (x, y) is a back edge, and y is an ancestor of x) if and only if we can reach y from s following through the* gray *child (which is in the first location of each of the lists of* gray *nodes) path in at most d steps.*
2. *y is* black *(i.e. (x, y) is a back edge, and x is an ancestor of y) if and only if*
    - *there is a path P of length at most $(max - d)$ from y to x (obtained by following through the first elements of the lists of every vertex in the path, starting from y), and*
    - *let z be the node before x in the path P. The node z appears after p in x's list.*
3. *y is* white *if y is not* gray *or* black.

Proof. The test for gray and white vertices is easy to see. The vertex y is black implies that y is a descendant of x in the partially explored DFS tree. This means that there is a path of length at most $(max - d)$ (obtained by following the parent which is in the first element of the adjacency list) from y to x through an already explored child z . By the way we process x's list, we first bring the parent to the front of the list, and then explore the nodes in sequence, and hence z, the explored neighbor of x must appear after p in x's list. Conversely, the unexplored neighbors of x appear before p in x's list. □

Now, if we use the above claim to test for colors of vertices, testing for gray takes at most d steps. Testing for black takes at most $(max - d)$ steps to find the path, and at most $deg(x)$ steps to determine whether p appears before. Thus for each vertex in x's list, we spend time proportional to $max + deg(x)$. So, the overall runtime of the algorithm is $\sum_{v \in V} deg(v)(deg(v) + \ell) = O(m^2/n + m\ell)$, where ℓ is the maximum depth of DFS tree. Maintaining the invariants for the gray and black vertices are also straightforward. We provide details for directed graphs next.

6.2.6 General-DFS using $O(\lg n)$ bits for directed graphs

We describe our $O(\lg n)$ bits algorithm for directed graphs in the rotate model. More specifically, we give a DFS algorithm to output all vertices reachable by a directed path from the source vertex s. If we assume that s can reach all vertices, we get to output all vertices. In the preprocessing step, the algorithm spends $O(m)$ time to bring the minimum valued neighbor (denote it by min) in the out-neighbor list of every vertex by rotation (hence we loose the lexicographic DFS property). For now assume that we can determine the color of a vertex. Given this, in the forward direction, when DFS arrives at a previously unvisited vertex v from the vertex u (hence u is the parent of v in the DFS tree), we rotate the in-neighbor list of v to bring u to the front and u stays there during the entire course of the exploration. Also in u's out-neighbor list, v is made the first location. Hence we maintain the following invariants.

Invariants: During the exploration of DFS, in the (partial) DFS tree

1. gray vertices have their current last child in the first location of their out-neighbor

lists;

2. all the visited (i.e., gray and black) vertices have their parent in the first location of their in-neighbor lists.

We also keep track of the depth of the current node (i.e., the last gray vertex in the gray path of the DFS tree) in a variable d, which, as before, is incremented by 1 every time DFS visits a white vertex and decremented by 1 whenever DFS backtracks. We also store the maximum depth the DFS tree has attained so far in a variable max. At a generic step during the execution of the algorithm, assume that we are at a vertex x's list, let p be x's parent (which can be found from the way x is visited by a forward or a backtracking step using the invariants being maintained) and let y be a vertex in x's list. We need to determine the color of y and continue the DFS based on the color of y and maintain the invariants. We use the following characterization.

Lemma 6.10. *Suppose the DFS has explored starting from a source vertex s up to a vertex x at level d. Let p be x's parent. Note that both s and x are* gray *in the normal coloring procedure. Let max be the maximum level of any vertex in the partial DFS exploration. Let y be a vertex in x's list. Then,*

1. *y is* gray *(i.e., (x, y) is a back edge and y is an ancestor of x) if and only if we can reach y from s following the* gray *child (which is in the first location of each of the out-neighbor lists of* gray *nodes) path in at most d steps.*

2. *y is* black *if and only if any of the following happens.*

    - *There is a path P of length at most $(max - d)$ from y to x obtained by following the first elements of the in-neighbor lists of every vertex in the path P starting from y. This happens when (x, y) is a forward edge, and x is an ancestor of y.*

    - *There is a path P of length at most max from y to a gray vertex $z \neq x$ (obtained by following through the first elements of the in-neighbor lists of every vertex*

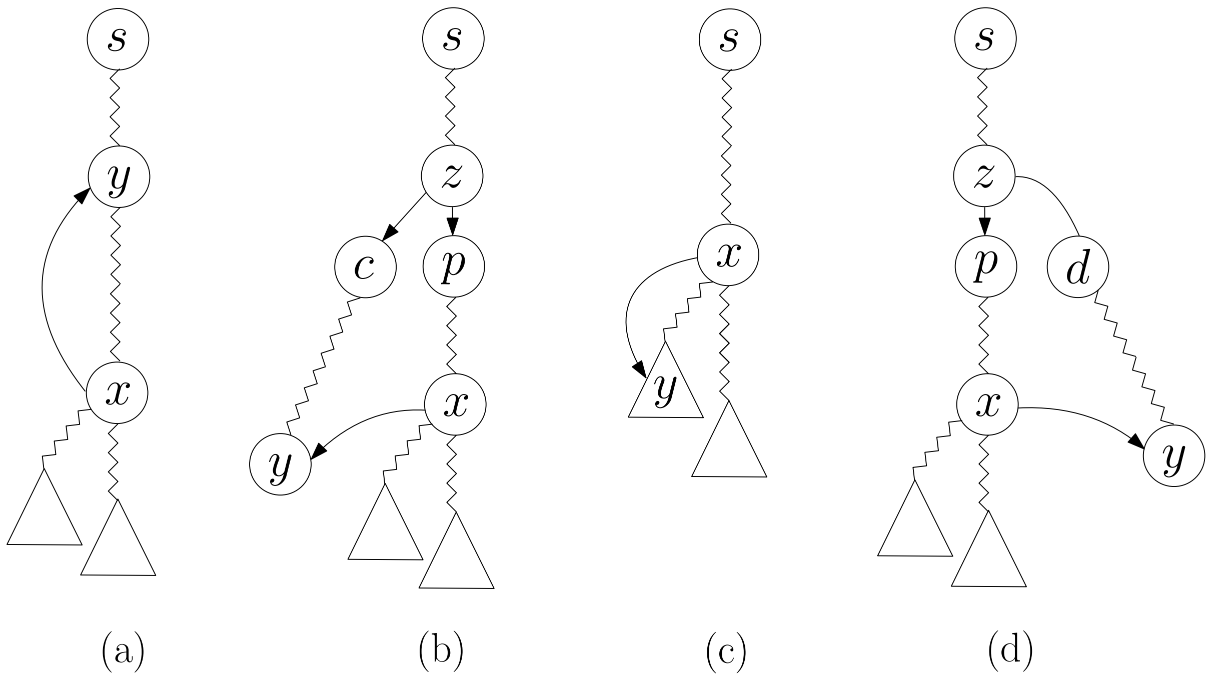

Figure 6.2: Illustration of the different cases of the possible positions of the vertex y when DFS considers the directed edge (x, y) at some intermediate step. Suppose the root of the DFS tree is the vertex s and the curvy path starting from s and going straight below through x is the current gray path in the DFS tree. Intuitively all the vertices on the left hand side of the path are black, and right hand side are white and yet to be explored. From left to the right are cases when (x, y) is (a) back edge, (b) cross edge, (c) forward edge, and (d) tree edge.

starting from y) which is the first gray vertex in the path. Let c be the node before z in the path P, then c must appear after min in z's list (this happens when (x, y) is a cross edge).

3. *The vertex y is* white *if it is not* black *or* gray*(i.e., (x, y) is the next tree edge with x being the parent of y in the DFS tree).*

Proof. See Figure 5.2 for a picture of all the cases. The test for gray and white vertices is easy to see.

From a vertex x, there could be two types of outgoing edges to a black vertex y. When (x, y) is a forward edge, y is a descendant of x and hence there must exist a path P of length at most $(max - d)$ (obtained by following the parent which is in the first location of the in-neighbor list of every vertex in P, starting from y) from y to x through an already explored child t of x. In the other case, when (x, y) is a cross edge, y has already been discovered and explored completely before DFS reaches to x. Hence there must exist a gray ancestor z of x (z could be x) such that y belongs to the subtree rooted at z in the DFS tree. Thus, from y's in-neighbor list if we follow the path starting with y's parent for at most max steps, we must hit the gray path and the first vertex we come across is z. Let c be the node before z in the path. By the way we process z's out-neighbor list, we first bring the min to the front of the list, and then explore the other neighbor nodes in sequence, and hence c, the explored neighbor of z must appear after min in z's list.

For the converse, suppose y is a white vertex. Either we never reach a gray vertex in max steps (and we will correctly determin its color in this case) or we reach x or x's ancestor z from y following through the (spurious) first vertices of the in-neighbor list of a white vertex y. Note that the parent of a white vertex is white or gray and it can never be black. Hence z's child in the path is white. Hence that child will appear before min in z's list. □

Given the above test, if y turns out to be white, the edge (x, y) is added to the DFS tree, and y now becomes the current gray vertex. Note that maintaining the invariants are straightforward. Also, when any vertex v has been completely explored by DFS, we retrieve its parent from the in-neighbor list to complete the backtracking step. This procedure is continued until DFS comes back to the source vertex s. We stop at this point. This is because, note that, our proof breaks down in the case when DFS in a directed graph produces a forest and some cross edges go across these trees. In that case, if we follow the path starting from y's parent, we would reach the root of the DFS tree containing y and this is different from the tree where x belongs to. As we cannot maintain informations regarding all such roots of these previously explored DFS trees, we might spuriously conclude that y is unvisited even though it is not the case. Thus our algorithm produces the DFS tree containing only the vertices reachable from the source vertex s via some directed path in G. We leave open the case for desigining such logspace algorithm for the general directed graphs.

Given the above lemma, if y turns out to be white, the edge (x, y) is added to the DFS tree, and y now becomes the current gray vertex. Note that maintaining the invariants are easy. When any vertex v has been completely explored by DFS, we retrieve its parent from the in-neighbor list to complete the backtracking step. This procedure is continued until DFS comes back to the source vertex s. We stop at this point and we have outputted all vertices reachable from s.

To analyse the running time of our algorithm observe that testing for gray takes at most d steps. Testing for black takes, in the worst case, at most max steps to find the path, and at each step of the path, we take d time to test whether the new node is gray. Once we reach a gray vertex, we spend at most $deg(z)$ steps to determine whether c appears before min. Thus for each vertex in x's list, we spend time proportional to $d + (d.max) + deg(z)$ time. As z (which is independent of x) can have degree at most n. Thus, the overall runtime of the algorithm is $\sum_{v \in V} deg(v)(d + d\ell + n)$ which is $O(m(n + (1 + \ell)\ell)$ which is $O(m(n + \ell^2))$, where ℓ is the maximum depth of DFS tree.

6.3 BFS Algorithms in the rotate model

In this section, we describe our space-efficient algorithms for BFS in the rotate model proving Theorem 6.2. We restate the theorem statement here again for reader's convenience.

Theorem 6.2 (Restated) Let G be a directed or an undirected graph, and ℓ be the depth of the BFS tree starting at the source vertex s. Then in the rotate model, the vertices of G can be output in a BFS order in

(a) $O(m + n\ell^2)$ time using $n + O(\lg n)$ bits, and

(b) $O(m\ell + n\ell^2)$ time using $O(\lg n)$ bits. Here we assume that the source vertex s can reach all other vertices.

6.3.1 BFS using $n + O(\lg n)$ bits

It is well-known that BFS actually computes the shortest path lengths in unweighted undirected or directed graph G from a given source vertex $s \in V$ to every vertex $v \in V$ that is reachable from s. I.e., if a vertex v belongs to the d-th level in the BFS tree (assuming the root s is at zero-th level), then we know that the length of the shortest path from s to v is d. We use this crucially to design our BFS algorithms. We use a bit array $\mathit{visited}[1, \cdots, n]$ that stores a 0 for an unvisited vertex, and 1 for a visited vertex. We also maintain a counter *dist* which stores the level of the vertex that is currently being explored in the BFS algorithm.

We start by setting $\mathit{visited}[s] = 1$, and initializing the counter *dist* to 0. At the next step, for every unvisited neighbor v of s, we rotate their adjacency list so that s appears as the first element in v's list, set $\mathit{visited}[v] = 1$, and output v. This step ensures that for each visited vertex, its parent is at the front of its adjacency list. We refer to this front element in the adjacency list of a visited vertex as its *parent pointer*. (Also, once we set the parent pointer for a vertex, we will not rotate its adjacency list in the remaining part of the algorithm.) Once the root s's list is fully processed as above, the *dist* is incremented to 1. The next step in the algorithm is to find all the vertices in the first level and mark all their unvisited neighbors as visited. As we haven't stored these vertices (in the first level), the challenge is to find them first. We use the following claim, to find the level number of a visited vertex. The proof easily follows from the fact that the parent pointers are set for all the visited vertices, and also that all the ancestors of a visited vertex are also visited.

Claim 6.11. *If the BFS has been explored till distance d from the source, then for any*

$k \leq d$, a vertex x marked visited is in level k if and only if we can reach the source s in exactly k steps by following through their parent pointers. Thus determining if a visited vertex is in level d takes at most d steps.

So now we continue the BFS by scanning through the vertices, finding those vertices in level d (using the above claim by spending d steps for each vertex), and marking their unvisited neighbors visited, and making in their adjacency lists, their parent vertex as the first element, and incrementing *dist*. We stop our algorithm when we discover no new unvisited vertex while exploring any level. The correctness of the procedure and the space used by the algorithm are clear. To analyze the runtime, note that the time spent at level d is $nd + \sum_{i \in V(d)} deg(i)$ where $V(d)$ is the set of vertices in level d and $deg(i)$ is the degree of vertex i. Summing over all levels, we get a runtime $O(m + n\ell^2)$, where ℓ is the depth of the BFS tree.
To handle directed graphs, we follow the outneighbor list as we go down, and we set up the parent at the first position in the in-neighbor list of every vertex v. To verify if v belongs to the d-th level, we take d steps from v by following the parent pointers in the in-neighbor lists of the (visited) vertices along the path, and check if we reach s at the end. This proves the first part of Theorem 6.2.

6.3.2 BFS using $O(\lg n)$ bits

To reduce the space to $O(\lg n)$ bits, we dispense with the color array and explain how to determine visited and unvisited vertices. Assume that we can determine this in constant time. Our first observation is that Claim 6.11 is true even for unvisited vertices even though the first vertex in the adjacency list of unvisited vertices can be an arbitrary vertex (not necessarily referring to their parent in the BFS tree). However, we know (by the property of BFS) that no unvisited vertex is adjacent to a vertex in level less than d, and hence they can not reach s by a path at most d. Using the same argument, we can show that

Claim 6.12. *If vertices up to level d have been explored and visited, then a vertex x is a visited vertex if and only if by following through the parent pointers, x can reach s by a path of length at most d. Furthermore, a vertex is in level d if and only if we can reach s by a path of length exactly d by following through the parent pointers.*

Thus, to check whether a vertex is visited, we spend $O(d)$ time when exploring vertices at level d instead of $O(1)$ time when the *visited* array was stored explicitly. Hence, the total the time spent at level d is $O(nd + d\sum_{i \in V(d)} deg(i))$, where the first term gives the time to find all the visited vertices at level d, and the second term gives the time to explore those vertices (i.e., going through their neighbor lists, identifying the unvisited neighbors and setting their parent pointers). Summing over all levels, we get a total runtime of $O(n\ell^2 + m\ell)$. Note that this algorithm works only when the input undirected graph is connected as Claim 6.12 breaks down if there are more than one component. The modifications to handle the directed graphs are similar to those for the directed graph BFS algorithm. This proves the second part of Theorem 6.2.

6.4 Simulation of algorithms for rotate model in the implicit model

We can implement a single rotate operation that moves the head of the list by one position (which is assumed to be a unit-cost operation in the rotate model) in the implicit model by moving all the elements in the adjacency list circularly. If d_v is the degree of a vertex v, then performing a rotation of the adjacency list of v can be implemented in $O(d_v)$ time in the implicit model. Thus, if we have an algorithm in the rotate model that takes $t(m, n)$ time, then it can be implemented in $O(D \cdot t(m, n))$ time in the implicit model, where D is the maximum degree of the graph. One can get a better runtime by analysing the algorithm for the rotate model more carefully. In particular, if the runtime of the algorithm in the rotate model can be expressed as $r(m,n) + f(m,n)$, where $r(m,n)$ is the number of rotations performed and $f(m,n)$ is the remaining number of operations, then the algorithm can be implemented in the implicit model in $O(D \cdot r(m,n)) + f(m,n)$ time. Furthermore, if $r(m,n) \leq \sum_{v \in V} r_v(m,n)$ where $r_v(m,n)$ is the number of rotations made in v's list, then the runtime of the algorithm in the implicit model can be bounded by $\sum_{v \in V} r_v(m,n) \cdot d_v + f(m,n)$.

If the input graph is given in the adjacency array representation and if the ordering of the elements in the adjacency array is allowed to be changed, then one can simulate the rotate operation even faster. The main idea is to simulate the algorithm for the rotate model after sorting the adjacency arrays of each vertex. Using an in-place linear time radix sorting algorithm [81], sorting all the adjacency arrays can be done in $O(m)$ time. Next, we note that in the rotate model, the head points to an element in an arbitrary position (called the front element) in the adjacency list of any vertex and the unit operation moves it one position. Thus, it is enough to show how to access the next element in sorted order. We maintain the following invariant: if the first element is the i-th element in sorted order, then the adjacency array consists of the sorted array of all the adjacent vertices, with the first element swapped with the i-th element. To bring the $(i+1)$-st element to the front, we first perform a binary search for the i-th element which is in the first position in the 'almost sorted' adjacency array to find the position i, and move the elements appropriately to maintain the invariant (with the $(i+1)$-st element at the front). This takes $O(\lg d_v)$ time to simulate the rotation of the adjacency array of a vertex v with degree d_v. Thus, if we have an algorithm in the rotate model that takes $t(m,n)$ time, then it can be implemented in $O(\lg D \cdot t(m,n))$ time in the implicit model. Moreover, if the runtime of the algorithm in the rotate model can be expressed as $\sum_{v \in V} r_v(m,n) + f(m,n)$, where $r_v(m,n)$ is an upper bound on the number of rotations performed on vertex v and $f(m,n)$ is the remaining number of operations, then the algorithm can be implemented in the implicit model in $O(\sum_{v \in V} r_v(m,n) \lg d_v + f(m,n))$ time. Most of our algorithms in the next section use these simulations with some enhancements. The following result summarizes the overhead incurred while simulating any rotate model algorithm in the implicit model.

Theorem 6.13. *Let D be the maximum degree of a graph G. Then any algorithm running in $t(m,n)$ time in the* rotate *model can be simulated in the* implicit *model in (i) $O(D \cdot t(m,n))$ time when G is given in an adjacency list, and (ii) $O(\lg D \cdot t(m,n))$ time when G is given in an adjacency array. Furthermore, let $r_v(m,n)$ denote the number of rotations made in v's (whose degree is d_v) list, and $f(m,n)$ be the remaining number of operations. Then any algorithm running in $t(m,n) = \sum_{v \in V} r_v(m,n) + f(m,n)$ time in the* rotate *model can be simulated in the the* implicit *model in (i) $O(\sum_{v \in V} r_v(m,n) \cdot d_v + f(m,n))$ time when G is given in an adjacency list, and (ii) $O(\sum_{v \in V} r_v(m,n) \lg d_v + f(m,n))$ time when G is given in an adjacency array.*

Most of our algorithms in the implicit model use these simulations often with some enhancements and tricks to obtain better running time bounds for some specific problems.

6.5 DFS Algorithms in the implicit model

In this section, we describe our space-efficient algorithms for DFS in the implicit model proving Theorem 6.3. We restate the theorem statement here again for reader's convenience.

Theorem 6.3 (Restated) Let G be a directed or an undirected graph with a source vertex s and $\ell \leq n$ be the maximum depth of the DFS tree starting at s that can reach all other vertices. Then in the implicit model, using $O(\lg n)$ bits the vertices of G can be output in

(a) the lex-DFS order in $O(m^3/n^2 + \ell m^2/n)$ time if G is given in adjacency list and in $O(m^2 \lg n/n)$ time if G is given in adjacency array for undirected graphs. For directed graphs our algorithm takes $O(m^2(n+\ell^2)/n)$ time if G is given in adjacency list and $O(m \lg n(n+\ell^2))$ time if G is given in adjacency array;

(b) a general-DFS traversal order in $O(m^2/n)$ time if the input graph G is given in an adjacency list and in $O(m^2(\lg n)/n + m\ell \lg n))$ time if it is given in an adjacency array.

6.5.1 Lex-DFS using $O(\lg n)$ bits

To obtain a lex-DFS algorithm, we implement the $O(\lg n)$-bit DFS algorithm in the rotate model, described in Section 6.2.5, with a simple modification. First, note that in this algorithm (in the rotate model), we bring the parent of a vertex to the front of its adjacency list (by performing rotations) when we visit a vertex for the first time. Subsequently, we explore the remaining neighbors of the vertex in the left-to-right order. Thus, for each vertex, if its parent in the DFS were at the beginning of its adjacency list, then this algorithm would result in a lex-DFS algorithm. Now, to implement this algorithm in the implicit model, whenever we need to bring the parent to the front, we simply bring it to the front without changing the order of the other neighbors. Subsequently, we simulate each rotation by moving all the elements in the adjacency list circularly. As mentioned in Section 6.4, this results in an algorithm whose running time is $O(\sum_{v \in V} d_v(d_v+\ell) \cdot d_v) = O(m^3/n^2 + \ell m^2/n)$ if the graph is given in an adjancecy list and in $O(\sum_{v \in V} d_v(d_v+\ell) \cdot \lg d_v) = O(m^2(\lg n)/n + m\ell \lg n))$ when the graph is given in the form an adjacency array. This proves the first part of Theorem 6.3 for undirected graphs. The results for the directed case follow from simulating the corresponding results for the directed graphs.

6.5.2 General-DFS using $O(\lg n)$ bits

We implement the linear-time DFS algorithm of Theorem 6.1 for the rotate model that uses $n+O(\lg n)$ bits. This results in an algorithm that runs in $O(\sum_{v \in V} d_v^2+n) = O(m^2/n)$ time (or in $O(\sum_{v \in V} d_v \lg d_v + n) = O(m \lg m + n)$ time, when the graph is given as an adjacency array representation), using $n+O(\lg n)$ bits. We reduce the space usage of the algorithm to $O(\lg n)$ bits by encoding the visited/unvisited bit for each vertex with degree at least 2 within its adjacency list (and not maintaining this bit for degree-1 vertices). We describe the details below.

Whenever a node is visited for the first time in the algorithm for the rotated list model, we bring its parent to the front of its adjacency list. In the remaining part of the algorithm, we process each of its other adjacent vertices while rotating the adjacency list, till the parent comes to the front again. Thus, for each vertex v with degree d_v, we need to rotate v's adjacency list $O(d_v)$ times. In the implicit model, we also bring the parent to the front when a vertex is visited for the first time, for any vertex with degree at least 3. We use the second and third elements in the adjacency list to encode the visited/unvisited bit. But instead of rotating the adjacency list circularly, we simply scan through the adjacency list from left to right everytime we need to find the next unvisited vertex in its adjacency list. This requires $O(d_v)$ time for a vertex v with degree d_v. We show how to handle vertices with degree at most 2 separately.

As before, we can deal with the degree-1 vertices without encoding visited/unvisited bit as we encounter those vertices only once during the algorithm. For degree-2 vertices, we initially (at preprocessing stage) encode the bit 0 using the two elements in their adjacency arrays - to indicate that they are unvisited. When a degree-2 vertex is visited for the first time from a neighbor x, we move to its other neighbor – continuing the process as long as we encounter degree-2 vertices until we reach a vertex y with degree at least 3. If y is already visited, then we output the path consisting of all the degree-2 vertices and backtrack to x. If y is not visited yet, then we output the path upto y, and continue the search from y, and after marking y as visited. In both the cases, we also mark all the degree-2 nodes as visited (by swapping the two elements in each of their adjacency arrays).

During the preprocessing, for each vertex with degree at least 3, we ensure that the second and third elements in its adjacency list encode the bit 0 (to mark it unvisited). We maintain the invariant that for any vertex with degree at least 3, as long as it is not visited, the second and third elements in its adjacency array encode the bit 0; and after the vertex is visited, its parent (in the DFS tree) is at the front of its adjacency array, and the second and third elements in its adjacency array encode the bit 1. Thus, when we visit a node v with degree at least 3 for the first time, we bring its parent to the front, and then swap the second and third elements in the adjacency list, if needed, to mark it as visited. The total running time of this algorithm is bounded by $\sum_{v \in V} d_v^2 = O(m^2/n)$.

We can implement the above DFS algorithm even faster when the input graph is given in an adjacency array representation. We deal with vertices with degree at most 2 exactly as before. For a vertex v with degree at least 3, we bring its parent to the front and swap the second and third elements to mark the node as visited (as before) whenever v is visited for the first time. We then sort the remaining elements, if any, in the adjacency array, in-place (using the linear-time in-place radix sort algorithm [81]), and implement the rotations on the remaining part of the array as described in Section 6.4. The total running time of this algorithm is bounded by $\sum_{v \in V} d_v \lg d_v = O(m \lg m + n)$. This completes the proof of the second part of of Theorem 6.3.

6.6 BFS algorithms in the implicit model

In this section, we describe our space-efficient algorithms for BFS in the implicit model proving Theorem 6.4. We restate the theorem statement here again for reader's convenience.

Theorem 6.4 (Restated) Let G be a directed or an undirected graph with a source vertex that can reach all other vertices by a distance of at most ℓ. Then in the implicit model, using $O(\lg n)$ bits the vertices of G can be output in a BFS order in

(a) $O(m + n\ell^2)$ time;

(b) the runtime can be improved to $O(m + n\ell)$ time if there are no degree 2 vertices;

(c) the runtime can be improved to $O(m)$ if the degree of every vertex is at least $2 \lg n + 3$.

Before giving the proof, we outline the main ideas involved in proving Theorem 6.4. One can simulate the BFS algorithm of Item 2 of Theorem 6.2 (for the rotate model) in the implicit model using the simulation described in Section 6.4. Since these BFS algorithms scan through each of the adjacency lists/arrays at most twice during the algorithm, there won't be any slowdown in the runtime. This results in an algorithm running in $O(m\ell + n\ell^2)$ time, using $O(\lg n)$ bits. To improve the running time, we simulate the algorithm in Item 1 of Theorem 6.2 using the trick of encoding the *visited* bit of each vertex in its adjacency list instead of storing the *visited* array explicitly. This requires special attention to degree-1 and degree-2 vertices along with few other technical issues which are dealt in the proof given next.

Proof. Here we give the full proof of Theorem 6.4. In particular, we provide all the details of the case when the degree of each vertex is at least 3, and in that case, we show that we can implement the BFS algorithm using 4 colors of Theorem 3.5, by encoding the 4 color of a vertex using the first three elements in its adjacency list, resulting in an algorithm that takes $O(m+n\ell)$ time. Moreover, when the degree of every vertex is at least $2 \lg n + 3$, then we show that the above algorithm can be implemented more efficiently, resulting in an algorithm that takes $O(m)$ time. Details follow. One can simulate the BFS algorithm of in Item 2 of Theorem 6.2 (for the rotate model) in the implicit model using the simulation described in Section 6.4. Since these BFS algorithms scan through each of the adjacency lists/arrays at most twice during the algorithm, there won't be any slowdown in the runtime. This results in a BFS algorithm that runs in $O(m\ell + n\ell^2)$ time, using $O(\lg n)$ bits.

To improve the running time further, we simulate the algorithm in Item 1 of Theorem 6.2. But instead of storing the *visited* array explicitly, we encode the *visited* bit of each vertex in its adjacency list, as explained below, resulting in an algorithm that takes $O(m + n\ell^2)$ time, using $O(\lg n)$ bits. For a vertex x with degree at least 3, we encode its *visited* bit using the second and third elements in its adjacency list. To set the parent pointer for x (when it is visited for the first time), we bring its parent to the front, and move the second and third elements, if necessary, to encode the *visited* bit. We now describe how to deal with the degree-1 and degree-2 vertices.

First, observe that in the original BFS algorithm, we can simply output any degree-1 vertex, when it is visited for the first time. Thus, we need not store the *visited* bit for degree-1 vertices. For degree-2 vertices, we encode the *visited* bit using the two neighbors. We do not bring its parent to the front, as we do for other vertices. Whenever we need to check whether a visited degree-2 vertex is at depth d, we follow parent pointers from both the neighbors - if one of them does not exist, then the other one is its parent - for a distance of length at most d. (While following the parent pointers from a vertex to the root, it is easy to find its parent - since there is only one alternative to follow.) Thus, the first part of the theorem follows from Item 1 of Theorem 6.2, with the above modification.

To improve the runtime further, we implement the BFS algorithm using 4 colors of Theorem 3.5, where all the unvisited vertices are white, all the visited, yet unfinished vertices of the two consecutive layers of BFS are gray1 and gray2 respectively, and all the vertices which are completely explored are black. Suppose the degree of every vertex in the given graph is at least three. In this case, we can encode the color of any vertex by permuting the first three elements in its adjacency array appropriately. This enables us to retrieve (or modify) the color of any vertex in constant time by reading (or modifying) the permuted order of the first elements in its adjacency array.

Since we haven't stored the gray1 or gray2 vertices in a queue (as in the standard BFS), we scan through the entire vertex set (in the increasing order of their labels), and when we find any vertex v colored gray1, we color all its white neighbors with gray2, and color v itself with black. We call this an *exploration phase*. The time taken for each exploration phase is equal to the sum of the degrees of all the gray1 vertices at the beginning of the phase, plus $O(n)$. At the end of each exploration phase, we change the colors of all gray2 vertices to gray1 and also output them, using an additional $O(n)$ time. We call this a *consolidation phase*. We need to repeat the exploration and consolidation phases for ℓ times before all the nodes are colored black, where ℓ is the height of the BFS tree. Thus the overall time taken by this procedure can be bounded by $O(m + n\ell)$. This proves the second part of the theorem.

If every vertex has degree at least $2 \lg n + 3$, then we can perform BFS in $O(m)$ time (note that $m \geq n \lg n$ in this case) – by encoding the current set of gray1 vertices as a linked list by using $2 \lg n$ vertices to encode (a pointer to) the next vertex in the list. The time to read all the gray1 vertices in a phase when there are k vertices colored gray1 becomes $O(k \lg n)$ instead of $O(n)$. This results in $O(m + n \lg n)$ time which is $O(m)$. This proves the third part of the theorem. □

6.7 Minimum Spanning Tree

In this section, we start by giving an in-place implementation of the Prim's algorithm [51] to find a minimum spanning tree of a given weighted undirected graph in the rotate model. Here we are given a weight function $w : E \to Z$. We also assume that the weights of non-edges are ∞ and that the weights can be represented using $O(\lg n)$ bits. The input representation also changes slightly to accommodate these weights. Now each element of the circular linked list has three fields, (a) the vertex label, (b) the weight, and (c) the pointer to the next vertex respectively. In what follows, when we talk about computing a minimum spanning tree, what we mean is reporting the edges of such a tree. Our result is the following,

Theorem 6.14. *A minimum spanning forest of a given undirected weighted graph G can be found using $O(\lg n)$ bits and in $O(mn)$ time in the* rotate *model.*

Proof. Our rotate model algorithm basically mimics Prim's algorithm with a few tweaks. Prim's algorithm starts with initializing a set S with a vertex s. For every vertex v not in S, it finds and maintains $d[v] = \min\{w(v,x) : x \in S\}$ and $\pi[v] = x$ where $w(v,x)$ is the minimum among $\{w(v,y) : y \in S\}$. Then it repeatedly deletes the vertex with the smallest d value from $V - S$ adding it to S. Then the d values are updated by looking at the neighbors of the newly added vertex. To implement this procedure using just extra $O(\lg n)$ bits of space, first we need to find a way to mark/unmark a vertex v if it has been taken into S without using n bits explicitly. The way we do this is as follows. In the preprocessing step, the algorithm spends $O(m)$ time to bring the minimum valued neighbor (denote it by *min*) in the neighbor list of every vertex v by rotation. Subsequently we would attach the following meaning with the position of *min* in the list of any vertex v. If the first element in the list of any vertex v is *min*, this means that v is not taken into S so far during the execution of the algorithm, otherwise it belongs to S. This way we can store the information regarding the status of any vertex v without using any extra space but actually figuring out this information takes time proportional to the degree of v. Note that for vertices having degree one, we cannot determine exactly its status correctly by this method. But a simple fact we can use here. If a vertex z has degree one (say its neighbor is y), then the edge (y,z) is always a part of the minimum spanning tree. Hence, after the preprocessing step, we can output all such edges at once and embark on executing the rest of the algorithm.

The algorithm initializes the set S with the starting vertex s, and goes to its list to give a rotation so that *min* does not stay in the first location in s's list. We call this step as the *marking* step. This is followed by finding the smallest weight neighbor (say u) of s. According to Prim's algorithm, u should now move to S. We achieve the same by marking u i.e., going to u's list to give a rotation to move *min* from the first location to indicate that u belongs to S now and subsequently continue to in u's list find its smallest weight neighbor and repeat. Thus at any generic step of the algorithm, we go over the list of unmarked vertices (i.e., those vertices having *min* at the first position of their respective lists) and collect the minimum weight vertex (say t), and t is then marked and we continue until all the vertices are marked.

Clearly this method returns all the correct edges of a minimum spanning tree of G. Space bound of this algorithm is easily seen to be $O(\lg n)$ bits for keeping a few variables. Total time spent by algorithm can be bounded by $O(mn)$ where at the preprocessing step, it spends $O(m)$ time and after the preprocessing, for reporting each edges of the minimum spanning tree, in the worst case, the algorithm spends $O(m)$ time, hence $O(mn)$ is the bound on the running time. $\square$

As mentioned in Section 6.4, simulating this algorithm in the implicit model would result in an algorithm having running time $O(mn.d_v)$=$O(mn^2)$ if the graph is given in an adjacency list and in $O(mn.\lg d_v)$=$O(mn \lg n)$ when the graph is represented in an adjacency array. Hence, we have the following,

Theorem 6.15. *In the* implicit *model a minimum spanning forest of a given undirected weighted graph G can be found using $O(\lg n)$ bits and*

1. *$O(mn^2)$ time if the graph is given in an adjacency list, and*

2. *$O(mn \lg n)$ time when the graph is represented in an adjacency array.*

6.8 Improved algorithm for DFS and applications in read-only memory

This section discusses some of the applications of our previously designed in-place DFS/BFS algorithms. More specifically, we show, using a little more space, how to simulate any `rotate` model algorithm in the read-only model. This results in improved space-efficient algorithms for various fundamental graph problems in the read-only model over the classical implementation for these problems. This also matches the space and time bounds of our earlier designed algorithms for these problems. Hence this gives alternate proofs for those previous algorithms as well. Details follow.

Observe that, the only modification of the input that we do in our `rotate` model algorithm is to make the head pointer point to an arbitrary element in the adjacency list (instead of a fixed element) at various times. To simulate this in read-only memory, we can simply maintain a pointer in each of the lists in the adjacency list. The resources required to store and update such pointers is proven in the following lemma which we have proven in Chapter 3. We restate the lemma statement here for reader's convenience.

Lemma 3.17 (Restated) *Given the adjacency list representation of a directed or an undirected graph G on n vertices with m edges, using $O(m)$ time, one can construct an auxiliary structure of size $O(n \lg(m/n))$ bits that can store a "pointer" into an arbitrary index within the adjacency list of each vertex. Also, updating any of these pointers (within the adjacency list) takes $O(1)$ time.*

To actually get to the element in the list, we need the graph to be represented as what is referred as adjacency array [69]. Here given an index in the list of a vertex, we can access the (adjacent) vertex in that position of the vertex's adjacency list in constant time. Now if we simulate our `rotate` model algorithm of Theorem 6.1 in read-only memory using the auxiliary structure as stated in Lemma 3.17 as additional storage, then we obtain,

Theorem 6.16. *A DFS traversal of an undirected or a directed graph G, represented by* *an adjacency array, on n vertices and m edges can be performed in $O(m+n)$ time using $O(n \lg(m/n))$ bits, in the read-only model.*

The above result improves the DFS tradeoff result of Elmasry et al. [69] for relatively sparse graphs in the read-only model. In particular, they showed the following (which we restate here),

Theorem 4.1 (Restated) *For every function $t : \mathbb{N} \to \mathbb{N}$ such that $t(n)$ can be computed within the resource bound of this theorem (e.g., in $O(n)$ time using $O(n)$ bits), the vertices of a directed or undirected graph G, represented by adjacency arrays, with n vertices and m edges can be visited in depth first order in $O((m+n)t(n))$ time with $O(n + n\frac{\lg \lg n}{t(n)})$ bits.*

Thus to achieve $O(m+n)$ time for DFS, their algorithm (Theorem 4.1) uses $O(n \lg \lg n)$ bits. This is $\Omega(n \lg(m/n))$ for all values of m where $m = O(n \lg n)$. Hence, for sparse graphs we obtain better tradeoff bounds. Note that in Chapter 3, we obtained exactly same result by a slightly different technique. In what follows, we show that using Theorem 6.16, we can improve the space bounds of some of the classical applications of DFS in read-only model.

We illustrate this by the means of giving a few examples. Note that, one of the many classical applications of DFS (see [51]) include (i) topological sorting of the vertices of a directed acyclic graph [100], and (ii) producing a sparse (having $O(n)$ edges) spanning biconnected subgraph of a undirected biconnected graph G [68] etc. For all of these problems, classical algorithms [100, 68] take linear time and $O(n \lg n)$ bits of space. Now the following result is implicit from Theorem 4.3 and Theorem 4.10 which were proved in Chapter 4.

Theorem 6.17. *In the read-only model, if the DFS of G on n vertices and m edges, can be performed in $t(m,n)$ time using $s(m,n)$, where $s(m,n) = \Omega(n)$, bits of space, then using $O(s(m,n))$ bits and in $O(t(n,m))$ time, we can output*

1. *the vertices of a directed acyclic graph in topologically sorted order,*

2. *the edges of a sparse spanning biconnected subgraph of a undirected biconnected graph G.*

Now plugging the improved DFS algorithm of Theorem 6.16 in the above theorem, we obtain for these applications an improved (over the classical implementation) space-efficient implementations taking $O(m+n)$ time and $O(n \lg(m/n))$ bits of space. There are many other examples for which we can prove similar results but we choose these two only for demonstration purpose.

6.9 Solving NP-hard problems in in-place models

We show how to solve some NP-hard graph problems using logspace and exponential time in the rotate and implicit models. In particular, this implies that problems such as vertex cover and dominating set can be solved in exponential time using $O(\lg n)$ bits in both the models. In constrast, note that, no NP-hard problem can be solved in the ROM model using $O(\lg n)$ bits unless P=NP.

Similar to Fomin et al. [78], we define a class of graph problems in NP which we call *graph subset problems* where the goal is to find a subset of vertices satisfying some property. We show a meta theorem showing that a restricted class of graph subset problems that are in NP admit log-space exponential algorithms in the rotate and implicit models.

Given a graph G with its adjacency list, we encode a subset of the vertices as follows. For every vertex in the subset, we bring in the minimum labelled vertex among its neighbors to the front of the list, and for others, we keep a vertex with a higher label (than the minimum) at the front of the list. So it takes a linear time to check whether a vertex is in the subset. The algorithm enumerates all subsets (by this encoding) and simply verifies using the NP algorithm whether that subset satisfies the required property until it finds a subset satisfying the property or it has enumerated all the subsets. By a standard theorem in complexity theory [7], every problem in NP is actually verifiable by a log-space ROM and hence the overall space taken by our algorithm is only logarithmic. Note that our algorithm requires that the adjacency list of any vertex has at least two values, i.e. that the degree of any vertex is at least two. Thus we have

Theorem 6.18. *Any graph subset problem in* NP *can be solved using* $O(\lg n)$ *bits of extra space (and exponential time) in the* rotate *and* implicit *models in graphs* G *having minimum degree* 2*.*

Remark 1 We remark that the above idea can work for other graph problems that are not necessarily subset problems. For example, for testing hamiltonicity, we can simply explore all neighbors of a vertex (starting at the smallest labelled neighbor so we know when we have explored them all) in a systematic fashion simply encoding them into the adjacency list by moving the current neighbor to the front of the list, and test whether together they form a cycle of length n.

If the graphs have larger minimum degree (say at least $2\lg n$), we can even encode pointers in each adjacency list and using that we can even test for graph isomorphism in logarithmic space.

Remark 2 The minimum degree 2 restriction in the above theorem is not a serious restriction as for many problems (like vertex cover, dominating set and travelling salesperson problem), we can handle the degree 1 vertices easily.

6.10 Concluding remarks and open problems

Our initial motivation was to get around the limitations of ROM to obtain a reasonable model for graphs in which we can obtain space efficient algorithms. We achieved that by introducing two new frameworks and obtained efficient (of the order of $O(n^3 \lg n)$) algorithms using $O(\lg n)$ bits of space for fundamental graph search procedures. We also discussed various applications of our DFS/BFS results, and it is not surprising that many simple corollaries would follow as DFS/BFS being the backbone of so many graph algorithms. We showed that some of these results also translate to improved space efficient algorithms in ROM (by simulating the rotate model algorithms in ROM with one pointer per list). With some effort, we can obtain log space algorithm for minimum spanning tree. These results can be contrasted with the state of the art results in ROM that take almost linear bits for some of these problems other than having large runtime bounds. All our algorithms are conceptually simple, and as they don't use any heavy data structures, we believe that they are also practical to implement. Still, there are plenty of algorithmic graph problems to be studied in this model. We believe that our work is the first step towards this and will inspire further investigation into designing in-place algorithms for other graph problems.

Surprisingly we could design log-space algorithm for some P-complete problems, and so it is important to understand the power of our model. Towards that we discovered that we can even obtain log-space algorithms for some NP-hard graph problems. More specifically, we defined *graph subset problems* and obtained log-space exponential time algorithms for problems belonging to this class in Section 6.9. One interesting future direction would be to determine the exact computational power of these models along with exploring the horizon of interesting complexity theoretic consequences of problems in this model.

Chapter 7

Conclusion

The concept of space complexity had already been explored in the 1970s; In particular, Savitch's theorem [7] was one of the initial results regarding the space complexity of graph algorithms. After that many beautiful results regarding the space bounds of different kinds of problems have been discovered. Mostly these algorithms are extremely space efficient at the cost of large, yet polynomial, running time. In this thesis we took a slightly different approach while designing space efficient algorithms for some basic and fundamental graph algorithms. Our main focus was to obtain/design space-efficient yet reasonably time-efficient graph algorithms on various restricted memory computational model. In this concluding chapter we take a look back on the results obtained, ponder on possible directions of future work and mention some open problems.

We started this thesis focusing on designing linear time algorithms for many fundamental graph problems in ROM. The following table summarizes the main results obtained along with mentioning a few problems which we leave open.

Problem	Time	Space (in bits)	Status
BFS	$O(m+n)$	$2n+o(n)$	Thm 3.5
BFS	$O(m\lg n)$ or $O(mn)$	$n+o(n)$	Open
DFS, 2-vertex (edge) connectivity	$O(m+n)$	$O(n\lg(m/n))$	Thm 3.19
Chain decomposition	$O(m+n)$	$O(n\lg(m/n))$	Thm 3.19
DFS, 2-vertex (edge) connectivity	$O(m+n)$	$O(n\lg\lg n)$	Thm 3.26
Chain decomposition	$O(m+n)$	$O(n)$ or $O(n\lg\lg n)$	Open
DFS, 2-vertex (edge) connectivity	$O(m+n)$	$O(n)$	Open
DFS	Polytime	$o(n)$	Open

In the next chapter, we focused on designing linear bits algorithms for some of well studied classical graph problems. This chapter also saw proper fusion of results from succinct data structure and algorithmic graph theory to design space efficient graph algorithms. We developed a technique called space-efficient tree cover method and use it successively to design very lightweight (in terms of space) algorithms for plethora of DFS applications in ROM. The following table summarizes the main results obtained along with mentioning a few problems which we leave open.

Problem	Time	Space (in bits)	Status
DFS, Topological sort	$O(m\lg\lg n)$	$O(n)$	Thm 4.1 & 4.3
DFS, Topological sort	$O(m)$ or $O(m\lg^* n)$	$O(n)$	Open
2-vertex (edge) connectivity	$O(m\lg\lg n\lg n)$	$O(n)$	Thm 4.18 & 4.19
2-vertex (edge) connectivity	$O(m)$ or $O(m\lg\lg n)$	$O(n)$	Open
st-numbering	$O(m\lg^2 n\lg\lg n)$	$O(n)$	Thm 4.16 & 4.20
st-numbering	$O(m)$ or $O(m\lg n\lg\lg n)$	$O(n)$	Open
MCS, Max-Independent set	$O(m^2/n+m\lg n)$	$O(n)$	Thm 4.24 & 4.26
Coloring chordal graphs	$O((m^2\lg n)/n+m\lg^2 n)$	$O(n\lg(m/n))$	Thm 4.27
Coloring chordal graphs	Polytime	$O(n)$	Open
MCS	$O(m)$/Polytime	$O(n)/o(n)$	Open

In the next chapter we shifted our focus to designing space efficient algorithms for a special class of optimization problems on bounded treewidth graphs. Toward that we use the result of Barba et al. [18] who introduced the *compressed stack technique*, a procedure to transform algorithms whose main memory consumption takes the form of a stack into memory-constrained algorithms, and showed various applications of this method by designing space efficient algorithms for problems in computational geometry. We extended the technique of Barba et al. slightly and showed an enhanced compression technique which can be applied to a broader class of stack algorithms. Using this, we proved the following meta theorem which roughly says, for bounded treewidth graphs, if any graph problem can be described in monadic second order (MSO) logic, we can obtain a smooth deterministic time-space trade-off from constant words to linear space. Our result can be seen as a generalization of the results of Elberfeld et al. [64] and Courcelle [53]. Our understanding of such general compression technique is limited only to a few very basic data structures, thus it would be an interesting task to design such general compression schemes for variety of other fundamental data structures as a possible future work. Also our algorithm is optimal upto a log factor in the space bound and particularly time efficient. It remains an open problem to make it optimal from space point of view i.e., shave off the multiplicative log factor from the space bound with very little or no compromise in the running time.

In contrast to sorting, various geometric and string problems (where plenty of algorithms with sublinear space bounds are known), most of these aforementioned fundamental polynomial time graph algorithms on a graph with n vertices seem to need almost $\Theta(n)$ bits of space in word RAM. Also under some reasonable complexity theoretic assumption, this is the best we can hope for. Motivated by this barrier and inspired by the classical in-place sorting algorithms, we introduced two frameworks for designing efficient *in-place graph algorithms* (i.e. with $O(\lg n)$ bits of extra space), and showed that we can beat, by exponential margin, the ROM space bounds for several fundamental graph algorithms like DFS, BFS, shortest path and many more in this new model. Our goal here is to allow restricted modifications to the input but still keeping the graph adjacency structure intact throughout the execution of the algorithm. To achieve this, in one framework (we refer to this as *implicit* model) we simply allow elements in the adjacency list of a vertex to be permuted, and in the other framework (we refer to this as *rotate* model), we assume that the adjacency lists of the vertices are circular lists and we can only modify the heads of the lists. We summarize our main results in both of these models and list a few open problems in the table below.

Model	Problem	Time	Space (in bits)	Status
Rotate	DFS	$O(mn^2)$	$O(\lg n)$	Thm 6.1
Rotate	BFS	$O(n^3)$	$O(\lg n)$	Thm 6.2
Rotate	MST	$O(mn)$	$O(\lg n)$	Thm 6.5
Rotate	BFS/DFS/MST	$O(m)$ or $o(n^3)$	$O(\lg n)$	Open
Implicit	DFS	$O(m^2/n)$	$O(\lg n)$	Thm 6.3
Implicit	BFS	$O(n^3)$	$O(\lg n)$	Thm 6.4
Implicit	MST	$O(mn^2)$	$O(\lg n)$	Thm 6.5
Implicit	BFS/DFS/MST	$O(m)$ or $o(n^3)$	$O(\lg n)$	Open
Implicit/Rotate	Other graph problems	Low-degree polytime	$O(\lg n)$	Open

One broad and general future research direction in this regard would be to design efficient in-place algorithms for various other graph problems and extend them to work even in recently introduced *restore model* by Chan et al. [46].

Bibliography

[1] A. Aggarwal and R. J. Anderson. A random NC algorithm for depth first search. *Combinatorica*, 8(1):1–12, 1988.

[2] A. Aggarwal, R. J. Anderson, and M. Kao. Parallel depth-first search in general directed graphs. *SIAM J. Comput.*, 19(2):397–409, 1990.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[5] R. J. Anderson and E. W. Mayr. Parallelism and the maximal path problem. *Inf. Process. Lett.*, 24(2):121–126, 1987.

[6] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

[7] S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[8] T. Asano, K. Buchin, M. Buchin, M.Korman, W. Mulzer, G. Rote, and A. Schulz. Reprint of: Memory-constrained algorithms for simple polygons. *Comput. Geom.*, 47(3):469–479, 2014.

[9] T. Asano, T. Izumi, M. Kiyomi, M. Konagaya, H. Ono, Y. Otachi, P. Schweitzer, J. Tarui, and R. Uehara. Depth-first search using O(n) bits. In *25th ISAAC*, pages 553–564, 2014.

[10] T. Asano, D. G. Kirkpatrick, K. Nakagawa, and O. Watanabe. $\tilde{\mathrm{O}}(\sqrt{\mathrm{n}})$-space and polynomial-time algorithm for planar directed graph reachability. In *39th MFCS LNCS 8634*, pages 45–56, 2014.

[11] T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-work-space algorithms for geometric problems. *JoCG*, 2(1):46–68, 2011.

[12] T. Asano, W. Mulzer, and Y. Wang. Constant-work-space algorithms for shortest paths in trees and simple polygons. *J. Graph Algorithms Appl.*, 15(5):569–586, 2011.

[13] B. Aspvall, J. A. Telle, and A. Proskurowski. Memory requirements for table computations in partial k-tree algorithms. *Algorithmica*, 27(3):382–394, 2000.

[14] N. Banerjee, S. Chakraborty, and V. Raman. Improved space efficient algorithms for BFS, DFS and applications. In *Computing and Combinatorics - 22nd International Conference, COCOON 2016, Ho Chi Minh City, Vietnam, August 2-4, 2016, Proceedings*, pages 119–130, 2016.

[15] N. Banerjee, S. Chakraborty, V. Raman, S. Roy, and S. Saurabh. Time-space tradeoffs for dynamic programming in trees and bounded treewidth graphs. In *21st COCOON*, volume 9198, pages 349–360. springer, LNCS, 2015.

[16] N. Banerjee, S. Chakraborty, V. Raman, and S. R. Satti. Space efficient linear time algorithms for BFS, DFS and applications. *Theory of Computing Systems*, Jan 2018.

[17] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation*

Techniques, 6th International Workshop, RANDOM 2002, Cambridge, MA, USA, September 13-15, 2002, Proceedings, pages 1–10, 2002.

[18] L. Barba, M. Korman, S. Langerman, K. Sadakane, and R. I. Silveira. Space-time trade-offs for stack-based algorithms. *Algorithmica*, 72(4):1097–1129, 2015.

[19] G. Barnes, J. Buss, W. Ruzzo, and B. Schieber. A sublinear space, polynomial time algorithm for directed s-t connectivity. *SIAM J. Comput.*, 27(5):1273–1282, 1998.

[20] V. Batagelj and M. Zaversnik. An $O(m)$ algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[21] Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991.

[22] A. Berry, R. Krueger, and G. Simonet. Maximal label search algorithms to compute perfect and minimal elimination orderings. *SIAM J. Discrete Math.*, 23(1):428–446, 2009.

[23] B. K. Bhattacharya, M. De, S. C. Nandy, and S. Roy. Maximum independent set for interval graphs and trees in space efficient models. In *26th CCCG*, 2014.

[24] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[25] H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, 1998.

[26] H. L. Bodlaender and A. M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *Comput. J.*, 51(3):255–269, 2008.

[27] H. L. Bodlaender and J. A. Telle. Space-efficient construction variants of dynamic programming. *Nord. J. Comput.*, 11(4):374–385, 2004.

[28] R. B. Borie, R. G. Parker, and C. A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7(5&6):555–581, 1992.

[29] A. Borodin and S. A. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11(2):287–297, 1982.

[30] A. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch, and M. Tompa. A time-space tradeoff for sorting on non-oblivious machines. *J. Comput. Syst. Sci.*, 22(3):351–364, 1981.

[31] U. Brandes. Eager st-ordering. In *10th ESA*, pages 247–256, 2002.

[32] G. S. Brodal, S. Chaudhuri, and J. Radhakrishnan. The randomized complexity of maintaining the minimum. *Nord. J. Comput.*, 3(4):337–351, 1996.

[33] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *6th WADS, LNCS 1663*, pages 37–48, 1999.

[34] H. Brönnimann, T. M. Chan, and E. Y. Chen. Towards in-place geometric algorithms and data structures. In *Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004*, pages 239–246, 2004.

[35] H. Buhrman, R. Cleve, M. Koucký, B. Loff, and F. Speelman. Computing with a full memory: catalytic space. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 857–866, 2014.

[36] H. Buhrman, M.l Koucký, B. Loff, and F. Speelman. Catalytic space: Non-determinism and hierarchy. In *33rd STACS 2016, February 17-20, 2016, Orléans, France*, pages 24:1–24:13, 2016.

[37] D. Chakraborty, A. Pavan, R. Tewari, N. V. Vinodchandran, and L. Yang. New time-space upperbounds for directed reachability in high-genus and h-minor-free graphs. In *FSTTCS*, pages 585–595, 2014.

[38] S. Chakraborty, S. Jo, and S. R. Satti. Improved space-efficient linear time algorithms for some classical graph problems. In *15th CTW*, 2017.

[39] S. Chakraborty, A. Mukherjee, V. Raman, and S. R. Satti. A framework for designing in-place graph algorithms. Manuscript, 2017.

[40] S. Chakraborty, V. Raman, and S. R. Satti. Biconnectivity, chain decomposition and st-numbering using O(n) bits. In *27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia*, pages 22:1–22:13, 2016.

[41] S. Chakraborty, V. Raman, and S. R. Satti. Biconnectivity, st-numbering and other applications of dfs using O(n) bits. *Journal of Computer and System Sciences*, 90(Supplement C):63 – 79, 2017.

[42] S. Chakraborty and S. R. Satti. Space-efficient algorithms for maximum cardinality search, stack BFS, queue BFS and applications. In *23rd COCOON*, 2017.

[43] S. Chakraborty and S. R. Satti. Space-efficient algorithms for maximum cardinality search, its applications, and variants of bfs. *Journal of Combinatorial Optimization*, Mar 2018.

[44] T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. *Discrete & Computational Geometry*, 37(1):79–102, 2007.

[45] T. M. Chan, J. I. Munro, and V. Raman. Faster, space-efficient selection algorithms in read-only memory for integers. In *Algorithms and Computation - 24th International Symposium, ISAAC 2013, Hong Kong, China, December 16-18, 2013, Proceedings*, pages 405–412, 2013.

[46] T. M. Chan, J. I. Munro, and V. Raman. Selection and sorting in the "restore" model. In *25th-SODA*, pages 995–1004, 2014.

[47] J. Cheriyan, M. Kao, and R. Thurimella. Scan-first search and sparse certificates: An improved parallel algorithms for k-vertex connectivity. *SIAM J. Comput.*, 22(1):157–174, 1993.

[48] J. Cheriyan and S. N. Maheshwari. Finding nonseparating induced cycles and independent spanning trees in 3-connected graphs. *J. Algorithms*, 9(4):507–537, 1988.

[49] D. R. Clark. *Compact Pat Trees.* PhD thesis. University of Waterloo, Canada, 1996.

[50] S. A. Cook and C. Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM J. Comput.*, 9(3):636–652, 1980.

[51] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009.

[52] B. Courcelle. In *Handbook of graph grammars and computing by graph transformation, Vol. 1.* River Edge, NJ.

[53] B. Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[54] B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic.* Cambridge University Press, 2012.

[55] S. Curran, O. Lee, and X. Yu. Finding four independent trees. *SIAM J. Comput.*, 35(5):1023–1058, 2006.

[56] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms.* Springer, 2015.

[57] O. Darwish and A. Elmasry. Optimal time-space tradeoff for the 2d convex-hull problem. In *22th ESA*, pages 284–295, 2014.

[58] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms.* McGraw-Hill, 2008.

[59] S. Datta, N. Limaye, P. Nimbhorkar, T. Thierauf, and F. Wagner. Planar graph isomorphism is in log-space. In *24th CCC*, pages 203–214, 2009.

[60] Y. Dodis, M. Patrascu, and M. Thorup. Changing base without losing space. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC)*, pages 593–602, 2010.

[61] M.E. Dyer and A.M. Frieze. On the complexity of partitioning graphs into connected subgraphs. *Discrete Applied Mathematics*, 10(2):139–153, 1985.

[62] J. Ebert. st-ordering the vertices of biconnected graphs. *Computing*, 30(1):19–33, 1983.

[63] J. Edmonds, C. K. Poon, and D. Achlioptas. Tight lower bounds for st-connectivity on the NNJAG model. *SIAM J. Comput.*, 28(6):2257–2284, 1999.

[64] M. Elberfeld, A. Jakoby, and T. Tantau. Logspace versions of the theorems of bodlaender and courcelle. In *51th FOCS*, pages 143–152, 2010.

[65] M. Elberfeld and K. Kawarabayashi. Embedding and canonizing graphs of bounded genus in logspace. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 383–392, 2014.

[66] M. Elberfeld and P. Schweitzer. Canonizing graphs of bounded tree width in logspace. In *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, pages 32:1–32:14, 2016.

[67] F. Ellen. Constant time operations for words of length w. *Unpublished Manuscript.*

[68] A. Elmasry. Why depth-first search efficiently identifies two and three-connected graphs. In *21st ISAAC*, pages 375–386, 2010.

[69] A. Elmasry, T. Hagerup, and F. Kammer. Space-efficient basic graph algorithms. In *32nd STACS*, pages 288–301, 2015.

[70] A. Elmasry, D. D. Juhl, J. Katajainen, and S. R. Satti. Selection from read-only memory with limited workspace. *Theor. Comput. Sci.*, 554:64–73, 2014.

[71] D. Eppstein, M. Loffler, and D. Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM Journal of Experimental Algorithmics*, 18, 2013.

[72] S. Even and R. E. Tarjan. Computing an st-numbering. *Theo. Comp. Sci.*, 2(3):339–344, 1976.

[73] S. Even and R. E. Tarjan. Corrigendum: Computing an st-numbering. TCS 2(1976):339-344. *Theor. Comput. Sci.*, 4(1):123, 1977.

[74] A. Farzan and J. Ian Munro. Succinct representation of dynamic trees. *Theor. Comput. Sci.*, 412(24):2668–2678, 2011.

[75] A. Farzan, R. Raman, and S. R. Satti. Universal succinct representations of trees? In *36th ICALP*, pages 451–462, 2009.

[76] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.

[77] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[78] F. V. Fomin, S. Gaspers, D. Lokshtanov, and S. Saurabh. Exact algorithms via monotone local search. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 764–775, 2016.

[79] G. Franceschini and J. Ian Munro. Implicit dictionaries with $O(1)$ modifications per update and fast search. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 404–413, 2006.

[80] G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, pages 533–545, 2007.

[81] G. Franceschini, S. Muthukrishnan, and M. Patrascu. Radix sorting with no extra space. In *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, pages 194–205, 2007.

[82] G. N. Frederickson. Upper bounds for time-space trade-offs in sorting and selection. *J. Comput. Syst. Sci.*, 34(1):19–26, 1987.

[83] K. Fredriksson and P. Kilpeläinen. Practically efficient array initialization. *Softw., Pract. Exper.*, 46(4):435–467, 2016.

[84] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific J. Math*, 15:835–855, 1965.

[85] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000.

[86] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[87] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3):231–246, 2006.

[88] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Trans. Algorithms*, 2(4):510–534, 2006.

[89] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* 2004.

[90] M. Grohe and D. Marx. Structure theorem and isomorphism test for graphs with excluded topological subgraphs. *SIAM J. Comput.*, 44(1):114–159, 2015.

[91] S. Guha and A. McGregor. Graph synopses, sketches, and streams: A survey. *PVLDB*, 5(12):2030–2031, 2012.

[92] A. Gupta, W. Hon, R. Shah, and J. S. Vitter. A framework for dynamizing succinct data structures. In *34th ICALP*, pages 521–532, 2007.

[93] E. Győri. Partition conditions and vertex-connectivity of graphs. *Combinatorica*, 1(3):263–273, 1981.

[94] T. Hagerup and F. Kammer. Succinct choice dictionaries. *CoRR*, abs/1604.06058, 2016.

[95] M. He, J. I. Munro, and S. R. Satti. Succinct ordinal trees based on tree covering. *ACM Trans. Algorithms*, 8(4):42, 2012.

[96] W. K. Hon, K. Sadakane, and W. K. Sung. Succinct data structures for searchable partial sums with optimal worst-case performance. *Theor. Comput. Sci.*, 412(39):5176–5186, 2011.

[97] A. Huck. Independent trees in planar graphs. *Graphs and Combinatorics*, 15(1):29–77, 1999.

[98] A. Itai and M. Rodeh. The multi-tree approach to reliability in distributed networks. *Inf. Comput.*, 79(1):43–59, 1988.

[99] F. Kammer, D. Kratsch, and M. Laudahn. Space-efficient biconnected components and recognition of outerplanar graphs. In *41st MFCS*, 2016.

[100] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition.* Addison-Wesley, 1973.

[101] M. Koucký. Catalytic computation. *Bulletin of the EATCS*, 118, 2016.

[102] R. Krueger, G. Simonet, and A. Berry. A general label search to investigate classical graph search algorithms. *Discrete Applied Mathematics*, 159(2-3):128–142, 2011.

[103] T. W. Lai and D. Wood. Implicit selection. In *SWAT 88, 1st Scandinavian Workshop on Algorithm Theory, Halmstad, Sweden, July 5-8, 1988, Proceedings*, pages 14–23, 1988.

[104] L. Lovasz. A homology theory for spanning tress of a graph. *Acta Mathematica Hungarica*, 30(3-4):241–251, 1977.

[105] H. Lu and C. Yeh. Balanced parentheses strike back. *ACM Trans. Algorithms*, 4(3):28:1–28:13, July 2008.

[106] A. McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014.

[107] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984.

[108] C. W. Mortensen, R. Pagh, and M. Patrascu. On dynamic range reporting in one dimension. In *37th STOC*, pages 104–111, 2005.

[109] J. I. Munro. Tables. In *FSTTCS*, pages 37–42, 1996.

[110] J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980.

[111] J. I. Munro and V. Raman. Selection from read-only memory and sorting with minimum data movement. *Theor. Comput. Sci.*, 165(2):311–323, 1996.

[112] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[113] J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in O($\log^2$ n) time. *J. Comput. Syst. Sci.*, 33(1):66–74, 1986.

[114] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[115] H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992.

[116] J. Pagter and T. Rauhe. Optimal time-space trade-offs for sorting. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 264–268, 1998.

[117] A. Poyias, S. J. Puglisi, and R. Raman. Compact dynamic rewritable (cdrw) arrays. *To appear in 19th ALENEX 2017.*

[118] R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In *7th WADS, LNCS 2125*, pages 426–437, 2001.

[119] J. H. Reif. Symmetric complementation. *J. ACM*, 31(2):401–421, 1984.

[120] J. H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985.

[121] O. Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4), 2008.

[122] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5(2):266–283, 1976.

[123] J. M. Schmidt. *Structure and constructions of 3-connected graphs.* PhD thesis, Free University of Berlin, 2010.

[124] J. M. Schmidt. A simple test on 2-vertex- and 2-edge-connectivity. *Inf. Process. Lett.*, 113(7):241–244, 2013.

[125] J. M. Schmidt. The mondshein sequence. In *41st ICALP*, pages 967–978, 2014.

[126] T. Tantau. Logspace optimization problems and their approximability properties. *Theory Comput. Syst.*, 41(2):327–350, 2007.

[127] R. E. Tarjan. Maximum cardinality search and chordal graphs. *Unpublished Lecture Notes CS 259.*

[128] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[129] R. E. Tarjan. A note on finding the bridges of a graph. *Inf. Process. Lett.*, 2(6):160–161, 1974.

[130] R. E. Tarjan. Two streamlined depth-first search algorithms. *Fund. Inf.*, 9:85–94, 1986.

[131] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.

[132] M. Tompa. Two familiar transitive closure algorithms which admit no polynomial time, sublinear space implementations. *SIAM J. Comput.*, 11(1):130–137, 1982.

[133] P. v. E. Boas. Preserving order in a forest in less than logarithmic time. In *16th FOCS*, pages 75–84, 1975.

[134] K. Wada and K. Kawaguchi. Efficient algorithms for tripartitioning triconnected graphs and 3-edge-connected graphs. In *19th WG*, pages 132–143, 1993.

[135] A. Zehavi and A. Itai. Three tree-paths. *Journal of Graph Theory*, 13(2):175–188, 1989.