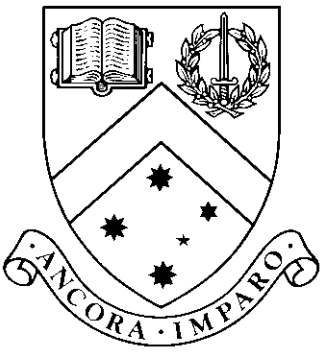
Graph parameters: theory, generation and dissemination

by

Srinibas Swain



Thesis

Submitted by Srinibas Swain

for fulfilment of the Requirements for the Degree of

Doctor of Philosophy (0190)

Supervisor: Prof. Graham Farr

Associate Supervisors: Dr. Kerri Morgan and Prof. Paul Bonnington

Faculty of Information Technology, Clayton
Monash University

July, 2019

. . . to my mother and father.

Contents

List of Tables . . . . . **vii**

List of Figures . . . . . **ix**

Abstract . . . . . **x**

Acknowledgments . . . . . **xii**

1 Introduction **1**
- 1.1 Motivation 2
- 1.2 Results 3
  - 1.2.1 OLGA updates 3
  - 1.2.2 Most Frequent Connected Induced Subgraph (MFCIS) 4
  - 1.2.3 OLGA analysis 4
- 1.3 Synopsis of the thesis 5

2 Definitions and notation **7**
- 2.1 Graph theory 7
- 2.2 Complexity theory 9

3 A survey of graph repositories **13**
- 3.1 Introduction 13
- 3.2 Graph repositories 14
  - 3.2.1 Print resources 14
  - 3.2.2 Electronic resources 21
  - 3.2.3 Interactive repositories 32
- 3.3 In a nutshell 40
- 3.4 Conclusion 40

4 Importance of OLGA **43**
- 4.1 OLGA as a conjecture (dis)prover 43
- 4.2 OLGA as a conjecture (theorem) generator 45
- 4.3 OLGA as an assistant for inductive proofs 45
- 4.4 Scope of OLGA in machine learning 46
- 4.5 Self-reducibility 46

5 Design and implementation **51**
5.1 Challenges in OLGA development 51
5.1.1 Graph generation 51
5.2 Isomorphism checking 53
5.3 Implementation priorities 55
5.3.1 I/O-bound jobs and CPU-bound algorithms in OLGA 59
5.4 Efficient storage 61
5.5 Query optimization 61
5.6 OLGA architecture 64

6 Graph parameters **65**
6.1 Metric parameters 65
6.1.1 Radius and diameter 65
6.1.2 Longest path 66
6.1.3 Girth and circumference 67
6.2 Algebraic parameters 68
6.2.1 Tutte polynomial 69
6.2.2 Eigenvalues 70
6.2.3 Automorphism group 71
6.3 Structural parameters 72
6.3.1 Graph degeneracy 72
6.3.2 Maximum matching 73
6.3.3 Independence number, vertex cover number, and clique number . . . 73
6.3.4 Edge-connectivity 75
6.3.5 Vertex-connectivity 76
6.3.6 Treewidth 76
6.3.7 Domination number 79
6.4 Graph colouring 81
6.4.1 Chromatic index 81
6.4.2 Chromatic number 83
6.4.3 Genus 84

7 Most Frequent Connected Induced Subgraph (MFCIS) **89**
7.1 Introduction 89
7.2 MFCIS of special classes of Graphs 90
7.3 Finding a MFCIS is #P-hard 92
7.4 MFCIS of perfect binary tree 99
7.4.1 Definition and notation 99
7.4.2 Observations and facts 100
7.4.3 Properties of the MFCIS(T_N) 100
7.4.4 A recursive construction for MFCIS of T_N 103
7.5 Conclusion 105

8 OLGA: a research tool . . . 107
8.1 Chromatic critical graphs . . . 107
8.1.1 Edge critically chromatic graphs . . . 107
8.1.2 Vertex-critically chromatic graphs . . . 109
8.2 MFCIS data from OLGA . . . 111
8.3 Conclusion . . . 113

9 Conclusion and future work . . . 117
9.1 Highlights of the thesis . . . 117
9.2 OLGA extension . . . 118
9.3 MFCIS characterisation . . . 119
9.4 Bound matching strategy . . . 119

Appendix A A guide to OLGA . . . 121
A.1 OLGA installation . . . 121
A.2 OLGA directory structure . . . 123
A.3 OLGA code structure . . . 123
A.4 A useful function . . . 127

List of Tables

1.1 Growth of number of graphs [66, 120]. 1

3.1 A summary of graphs in "An Atlas of Graphs". 20
3.2 A summary of parameters in "An Atlas of Graphs". 21
3.3 A summary of graphs from McKay's combinatorial data. 24
3.4 Royle's data on graph counting. 26
3.5 A summary of graphs from Royle's repository. 28
3.6 Meginger's repository. 30
3.7 A summary of Haggard's repository. 31
3.8 A summary of Bailey et al's repository. 31
3.9 A summary of other useful repositories. 32
3.10 A summary of graphs from Encyclopedia of Graphs. +: data from McKay's repository; *: data from Royle's repository; %: data from Conder's repository; #: data from house of graphs; &: data from Potočnik's repository. . . 36
3.11 A summary of graphs from House of Graphs. +: data from McKay's repository; *: data from Royle's repository; #: data from Spence's repository; $: data from Meginger's repository. 38
3.12 Parameters in House of Graphs. 38
3.13 Parameter filter used in Discrete ZOO. 39
3.14 Summary of graph repositories. 41

4.1 Parameters and OLGA-self-reducibility. 49

5.1 $P(G)$ for all unlabelled connected graphs of 4 vertices. 56
5.2 $P(G)$ for some unlabelled connected graphs of 5 vertices. 58
5.3 Space requirements to store graphs in graph6 format. 62
5.4 Query execution using the first approach. 62
5.5 Query execution using the second approach. 63
5.6 Query execution independent of execution sequence. 63

6.1 Treewidth bound unmatched cases. 79
6.2 Chromatic index bound unmatched cases. 82
6.3 Genus bound unmatched cases. 87

8.1 Number of edge-critical chromatic graphs with up to 10. 108

8.2 Number of edge-critical k-chromatic graphs, for $k \leq 10$ with order up to 10. 109
8.3 Number of vertex-critical chromatic graphs with order up to 10. 111
8.4 Number of vertex-critical k-chromatic graphs, for $k \leq 10$ with order up to 10.111
8.5 Multiplicity of MFCIS 112
8.6 Graphs with multiple MFCISs with the size of their automorphism group. . 114

9.1 Space requirements for graphs. 118

A.1 Parameter files. 126

List of Figures

3.1 An extract from the report [7]. 15
3.2 Foster's Census with Foster, from [51]. 16
3.3 Data flow between graph repositories. 41

4.1 Some graphs with unique eigenvalues. 44
4.2 Supervised learning on OLGA data. 46

5.1 Vertex deletion in OLGA. 56
5.2 Edge deletion in OLGA. 58

6.1 Sphere and Tori. 85

7.1 K_5 and its connected induced subgraphs. 90
7.2 $\mathscr{C}(T_2, T_1) = \{H_1, H_2, H_3, H_4\}$ and $J \notin \mathscr{C}(T_2, T_1)$. 99
7.3 MFCISs of T_3. 104
7.4 A MFCIS of T_5 constructed from MFCIS of T_3 using the construction used in Lemma 7.3. 105

8.1 Some edge-critically 4-chromatic graphs with order up to 8 (graphs drawn with reference to [142]). 108
8.2 Edge-critically 4-chromatic graphs with trivial automorphism group. 109
8.3 All vertex-critical 4-chromatic graphs with order up to 8 that are not edge-critical 4-chromatic 110
8.4 Graph of order 9 with graph id 3669386708, with its 8 MFCISs. 113
8.5 Graph of order 9 with graph id 1355399303, with its 8 MFCISs. 113
8.6 Graph of order 10 with graph id 6450870734291, with its 11 MFCISs. . . . 114
8.7 Graphs with graph ids from Table 8.6. 115

Graph parameters: theory, generation and dissemination

Srinibas Swain
srinibas.swain@monash.edu
Monash University, 2019

Supervisor: Prof. Graham Farr
graham.farr@monash.edu
Associate Supervisors: Dr. Kerri Morgan and Prof. Paul Bonnington
kerri.morgan@deakin.edu.au and paul.bonnington@monash.edu

Abstract

We create a large, complex, novel system, an Online Graph Atlas (OLGA), which is a digital repository of graphs and parameter values and may be queried in a variety of ways, considerably extending previous work of Barnes, Bonnington, Farr and Sio. OLGA stores over 20 important parameters for each graph up to some order. The current version of OLGA stores all connected undirected unlabelled graphs with order up to 10 vertices. However the system can generate all connected unlabelled graphs with order up to 12 vertices.

We design recursive algorithms for parameters present in OLGA. However, for parameters for which we could not find a useful recursion, we computed them using recursive upper and lower bounds or using exact algorithms. If these bounds coincide we report that value as the value for the parameter. Otherwise in some cases we use an exact algorithm to compute the parameter value and mark the graph as an unmatched candidate under the recursive bounds. For some parameters for which we use recursive bounds, we did an empirical study on the proportion of graphs with unmatched bounds among all the graphs of a given order. We report our findings and propose a few questions based on our empirical data.

During this study, we introduced the problem of determining the Most Frequent Connected Induced Subgraph (MFCIS) of a graph, to challenge the limitations of OLGA. However MFCIS is theoretically rich and an interesting parameter on its own merits. We proved that finding a MFCIS of a graph is #P-hard. A framework to generate the MFCIS of all graphs (currently) up to 12 vertices has been developed. We computed and stored the MFCIS of all unlabelled graphs with order up to 10 vertices. We determine MFCIS of some special classes of graphs. We also proved that the order of MFCIS for a perfect binary tree is a large portion of the order of the tree.

We also demonstrate how OLGA can be used as a research tool to extract information about parameters that it contains.

Graph parameters: theory, generation and dissemination

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Srinibas Swain
July 23, 2019

Acknowledgments

First and foremost I would like to express my gratitude to my thesis advisor Professor Graham Farr. To me Graham personifies the word "professor" in its true sense. He took the pain and challenge of teaching precise mathematical writing to me. The virtue that separates Graham from other academics is his attitude towards everything. It's simply amazing and most of the times contagious. After every meeting with Graham I have always felt happy and intellectually enlightened. I must thank Graham for introducing me to complexity theory, which I thoroughly enjoyed. He is one of the few professors who works on the student's overall growth. I learned so much from him personally and professionally. This thesis would not have been possible without his herculean effort, unconditional support, unbounded patience, unlimited critical feedback and priceless experience of guidance. I must not forget to thank Graham for arranging the social events at his home. My exposure to Melbourne expanded immensely thanks to the discrete mathematics tours arranged by Graham. He is one of the best persons that I have met so far. I am extremely lucky to have Graham as my advisor.

This journey of PhD would not have been possible without my associate supervisor Dr. Kerri Morgan. What amazes me is she always has the time for her students. She knows the trick of injecting confidence in a student. Kerri has this uncanny ability of simplifying complicated concepts. I must thank Kerri for her selfless unconditional support. She is extremely friendly and caring as a person and as a supervisor.

This journey would have been incomplete without Professor Paul Bonnington. I thoroughly enjoyed my discussions with Paul. He is extremely patient. He has always given me his unconditional support. I must specially thank Paul for his support during 41 ACCMCC at Rotorua, New Zealand. Paul has been instrumental in providing me with technical assistance during my PhD.

I must thank Professor David Wood for teaching the beautiful course on graph theory. I also thank Professor Jessica Purcell for introducing me to topology. I am grateful to Professor Ian Wanless for co-organising the discrete mathematics seminars and for the badminton sessions.

I appreciate the constructive feedback from my panel members Dr. John Betts, Professor Balasubramaniam Srinivasan, Dr. Amin Sakzad, Dr. Ron Steinfeld and Associate Professor Chung-Hsing Yeh (panel chair) during my milestone reviews. I also thank Associate Professor Michael Brand for numerous interesting discussions on mathematics and computer science. I also thank Danette Deriane (Graduate Research Student Services Coordinator) for her support all these years.

The people who made my stay at Monash memorable and enjoyable are my friends. I owe a lot to Tenindra Abeywickrama (the first person I met at Monash), Harald Bögeholz (for the long interesting mathematical walks), Rui Chen, Nader Chmait, Hooman R.

Dehkordi, Michael Gill (for the algorithmic discussions), Xuanli He (my current neighbour), Bhagya Hettige, Mahmoud HossamEldeen, Ben R. Jones (for introducing many interesting games), Parthan Kasarapu, Arun P. Mani (for useful suggestions and advice), Ranjie Mo (my friendly former neighbour), Dhananjay Thiruvady (for the long chats and temple trips), Ruangsak Trakunphutthirak (the chef), Laksri Wijerathna, Kai Siong Yow (for the nice chats), Shenjun Zhong (for arranging badminton sessions) and many others for being with me in my highs and lows.

I take this opportunity to thank my father for teaching me the most valuable lessons of life. I must thank my mother for providing an unlimited supply of love, affection and support. I also thank my brother, brothers-in-law and my sisters for being a constant source of inspiration. I would like to give a big hug to my nephews Mrityunjaya and Pranshu.

Last but not least, I thank Nectar and Monash eResearch Centre for providing me with their computation facilities and Monash University for providing a Monash Graduate Scholarship (MGS). I also wish to thank the Faculty of Information Technology (Monash University) for the tuition fee waiver throughout my candidacy and for providing additional funding for the last six months of my candidacy.

Srinibas Swain

Monash University
July 2019

Chapter 1

Introduction

The history of graph theory may be traced to 1736, when the Swiss mathematician Leonhard Euler solved the Königsberg bridge problem [42]. In recent decades, graph theory has established itself as an important mathematical tool in a wide variety of subjects, ranging from operational research and chemistry to genetics and linguistics, and from electrical engineering and geography to sociology and architecture [87]. At the same time it has also become established as a sophisticated mathematical discipline in its own right.

Readily available graph data may help researchers in developing insight and intuition in graph theory. However generating and storing graph data is extremely challenging due to its large magnitude. The number of graphs grow exponentially with increasing number of vertices [66, 120]. The number of connected labelled graphs and connected unlabelled graphs of order < 13 are listed in Table 1.1.

No. of vertices	No. of connected labelled graphs	No. of connected unlabelled graphs
1	1	1
2	1	1
3	4	2
4	38	6
5	728	21
6	26704	112
7	1866256	853
8	251548592	11117
9	66296291072	261080
10	34496488594816	11716571
11	35641657548953344	1006700565
12	73354596206766622208	164059830476

Table 1.1: Growth of number of graphs [66, 120].

An aspect of graph theory that attracted major attention is functions of graphs that are invariant under isomorphism, known as parameters. Therefore it is useful to compute and store important parameters. Collection of graphs and parameters are a useful resource for graph theorists as it can provide assistance on their research.

The main outcome of this thesis is a large, complex, novel system, an *"Online Atlas of Graphs" (OLGA)*, which is not only a digital repository of graphs and parameter values but also queryable in a variety of ways.

This thesis studies the problem of designing recursive algorithms for graph parameters to be used in OLGA. In particular, it provides new empirical results about parameters that do not have a simple recursive structure. This thesis also discusses various design and implementation details of algorithms used for computing these parameters. In this process, we verified the correctness of some existing graph-data.

In this new computational age many discoveries are based on large amounts of data and efficient techniques to compute, analyse and retrieve it. This thesis is a nexus of these techniques and some theoretical discoveries. Therefore, this thesis may be classified as a mix of pure mathematics and computer science.

Section 1.1 motivates the study, describing three reasons for it. Section 1.2 gives a summary of the results obtained in this thesis. Finally, Section 1.3 contains a synopsis of the thesis.

1.1 Motivation

The most obvious reason for undertaking this study is that it produces a useful tool for researchers. Creating graph repositories is not new, it goes back to 1967, when Baker et al. [7] created the first printed graph repository. Since then there have been many attempts to create graph repositories, for details refer to Chapter 3. Some of these repositories have an exhaustive collection of graphs up to some size, possibly with values of some associated parameters, but they are not interactive (queryable). Others are interactive, but do not contain a comprehensive list of graphs up to some size. Therefore, a repository which is both interactive and also has an exhaustive list of graphs is novel and useful.

The usefulness of a graph repository is determined in part by the amount of information (in size and variety) it stores about graphs. Graph parameters are interesting and useful but often hard to compute. Many important parameters are NP-hard. In this project we exploit the inherent recursiveness of some of these parameters by storing their values for the smaller graphs and using that information to compute the parameter values for bigger graphs.

Not all graph parameters follow a nice recursive structure. In some cases one may still be able to get a recursive lower bound and upper bound on the parameter. If the bounds match then that value is the exact value, otherwise one has to find the exact solution by another method. This leads to an interesting question, "given a recursive upper bound and lower bound for a graph parameter, what is the probability that these bounds match?"

Since OLGA stores many graph parameters, some of which are computed using recursion and others using exact algorithms, the natural question is when to use recursion

(which needs look-ups[1], which is a costly operation) and when to use exact computation, which can take exponential time if the parameter is NP-hard.

1.2 Results

The results obtained in this thesis can be broadly divided into two parts. One part highlights the practical outcome from this work, whereas the other one outlines new theoretical discoveries. We summarise our results in Sections 1.2.1, 1.2.2 and 1.2.3.

1.2.1 OLGA updates

Barnes et al. [10] created the first prototype of OLGA in 2009. Barnes's version of OLGA used McKay's `nauty` [90] and it contained all unlabelled graphs with up to 9 vertices, although the system was capable of generating data up to 11 vertices. It contained the following parameters: domination number, matching number, circumference, diameter, radius, girth, clique number, vertex cover number and independence number. It was a very good first attempt based on the resources available then, however the design of this version was not scalable, as it uses static data structures and did not use parallel computation. We reuse some of the recursive algorithms developed by Barnes et al. [10], but we do not use any code from Barnes's system.

The current version of OLGA is the outcome of a scalable, portable design. The highlights of the current version are:

- We store all connected unlabelled graphs with up to 10 vertices. Our system can generate all connected unlabelled graphs up to 12 vertices.
- The current version of OLGA does not depend on any other software. However, we have enabled the use of `nauty` in a separate branch of OLGA to speed up the graph generation.
- Most of our algorithms used for computing parameters are recursive in nature. There are lot of repetitions in computation. Therefore to accelerate the computation, we adopted an approach which minimises the repetition in computation, for details refer to Chapter 5.
- Parameters for which we do not have a simple recursion, we implemented exact algorithms to compute their values and validated the results of the recursive algorithm with the output of the exact algorithm.
- We report our findings on the correctness of the recursion for those parameters that do not follow a simple recursive structure.
- We also verified some existing graph-data and produced some new data.

[1]When the size of data is so large that it cannot fit into main memory, then the data gets stored in secondary storage devices (disks). Any request to access data from the secondary storage is called an *I/O-look-up* or *look-up*

- For each graph, OLGA computes its degree sequence, number of connected components, girth, radius, diameter, independence number, clique number, vertex cover number, domination number, circumference, length of the longest path, size of the maximum matching, vertex connectivity, edge connectivity, chromatic number, chromatic index, treewidth, Tutte polynomial (for graphs of up to 7 vertices), size of automorphism group, genus, and eigenvalues.

- We designed an interface for querying OLGA. The queries can consist of multiple constraints on the parameters, e.g. list all graphs of order between 4 to 7 that have domination number at least 3 and treewidth at most 4 and independence number 4. We also provide the facility to download the information resulting from queries.

1.2.2 Most Frequent Connected Induced Subgraph (MFCIS)

We introduced a new parameter: Most Frequent Connected Induced Subgraph (MFCIS) to test the limits of our system. The study of MFCIS introduced interesting theoretical questions. A summary of our results on MFCIS is listed here.

- We proved that finding MFCIS is #P-hard.

- We determined MFCIS of some special classes of graphs.

- We found a large class of graphs whose order of MFCIS is a large proportion of the order of the graph.

1.2.3 OLGA analysis

We analysed some of the empirical data generated from the computation of parameters that uses recursive upper and lower bounds. Some results of our analysis are:

- For chromatic index, we counted the number of graphs for which the recursive upper bound and the recursive lower bound did not match. We observed as the order of the graphs increased the percentage of graphs with unmatched recursive upper and lower bounds decreased.

- For treewidth, as the order of the graphs increased the percentage of graphs whose recursive upper and lower bounds did not match decreased up to 9 vertices and increased slightly on 10 vertices.

- For degeneracy, the recursive upper and lower bounds matched for all graphs. We confirmed our results up to 12 vertices by comparing them with the output of an exact algorithm.

We manipulated some portions of OLGA code to verify some existing conjectures. We also demonstrated how OLGA can be used as a research tool by generating critical graphs (discussed in Chapter 8).

We listed graphs satisfying some interesting properties related to chromatic number and MFCIS. We also list some open questions resulting from the analysis of OLGA data.

1.3 Synopsis of the thesis

In this thesis we consider simple connected undirected graphs, unless otherwise stated.

Chapter 2 provides the required definitions and concepts used in this thesis.

We give a thorough literature review on graph repositories in Chapter 3. In Section 3.2.1 we describe the significant printed graph repositories. In Section 3.2.2 we discuss the static electronic graph repositories and in Section 3.2.3 we discuss the interactive electronic graph repositories. We conclude the chapter with some recommendations on which repository is best suited for which kind of data.

Chapter 4 outlines the theoretical background and potential applications of OLGA. In Section 4.1 we demonstrate how OLGA may be used to gather information about conjectures, open questions and new theorems. In Section 4.4 we propose a potential use of OLGA-generated data in machine learning. Finally, in Section 4.5 we conclude by linking the OLGA recursive approach to self-reducibility from computational complexity theory.

Implementation issues are discussed in Chapter 5. We discuss the main operations used for parameter computation in Section 5.3. We also briefly introduce MFCIS in this chapter. In Section 5.3.1 we make a comparison between I/O-bound and CPU-bound jobs with reference to OLGA. We conclude the chapter discussing query optimisation and OLGA architecture design.

The recursions and the recursive algorithms used in OLGA are discussed in Chapter 6. Section 6.1 discusses the recursions used in computing metric parameters like radius, diameter, girth, etc. The algebraic properties like eigenvalues, size of automorphism group and the Tutte polynomial are discussed in Section 6.2. Recursive relations for structural properties like clique number, treewidth, independence number, etc. are discussed in Section 6.3. We conclude the chapter by discussing the recursive structure of chromatic number, which is different to all other recursions discussed in this chapter.

We discuss MFCIS in detail in Chapter 7. In Section 7.2 we determine MFCIS for special classes of graphs. The #P-hardness of MFCIS is established in Section 7.3. In Section 7.4 we proved the order of the MFCIS for perfect binary tree is a large portion of the order of the tree. We conclude the chapter with some open questions related to MFCIS.

In Chapter 8 we demonstrate an extended use of OLGA in identifying chromatic critical graphs in Section 8.1. We also listed these critical graphs based on their chromatic number. We depict some specific graphs which have multiple MFCIS. We conclude the chapter by suggesting the potential use of OLGA in exploring other parameters.

We conclude the thesis by suggesting some natural extensions of OLGA in Section 9.2. We list some interesting questions about MFCIS in Section 9.3. We suggest some future work on the parameters that do not have a simple recursive structure in Section 9.4.

Chapter 2

Definitions and notation

In this chapter we define the necessary notation and definitions used in this thesis across different chapters. Most of them are widely used in the field of graph theory. For a good introduction to graph theory, we refer the reader to [18, 42, 152].

2.1 Graph theory

Let G be a graph. The vertex set of G is denoted by $V(G)$ (or V) and the edge set of G by $E(G)$ (or E). We also refer to $|V|$ as the *order* of G. We write $G = (V, E)$. We use $|G|$ to denote the order of G. Note $|V|$ is defined on a set whereas $|G|$ is defined on the graph G.

The *falling factorial* $\lambda^{(n)}$, is defined by $\lambda^{(n)} = \lambda(\lambda-1)(\lambda-2)\cdots(\lambda-(n-1))$. The number of all possible edges over the set of n vertices is $n^{(2)}/2$.

Two vertices $v, w \in V(G)$ are *adjacent* if $\{v, w\} \in E(G)$. We also use vw interchangeably for representing an edge. We say that a vertex $v \in V(G)$ is *incident* to an edge $e \in E(G)$ if $v \in e$. Two edges are called *incident* if they share a vertex. If all vertices of G are pairwise adjacent, then G is *complete*. Let $V^{(2)}$ denote the set of all unordered pairs of distinct elements of V. A complete graph on n vertices is denoted by K_n.

A graph G is called *bipartite* if its vertex set V can be partitioned into two disjoint sets V_1 and V_2 such that every edge in the graph joins a vertex in V_1 and a vertex in V_2 (so that no edge in G joins either two vertices in V_1 or two vertices in V_2).

A *complete bipartite* graph $G = (V_1 \cup V_2, E)$ is a bipartite graph with parts V_1 and V_2 such that for any two vertices $v_1 \in V_1$ and $v_2 \in V_2, \{v_1, v_2\}$ is an edge in G. The complete bipartite graph with parts of order $|V_1| = m$ and $|V_2| = n$ is denoted by $K_{m,n}$.

The *degree* of a vertex $v \in V(G)$ is the number of edges which v is incident to and is denoted by $\deg_G(v)$ (or simply $\deg(v)$ or $d(v)$). The minimum and maximum degree of vertices in G are denoted by $\delta(G)$ and $\triangle(G)$, respectively. A graph G is called d-regular if $\delta(G) = \triangle(G) = d$. A 3-regular graph is known as a *cubic* graph. The set of neighbours (i.e., adjacent vertices) of a vertex $v \in V(G)$ is denoted by $N_G(v)$ (or $N(v)$).

A graph G' is a *subgraph* of G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$ and we denote this by $G' \leq G$. If G' is a subgraph of G and $V(G') = V(G)$, then G' is a *spanning subgraph* of G. If G' is a subgraph of G and for all $v, w \in V(G')$ we have $\{v, w\} \in E(G) \Rightarrow \{v, w\} \in$

$E(G')$, then G' is called an *induced subgraph* of G. We also refer to G' as the subgraph of G *induced* by the set of vertices $X = V(G')$ and denote it by $G[X]$.

If G' is a subgraph of G, then G is also called a *supergraph* of G'. The *complement* of a graph $G = (V, E)$ (denoted by G^c), is the graph $G^c = (V, V^{(2)} \setminus E(G))$.

A *walk* in a graph is an alternating sequence

$$v_0, e_1, v_1, e_2, v_2, \cdots, e_k, v_k$$

where $v_0, v_1, v_2, \cdots, v_k$ are vertices, $e_1, e_2, \cdots, e_k$ are edges, and $e_i = \{v_{i-1}, v_i\}$ for $1 \leq i \leq k$. This is called a (v_0, v_k)-*walk* of *length* k. The *length* of a walk is the number of edges in it. A single vertex is a walk of length 0. A *path* is a walk in which no vertex is repeated.

G is a *cycle* C_n of order n, if it has n vertices and these can be ordered $v_0, v_1, \cdots, v_{n-1}$ so that $E(C_n) = \{\{v_i, v_{i+1}\} : 0 \leq i < n-1\} \cup \{\{v_0, v_{n-1}\}\}$. The *length* of the cycle is its number of edges.

If $\{u\} \notin V(G)$, then $G \cup \{u\}$ is defined as $(V(G) \cup \{u\}, E(G))$. Similarly if $\{uv\} \notin E(G)$ is an edge then $G \cup \{uv\}$ is defined as $(V(G), E(G) \cup \{uv\})$. We use $G \setminus u$ to denote the graph induced by $V(G) \setminus \{u\}$.

The *subdivision* of an edge $e = \{u, v\}$ in a graph $G = (V, E)$ yields a graph G' containing one new vertex w, and e replaced by two new edges $\{u, w\}$ and $\{w, v\}$, i.e., $V(G') = V(G) \cup \{w\}$ and $E(G') = E(G) \setminus \{e\} \cup \{\{u, w\}, \{w, v\}\}$.

The graph G/e is the graph obtained by *contracting* an edge $e = \{v_i, v_j\}$ in the graph G. This *edge contraction* introduces a new vertex w and new edges such that w is adjacent to all the vertices in $N(v_i) \cup N(v_j)$ and deletes vertices v_i and v_j and all edges incident to v_i or v_j.

A graph H is a *minor* of a graph G if H can be obtained from G by contracting edges and by removing edges and vertices.

Two graphs G and G' are *isomorphic* (denoted by $G \cong G'$) if there is a bijective function $\phi : V(G) \to V(G')$ such that $\{v, w\} \in E(G) \iff \{\phi(v), \phi(w)\} \in E(G')$. The function ϕ is called an *isomorphism* from G to G'. An isomorphism from G to itself is called an *automorphism*. The composition of two automorphisms is another automorphism, and the set of automorphisms of a given graph, under the composition operation, forms a group, the *automorphism group* $\mathrm{Aut}(G)$ of the graph.

The *adjacency matrix* A of a graph G of order n is an $n \times n$ matrix with rows and columns indexed by $V(G)$, for which $A_{u,v} = 1$ if $\{u, v\} \in E$, and $A_{u,v} = 0$ otherwise.

A graph $G = (V, E)$ is *connected* if for every $v, w \in V$ there is a path from v to w. A *disconnected* graph is a graph which is not connected. Given a connected graph $G = (V, E)$, a set $S \subseteq V$ is a *vertex-cut* if $G[V \setminus S]$ is disconnected. If a vertex-cut only contains one vertex, we call that vertex a *cut-vertex*. A graph G is k-*connected* if G is connected and no vertex-cut S exists with $|S| < k$.

A *tree* is a connected graph with no cycles in it. A *block* of a graph is a maximum connected subgraph with no cut vertex, i.e., a subgraph with as many edges as possible

and no cut vertex. So a block is either K_2 or is a graph which contains a cycle. For example in a tree, every block is K_2.

Given a connected graph $G = (V, E)$, a set $T \subseteq E$ is an *edge-cut* if the graph $G' = (V, E \backslash T)$ is disconnected. If the edge cut only contains one edge, we call that edge a *bridge*. A graph G is k-edge-connected if G is connected and no edge-cut T exists with $|T| < k$. We call an (edge) cut T with $|T| = k$, a k-(edge-)cut. Thus, if a graph is *k-edge-connected* but not $(k+1)$-edge-connected, it has a k-edge-cut.

A *connected component* (or simply a *component*) of a graph G is a maximal connected induced subgraph of G. Here maximal means that the connected subgraph is not contained in any larger connected subgraph.

The *rank* of a graph G is defined as $r(G) = n - c$, where n is the number of vertices in G and c is the number of connected components in G.

A *proper colouring* of a graph G is a function assigning to each vertex a colour where no two adjacent vertices in G are assigned the same colour [42]. The *chromatic number*, denoted $\chi(G)$, is the minimum number of colours needed to colour the vertices of G such that no two adjacent vertices share the same colour. A proper colouring which uses at most λ colours is called a *λ-colouring*. The *chromatic polynomial* $P(G, \lambda)$ counts the number of λ-colourings of a graph G [14].

A *k-(vertex-)colouring* of a graph G is a proper colouring of the vertices of G using k colours.

An *edge-colouring* of a graph G is a function assigning to each edge a colour where no two incident edges in G are assigned the same colour [42]. The *chromatic index* $\chi'(G)$ of a graph G is the minimum number of colours required for an edge-colouring.

An *independent set* of G is a set of pairwise non-adjacent vertices. That is, a set $I \subseteq V(G)$ is independent if no edge of G has both endpoints in I. Let $\alpha(G)$ be the maximum size of an independent set in G, called the *independence number* of G.

A *clique* of a graph G is a set of pairwise adjacent vertices. That is, a set $K \subseteq V(G)$ is a clique if every pair of vertices of K is joined by an edge. Let $\omega(G)$ be the maximum size of a clique in G, called the *clique number* of G.

A *vertex cover* of a graph G is a set of vertices such that each edge of the graph is incident to at least one vertex of the set. That is, a set $S \subseteq V(G)$ is a vertex cover if every edge of G has at least one endpoint in S. Let $\mathrm{vc}(G)$ be the minimum size of a vertex cover in G, called the *vertex cover number* of G.

2.2 Complexity theory

In this section we give complexity theory definitions which are useful to this project. We refer the reader to [3, 55] for a good introduction to complexity theory.

A *language* L is a set of strings defined over an alphabet set Σ. The *decision problem* for a language L is the problem of deciding whether a given input x belongs to L. The decision problem for L is: given x, determine if $x \in L$. The *answer* for x is "YES" if $x \in L$, and "NO" otherwise.

A *Turing machine* is a hypothetical machine introduced by Alan Turing in 1936 [143]. A Turing machine is an abstract model of computation. It is a type of abstract machine that, given an arbitrarily large amount of space and time, can be programmed to compute any computable function. As per the Church-Turing thesis, the machine can simulate "any" computational algorithm, no matter how complicated it is.

A *verifier* for a language L is a Turing machine M, taking as input a pair of strings, such that $L = \{x : M(x, w) = 1, \text{ for some string } w\}$. M is called a *polynomial time verifier* if M runs in polynomial time. M outputs only 0 or 1.

An *oracle* for a language A is a conceptual device that is capable of reporting whether any string w is a member of A. An *oracle Turing machine* M^A is a Turing machine that has access to an oracle for consultation. Whenever M^A writes a string on a special *oracle tape* it is informed whether that string is a member of A, in a single computation step.

An *oracle* for a function f is a conceptual device that is capable of returning $f(x)$ for any string x. An *oracle Turing machine* M^f is a modified Turing machine that has the additional capability of querying an oracle. Whenever M^f writes a string x on a special *oracle tape* it is informed with $f(x)$, in a single computation step.

A *polynomial time reduction*, also known as a *Karp-reduction*, from a decision problem X to a decision problem Y is an algorithm $\mathcal{A}$ that has the following properties:

- Given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y.
- $\mathcal{A}$ runs in time polynomial in $|I_X|$. This implies that $|I_Y|$ is bounded above by a polynomial in $|I_X|$.
- The answer to I_X is YES if and only if the answer to I_Y is YES.

Notation: $X \leqslant_P Y$ if there is a polynomial time reduction from X to Y.

Language A is *polynomial time Turing reducible* to language B, written $A \leqslant_T B$, if A is decidable relative to B, that is, if there is an oracle Turing machine M which accepts A in polynomial time using an oracle that decides B.

L is in P if L is decidable in polynomial time by a deterministic Turing machine.

L is in *NP (nondeterministic polynomial time)* if there exists a polynomial time verifier M for L, such that $x \in L$ if and only if there exists w such that $M(x, w) = 1$. Such a string w is called a *certificate* for x in L.

L is in *co-NP*, if $\overline{L} = \Sigma^* \setminus L$ is in NP.

A language L' is in *NP-hard* if there exists a polynomial time Turing reduction from every language L in NP to L'.

L is *NP-complete* if $L \in$ NP and every $L' \in NP$ is polynomial time reducible to L.

L is in D^P if $L = L_1 \cap L_2$ for some $L_1 \in$ NP, $L_2 \in$ co-NP.

The complexity class $\#P$ was first defined by Leslie Valiant in 1979 [149]. A function $f : \{0,1\}^* \to \mathbb{N} \cup \{0\}$ is in #P if there exists a polynomial p and a polynomial-time verifier M such that for every $x \in \{0,1\}^*$:

$$f(x) = |\{y \in \{0,1\}^{p(|x|)} : M(x, y) = 1\}|.$$

$f(x)$ counts the number of certificates y for which $M(x, y) = 1$.

Given two functions $f, g : \{0,1\}^* \to \mathbb{N} \cup \{0\}$, we say that there is a *polynomial time Turing-reduction* from g to f (denoted by $g \propto f$) if the function g can be computed in polynomial time using an oracle for f. The function f is *#P-hard* if for every $g \in \#P$ there is a polynomial time Turing reduction $g \propto f$. A function f is *#P-complete*, if $f \in \#P$ and f is #P-hard.

Chapter 3

A survey of graph repositories

This chapter outlines some of the useful graph repositories and summarises the contents of each repository. We also describe the scope and limitations of each graph repository. We conclude the chapter recommending which repositories are best suited for which type of data.

3.1 Introduction

A *graph parameter* is defined as a function $f : \{\text{graphs}\} \to \mathbb{Z}$ invariant under isomorphism. We sometimes extend the term to include non-integer-values like complex numbers and polynomials. Some examples of graph parameters are chromatic index, treewidth, and the Tutte polynomial. Finding the chromatic index is known to be NP-hard [55]. The chromatic index is useful in scheduling problems [76] and routing problems [30]. Treewidth was introduced by Robertson and Seymour in 1983 [124]. Treewidth is discussed in detail in [46]. Treewidth plays an important role in parameterized complexity [46]. Many graph theoretic problems which are NP-complete are polynomial time solvable when restricted to bounded treewidth [46]. The problem of finding the treewidth of a graph is NP-hard [55]. The Tutte polynomial is a two-variable polynomial introduced by W. T. Tutte [146]. It plays a key role in the study of counting problems on graphs, and has close connections with statistical mechanics and knot theory [29, 58, 104].

A repository of graphs and their related parameters will provide a framework that will enable researchers to test new theorems and conjectures and may also facilitate new mathematical discoveries related to the data present in the repository.

Some graph parameters are computable in polynomial time, some take exponential time and some of them are uncomputable [3, 55]. There are thousands of graph parameters that are of interest to researchers. This makes the existence of any graph repository that contains all published graph parameters impossible. Several researchers have created useful graph repositories for specific graph parameters. Some of the major challenges in creating these repositories are computing the parameters, representing them and storing the data. Therefore, these repositories can be broadly classified as repositories before and after the advent of magnetic media.

3.2 Graph repositories

In this section, we discuss the various commonly used graph repositories. We discuss the parameters that each of these repositories contains and also list the limitations of each of these repositories.

Note that the information listed in this Chapter are based on the time it had appeared in a repository, although in some cases the process of generating these data might have started prior to the time the data appeared in the repositories. We list some of the attempts at data generation in Section 3.2.1.

3.2.1 Print resources

The following repositories of the late twentieth century used printed form to store information. A large collection of graph data is very difficult to present in this form because of the inherent limitation of print media. Despite this limitation, these repositories were quite significant for specific problems and very well structured. Although the repositories listed in this section are quite rich in content, it is hard to search, insert, update or extend data in these type of repositories. These repositories fail to handle queries which look up some range of parameter values or combination of several parameters, e.g. list all connected graphs whose diameter is between 4 and 7 with chromatic index 5.

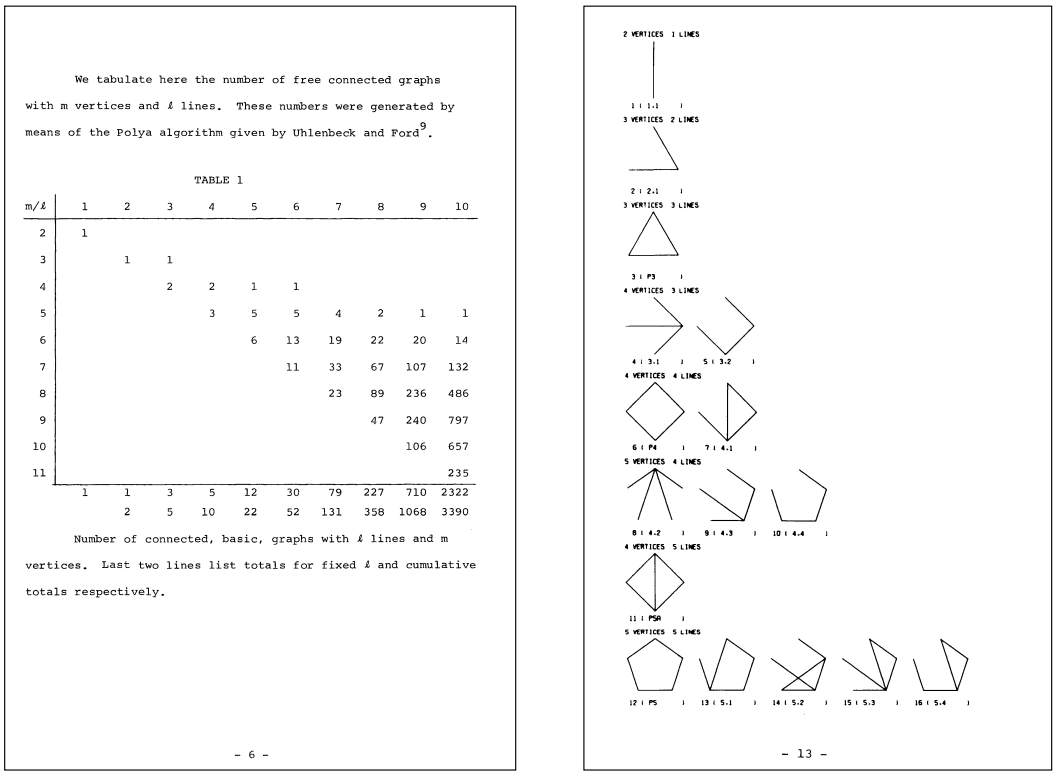
Initial attempts (1946 – 1976)

In this section we list the initial attempts at graph generation. The first such attempt was reported by Kagno [78] in 1946. He attempted to generate graphs with order up to 6. Subsequently Heap [69], Baker et al. [8] attempted to generate graphs with order up to 8 and 9 respectively. Similarly Read [116] attempted to generate digraphs with order up to 5. Morris [100] and Frazer [95] attempted to generate trees with order up to 13 and 18 respectively. McWha [53] attempted to generate tournaments with order up to 7 and Morris [101] attempted to generate self-complementary graphs with order up to 9. Bussmaker et al. [28] attempted to generate cubic graphs with order up to 14.

Understandably, all the above mentioned lists miss out on some of the graphs of the order they were trying to generate, due to limitations on computation [117]. However, these attempts were useful as they were the first ones to use computers to generate graphs and later provided information on designing algorithms for efficient graph generation.

Data compendium of linear graphs (1967)

In 1967, Baker, Gilbert, Eve and Rushbrooke published a list of 1460 simple connected sparse graphs with number of edges ≤ 10 and order ≤ 11 [7]. They generated these graphs for a project aimed at computing high-temperature expansions for the field dependent free energy of a Heisenberg model ferromagnet. It took three years of computation to accumulate this data. Although this list of graphs reported by Baker et al. is not the complete set of graphs of order ≤ 11, it still had lot of practical significance in the field



We tabulate here the number of free connected graphs with m vertices and ℓ lines. These numbers were generated by means of the Polya algorithm given by Uhlenbeck and Ford[9].

TABLE 1

m/ℓ	1	2	3	4	5	6	7	8	9	10
2	1									
3		1	1							
4			2	2	1	1				
5				3	5	5	4	2	1	1
6					6	13	19	22	20	14
7						11	33	67	107	132
8							23	89	236	486
9								47	240	797
10									106	657
11										235
	1	1	3	5	12	30	79	227	710	2322
		2	5	10	22	52	131	358	1068	3390

Number of connected, basic, graphs with ℓ lines and m vertices. Last two lines list totals for fixed ℓ and cumulative totals respectively.

- 6 -

- 13 -

Figure 3.1: An extract from the report [7].

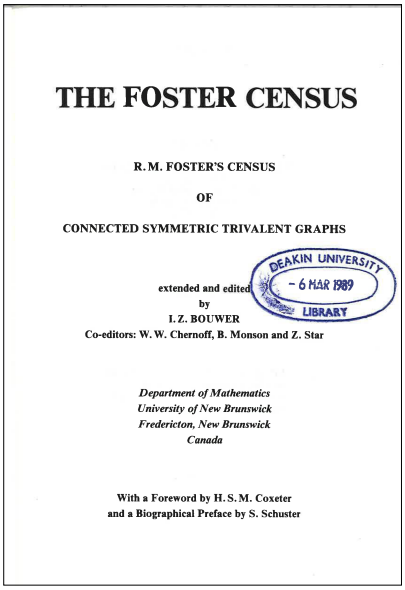
of cooperative phenomena [45, 73]. An example of the data presented in [7] is given in Figure 3.1.

Given the resources available at that time this repository, although small, was a valuable resource and one of the first few repositories of graphs. However, this repository does not store any graph parameters.

The Foster Census (1930–1988)

Although the data compendium by Baker et al. was published in 1967, Ronald M. Foster started storing graph data decades before the compendium was published. In the 1930s, Foster started collecting specimens of small cubic symmetric graphs while Foster was employed by Bell Labs [51]. Interestingly, Foster's list was hand-prepared, so understandably had missed some graphs. Foster's hand-prepared list was surprisingly accurate and had only one omission for graphs of order up to 240. On the other hand, for order between 240 and 512 there were several other omissions. In 1988, when Foster was 92, the Foster Census listing all cubic symmetric graphs up to 512 vertices was compiled by I. Z. Bouwer, W. W. Chernoff, B. Monson and Z. Star ("The Foster Census" [51]). The cover page of the Foster Census and a picture of Foster is shown in Figure 3.2.

The missing graphs and the discovery:
Conder and Dobcsányi [36] found all the missing graphs from the Foster Census [128] while they were compiling all cubic symmetric graphs with order up to 768.


THE FOSTER CENSUS

R. M. FOSTER'S CENSUS

OF

CONNECTED SYMMETRIC TRIVALENT GRAPHS

extended and edited
by
I. Z. BOUWER
Co-editors: W. W. Chernoff, B. Monson and Z. Star

Department of Mathematics
University of New Brunswick
Fredericton, New Brunswick
Canada

With a Foreword by H. S. M. Coxeter
and a Biographical Preface by S. Schuster

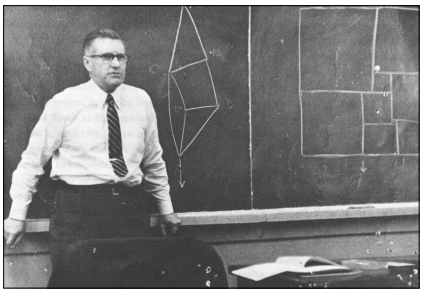


Figure 3.2: Foster's Census with Foster, from [51].

There was a graph among these missing graphs (448C, as per Foster's convention) which was the smallest graph that is arc-transitive but has no involutory automorphism reversing an arc. Interestingly, the previous smallest known graph of this type was a graph with 6652800 vertices [36]. This illustrates the research benefits of compiling and using graph repositories.

We will discuss more about the Foster Census in the magnetic media section. The first version of the Foster census not only had the list of cubic symmetric graphs, but also stored some useful graph parameters like diameter, girth, s-transitivity, hamiltonicity and bipartiteness of each listed graph.

An Atlas of Graphs (1988)

As we noted in the previous sections, all published graph repositories were narrow in scope and unidirectional. The first comprehensive graph data resource was Read and Wilson's "An Atlas of Graphs" [120]. This repository has all graphs with up to 7 vertices and all digraphs with up to 5 vertices. For each graph, Read and Wilson included values of some parameters and also gave information on whether or not certain properties hold. The Atlas also contains some families of graphs. We list some of the graphs and graph parameters from the book.

- **Simple graphs and parameters**
  This book not only lists all the unlabelled simple graphs of order ≤ 7 but also depicts the graphs. The graphs are listed in increasing order of number of vertices. For a fixed number of vertices the graphs are listed in increasing order of edges. For fixed number of vertices and edges, the graphs are listed in increasing order of degree sequence.

For each of the aforementioned graphs the book lists the following graph parameters: order, number of edges, degree sequence, number of connected components, girth, number of cycles of shortest length, diameter, clique number, independence number, vertex connectivity, edge connectivity, number of automorphisms, complement, whether or not certain properties hold (like bipartiteness, Eulerian, forest, Hamiltonian, planar, tree, uniquely colourable), chromatic number, chromatic index, chromatic polynomial and spectral polynomial.

- **Trees**
  The book lists trees, rooted trees, homeomorphically irreducible trees and identity trees of order ≤ 30. It also depicts the 987 trees with up to 12 vertices (together with their degree sequences), homeomorphically irreducible trees with up to 16 vertices, the identity trees up to 14 vertices and binary trees with up to 7 vertices. It also lists some of the parameters of trees: order, number of edges, degree sequence, diameter, independence number, number of automorphisms of the graph, properties (like bicentral, bicentroidal, central, centroidal, homeomorphically irreducible, identity tree) and spectral polynomial.

- **Regular graphs**
  It lists the number of labelled cubic and connected cubic graphs up to 40 vertices and 4-regular graphs up to 15 vertices. It also depicts the connected cubic graphs with up to 14 vertices, 4-regular graphs up to 11 vertices, and 5-regular and 6-regular graphs up to 10 vertices. It also depicts the connected bicubic graphs with up to 16 vertices and the cubic polyhedral graphs (without triangles) with up to 18 vertices, connected cubic transitive graphs with up to 34 vertices, 4-regular transitive graphs with up to 19 vertices and symmetric graphs with up to 54 vertices. The book also lists all the parameters for the regular graphs which are listed for simple graphs except the graph polynomials.

- **Other classes of graphs**
  The scope of this book with respect to variety of graphs is wide. This book lists the number of bipartite graphs, connected bipartite graphs, unicyclic graphs and self-complementary[1] graphs with up to 20 vertices. It also listed even graphs[2], Eulerian graphs, and connected line graphs up to 16 vertices. This lists all Hamiltonian graphs up to 11 vertices.

  The book also depicts all connected bipartite graphs with up to 8 vertices, Eulerian graphs with up to 8 vertices, self-complementary graphs up to 9 vertices, connected triangle-free graphs up to 10 vertices (none of degrees less than 3), connected line graphs up to 8 vertices and unicyclic graphs up to 8 vertices.

- **Planar graphs**
  The authors depict planar graphs as proper planar embeddings marking the interior

[1] A self-complementary graph is one that is isomorphic to its complement.

[2] A connected graph G is called *even* if for each vertex v of G there is a unique vertex u such that $d(v, u) = \text{diam } G$

and exterior faces. The book depicts 2-connected plane graphs up to 7 vertices and 3-connected graphs with up to 8 vertices together with their respective degree sequences. The book also depicts the outerplanar graphs with up to 9 vertices.

- **Digraphs**

  The book is a repository of digraphs as well. It lists various types of digraphs (connected, unilateral, strong, acyclic, self-complementary, self-converse and tournaments) with up to 11 vertices. The digraphs are listed in increasing order of number of vertices. For a fixed number of vertices, they are listed in increasing order of number of arcs. For fixed number of vertices and arcs, digraphs are listed in increasing order of degree sequences. For fixed numbers of vertices and arcs, and fixed degree sequences, they are listed in order of increasing number of automorphisms.

  The book depicts the acyclic digraphs with up to 5 vertices. It also depicts Eulerian digraphs up to 5 vertices, 2-regular digraphs up to 7 vertices, self-complementary digraphs up to 5 vertices and tournaments up to 7 vertices. It also differentiates the strong tournaments among the depicted tournaments. Weakly connected transitive digraphs up to 5 vertices are also depicted.

  Along with the digraphs the book also lists some important graph parameters associated with them. The parameters listed for digraphs are: number of vertices, number of arcs, out-degree sequence and in-degree sequence, connectivity (whether digraph is disconnected, connected but not unilateral, unilateral but not strong or strong), number of automorphisms and whether or not certain properties hold (acyclic, Eulerian, Hamiltonian, self-complementary, tournament, self-converse).

- **Signed graphs**

  This book is one of the few repositories that lists and depicts signed graphs. It lists signed graphs, balanced signed graphs and signed trees with up to 12 vertices. The book depicts signed graphs with up to 5 vertices and signed trees with up to 7 vertices.

  The book lists parameters including number of vertices, number of edges (number of positive and negative edges), number of 3-cycles, number of 4-cycles, and number of 5-cycles, and also specifies whether the signed graph is balanced or unbalanced for all the signed graphs.

- **Graphs and Ramsey numbers**

  Ramsey numbers play an important role in graph theory and in probabilistic methods. This book gives the Ramsey numbers for some pairs of connected graphs with up to 5 vertices. It also depicts all isolate-free graphs[3] with up to 7 edges, with their Ramsey numbers. Each depiction also contains the serial number of the graph in Burr's catalogue [27]. The authors also depict graphs with more than 7 edges for which the Ramsey number is known.

[3] A graph is *isolate-free* if it has no isolated vertex

- **Polynomials**

  Graph polynomials can be useful as some of them encapsulate a lot of information about the graph. Read and Wilson listed the chromatic polynomials of graphs (up to 7 vertices), cubic graphs (up to 14 vertices), and 4-regular graphs (up to 11 vertices). Each polynomial is presented in the *power form* (in decreasing powers of λ), in sum of falling factorial form, and in *tree form*, which for a graph with k components, is the polynomial P such that $\lambda^k P(\lambda - 1)$ is the chromatic polynomial.[4]

  The authors also list the characteristic polynomials of graphs (up to 7 vertices), trees (up to 12 vertices), cubic graphs (up to 14 vertices) and 4-regular graphs (up to 11 vertices). For graphs and trees, each polynomial is presented in power form, together with its frequency and spectrum (usually with zero and integer eigenvalues listed first, and the remainder listed in decreasing order); for cubic and 4-regular graphs only the polynomial is given.

- **Special graphs**

  There are some graphs which are considered special due to some specific properties that these graphs satisfy. They play an important role in theorem and conjecture verification. The authors of this book lists some of them. The special graphs listed and depicted in this book are *platonic graphs, Archimedean graph, Möbius graph, cages, non-Hamiltonian cubic graphs, generalised Petersen graphs, snarks, graphs drawn with minimum crossings, the two smallest cubic identity graphs, the hypercube, the Greenwood-Gleason graph, cubic graphs with no perfect matching, the Goldner-Harary graph, the Biggs-Smith graph, Folkman's graph, Tietze's graph, Meredith's graph, Chvátal's graph, Franklin's graph, the Moser spindle, the Herschel graph, Mycielski's graph or Grötzsch's graph, and Royle's graph.* All these graphs are defined in Chapter 6 of the book [120].

We discussed the limitations of print media in Section 3.2.1. "An Atlas of Graphs" is the most comprehensive printed graph repository. A summary of the contents of "An Atlas of Graphs" is given in Tables 3.1 and 3.2. These tables demonstrate the richness of this repository. Searching for a graph that the book contains is straightforward. However, constructing a collection of graphs satisfying some specific conditions can be a laborious task with any printed repository. If the data present in "An Atlas of Graphs" is represented in electronic media, searching and constructing data would be easier and efficient.

[4] Read and Wilson [120] give the example of chromatic polynomial of the butterfly graph G, with $V(G) = \{0, 1, 2, 3, 4\}$ and $E(G) = \{01, 02, 03, 04, 12, 34\}$, is presented in different forms:

- Power form: $\lambda^5 - 6\lambda^4 + 13\lambda^3 - 12\lambda^2 + 4\lambda$
- Falling factorial form: $\lambda^{(5)} + \lambda^{(4)} + \lambda^{(3)}$
- Tree form: $\lambda P(\lambda - 1) = \lambda((\lambda - 1)^4 - 2(\lambda - 1)^3 + (\lambda - 1)^2) = \lambda(\lambda - 1)^4 - 2\lambda(\lambda - 1)^3 + \lambda(\lambda - 1)^2$, So $P(x) = x^4 - 2x^3 + x^2$.

Graph type	Order	Exhaustive list (Y/N)
Unlabelled simple	1–7	Y
Trees	1–12	Y
Homeomorphically irreducible trees	1–16	Y
Identity trees	7–14	Y
Binary trees	1–7	Y
Connected cubic	4–14	Y
Connected 4-regular	5–11	Y
Connected 5-regular	6–10	Y
Connected 6-regular	7–10	Y
Connected bicubic	4–16	Y
Connected polyhedral	8–18	Y
Connected cubic transitive	4–34	Y
Connected 4-regular transitive	5–19	Y
Symmetric cubic	4–54	Y
Connected bipartite	2–8	Y
Eulerian	1–8	Y
Self-complementary	4–9	Y
Connected triangle-free	6–10	Y
Connected line	1–8	Y
Unicyclic	3–9	Y
Plane 2-connected	3–7	Y
Plane 3-connected	4–8	Y
Outerplanar	3–9	Y
Digraphs	1–4	Y
Acyclic digraphs	1–5	Y
Eulerian digraphs	1–5	Y
2-regular digraphs	3–7	Y
Self-complementary digraphs	1–5	Y
Tournaments	1–7	Y
Weakly connected transitive digraphs	1–4	Y
Signed	1–5	Y
Signed trees	1–7	Y
Other known graphs	–	N

Table 3.1: A summary of graphs in “An Atlas of Graphs”.

Type of graph(s)	List of parameters
Unlabelled graph, tree, Regular graph	automorphism group size, diameter, independence number, degree sequence, spectral polynomial
Unlabelled graph, Regular graph	girth, circumference, tree, clique number, chromatic number, bipartiteness, chromatic index, chromatic polynomial, uniquely colourable, Eulerianness, planarity, Hamiltonicity, number of shortest-length cycles, vertex-connectivity, edge-connectivity
Unlabelled graph	number of components, tree
Tree	bicentral, bicentroidal, central, centroidal, identity tree, homeomorphically irreducible
Digraph	acyclic, disconnected, connected but not unilateral, unilateral but not strong, strong, Eulerianness, Hamiltonicity, degree sequence (out-degree and in-degree), automorphism group size, self-complementary, tournament, self-converse
Signed graph	number of edges (positive, negative), 3-cycle, 4-cycle, 5-cycle, balanced or unbalanced

Table 3.2: A summary of parameters in "An Atlas of Graphs".

3.2.2 Electronic resources

The limitations of printed form, the challenges in generating and storing graph data of higher order graphs prompted researchers to build electronic representations of graph data that can be accessed easily.

As discussed, one of the major challenges in creating a graph repository is the representation of graphs. A straightforward way to store a graph in electronic form is via its adjacency matrix, which requires n^2 bits to store a graph of order n. McKay introduced `graph6` and `sparse6` formats for storing undirected graphs in a compact manner [90], using only printable ASCII characters obtained from bit manipulation of the adjacency matrix. Files in these formats have text type and contain one line per graph. The format `graph6` is suitable for small graphs, or large dense graphs, while the format `sparse6` is more space-efficient for large sparse graphs.

Many graph theorists generate various sets of graph data in the course of their research. Among the researchers, substantial contributions of this type have been made by Conder [34], McKay [91] and Royle [127].

Brendan McKay's combinatorial data (1984 – present)

McKay's collection is one of the largest available repositories of graphs. It contains more than 150 million graphs in total [91]. McKay's collection of graphs is interesting and useful for its variety and magnitude. We list various graphs from McKay's repository below.

- **Simple graphs**
  McKay stored all connected unlabelled graphs of order ≤ 10, and generated all such graphs of order ≤ 16. McKay generated these graphs according to their numbers of edges and vertices. The program used to generate these graphs is `geng` [91].

- **Special class of graphs**

  McKay's repository contains all Eulerian graphs (and all connected Eulerian graphs) and chordal graphs up to 12 vertices.

  A graph is *perfect* if every odd cycle of length at least 5 has a chord and the same is true of the complement graph. McKay's repository contains all perfect graphs up to 11 vertices.

  Let G be a regular graph with n vertices and degree k. G is said to be *strongly regular* if there exist integers λ and μ such that:

  – every two adjacent vertices have λ common neighbours;
  – every two non-adjacent vertices have μ common neighbours.

  A graph of this kind is sometimes said to be an SRG(n, k, λ, μ). Strongly regular graphs were introduced by Raj Chandra Bose in 1963 [21]. McKay's collection of strongly regular graphs is one of the most comprehensive lists of strongly regular graphs. Most of these graphs have been computed by McKay and/or Ted Spence. One of the most significant results by McKay and Spence is the classification of regular two-graphs on 36 and 38 vertices [94]. An immediate consequence of this was that all strongly regular graphs with parameters (35, 16, 6, 8), (36, 14, 4, 6), (36, 20, 10, 12) and their complements are known.

  A graph is *hypohamiltonian* if it is not Hamiltonian but each graph that can be formed from it by removing one vertex is Hamiltonian. The smallest is the Petersen graph. Table 3.3 summarises McKay's collection of hypohamiltonian graphs.

  All non-isomorphic connected planar graphs with up to 11 vertices are stored in McKay's collection of graphs. McKay also stores the planar embeddings of the graphs. McKay's repository stores plane 5-regular simple connected graphs up to 36 vertices and nonhamiltonian planar cubic graphs (this has graphs with no faces of size 3, cyclically 4-connected graphs, graphs with no faces of size 3 or 4 with cyclic connectivity exactly 4, and cyclically 5-connected graphs) and hypohamiltonian planar graphs (this includes cubic graphs of girth 4, cubic planar graphs of girth 5, and cubic planar graphs with an α*-edge*, where an α-edge in a graph is an edge which is present on every Hamiltonian cycle).

  Self-complementary graphs can have only orders congruent to 0 or 1 modulo 4. McKay stores all such graphs up to 17 vertices. However, he has a partial list of graphs for 20 vertices. McKay stores 8571844 self-complementary graphs (for 20 vertices) out of 9168331776 graphs.

  A connected graph is *highly irregular* if the neighbours of each vertex have distinct degrees. Such graphs exist for all orders except 3, 5 and 7. All highly irregular graphs with up to 15 vertices are listed in McKay's combinatorial data.

- **Ramsey graphs**:

  A *Ramsey*(s, t, n)*-graph* is a graph with n vertices, no clique of size s, and no independent set of size t. A *Ramsey*(s, t)*-graph* is a Ramsey(s, t, n)-graph for some

n. There are finite number of Ramsey(s,t)-graphs for each s and t [1], but finding all such graphs, or even determining the largest n for which they exist, is a difficult problem. McKay's repository stores a large number of Ramsey(s,t)-graphs for different combinations of s and t.

McKay's repository stores all Ramsey$(3,4)$-graphs, all Ramsey$(3,5)$-graphs, all Ramsey$(3,6)$-graphs, all Ramsey$(3,7)$-graphs[5], all Ramsey$(3,8)$-graphs[6], all maximal Ramsey$(3,9)$-graphs[7], all Ramsey$(4,4)$-graphs, and all maximal Ramsey$(4,5)$-graphs. In 1995, McKay and Radziszowski proved that there are no Ramsey$(4,5)$-graphs with more than 24 vertices and found 350 904 of them with 24 vertices. The remainder were found in 2016 by McKay and Angeltveit. There are 352 366 altogether.

A *Ramsey*$(4,4;3)$*-hypergraph* is a 3-uniform hypergraph (all hyperedges have size 3) with this property: every set of 4 vertices contains $1, 2$ or 3 edges. The smallest order for which no such hypergraph exists is called the *hypergraph Ramsey number* $\mathrm{R}(4,4;3)$. McKay computed all Ramsey$(4,4;3)$-hypergraphs up to 12 vertices.

- **Trees sorted by diameter**
  McKay computed all possible trees up to 22 vertices. McKay also computed and stored all homeomorphically irreducible trees[8] up to 30 vertices. They are also called series-reduced trees.

- **Digraphs**
  `digraph6` is a format used for storing directed graphs similar to the format used to store undirected graphs. McKay used it to store all the non-isomorphic tournaments up to 10 vertices. A tournament of odd order n is *regular* if the out-degree of each vertex is $(n-1)/2$. A tournament of even order n is *semi-regular* if the out-degree of each vertex is $n/2-1$ or $n/2$. McKay stores all regular and semi-regular tournaments of order up to 13.

  A regular tournament is *doubly-regular* if each pair of vertices is jointly connected to exactly $(n-3)/4$ others [123]. The order must be one less than a multiple of 4. These tournaments are equivalent to skew Hadamard matrices [123]. McKay computed and stored these graphs up to 51 vertices, however, the list is incomplete for graphs of order > 27. A tournament is *locally-transitive* if, for each vertex v, the in-neighbourhood and the out-neighbourhood of v are both transitive tournaments. McKay listed all the non-isomorphic locally-transitive tournaments up to 20 vertices.

  McKay also computed acyclic digraphs up to 8 points.

All these graphs are presented in the form of static HTML pages and some of them can be downloaded as plain text files. McKay's repository is recognized for its sheer

[5] Brinkmann et al. found 1 118 436 graphs from this list [23]
[6] Brinkmann and Goedgebeur found the full list in 2012 [23]
[7] The maximal Ramsey$(3,9)$-graph has 35 vertices and was found by Kalbfleisch in 1966 [79], but it took 47 years to prove its uniqueness [57]
[8] A tree in which all nodes have degree other than 2 is called a *homeomorphically irreducible tree*.

Graph type	Order	Exhaustive list (Y/N)
Simple connected unlabelled	1–10	Y
Eulerian, Connected eulerian, chordal	1–12, 1–12, 1–12	Y
Perfect	1–11	Y
Strongly regular	refer [91, 94]	Y
Hypohamiltonian	1–16	Y
Hypohamiltonian cubic	1–26	Y
Hypohamiltonian cubic, girth ≥ 5	1–28	Y
Hypohamiltonian cubic, girth ≥ 6	1–30	Y
Connected planar	1–11	Y
Plane 5-regular simple connected	1–36	Y
Self-complementary	1–17, 20	Y,N
Highly irregular	1–15	Y
Ramsey(3,4), Ramsey(3,5), Ramsey(3,6), Ramsey(3,7), Ramsey(3,8), Ramsey(4,4)	–	Y
Maximal Ramsey(3,9), maximal Ramsey(4,5)	–	Y
Tournament	1–10	Y
Regular tournament	1–13	Y
Semi-regular tournament	1–13	Y
Doubly-regular tournaments	1–51	N (The list is exhaustive with order up to 27)
Locally-transitive tournament	1–20	Y
Acyclic digraph	1–8	Y

Table 3.3: A summary of graphs from McKay's combinatorial data.

volume and variety. An outline of his repository is presented in Table 3.3. McKay's repository expanded over the last three decades based on his research requirements. McKay's repository is a very good resource of special classes of graphs. It does not record graph parameters. Searching graphs of a special class (e.g. find a hypohamiltonian graph) is straightforward in McKay's "combinatorial data", however it does not facilitate searching for graphs satisfying more than one property (e.g. find a graph which is hypohamiltonial and self-complementary).

Conder's combinatorial data (2002–present)

Marston Conder maintains a collection of combinatorial group data, graphs and graph embeddings. In this section we will only discuss the graph-related data.

Conder's collection of cubic graphs is arguably the largest collection of cubic graphs. Conder stores all cubic symmetric graphs up to isomorphism, on up to 10000 vertices. One interesting thing to note here is all cubic symmetric graphs except the Petersen graph and the Coxeter graph[9] have a Hamilton cycle. For each of these graphs, Conder computes their type (defined in [35]), size of automorphism group, girth, diameter and bipartiteness. Conder also computed all symmetric graphs up to isomorphism, of order 2 to 30, with some information about their automorphism groups. A regular graph is called *semi-symmetric* if it is edge-transitive but not vertex-transitive (and then the automorphism group has two orbits on arcs). Conder computed all cubic semi-symmetric graphs up to isomorphism, on up to 10000 vertices, listed by order, type (refer [35]), girth, and diameter.

Conder's contribution towards graph embeddings is also noteworthy. Conder gives a summary of the maximum orders of group actions on compact Riemann surfaces of genus up to 301 and on compact non-orientable Klein surfaces of genus up to 302. For definitions and details on these data, refer to [34].

Conder's repository focuses on graph embedding and symmetry data. Like McKay, Conder stored the graphs in static HTML pages. Conder's repository is a unique repository as it is one of the few repositories with a rich collection of geometric and combinatorial group-related graph data.

Royle's combinatorial catalogue (2007–present)

Gordon Royle's catalogue has a broad variety of parameters. It contains interesting data about other combinatorial objects like those used in finite geometry and design theory. In this survey we will focus on the graph-related data of the repository. We will briefly describe the scope of the repository and various graph data stored in the repository.

- **Small graphs**
  Royle's catalogue gives the total number of some specific types of graphs. They are listed in Table 3.4.

[9] The *Coxeter graph* is a 3-regular graph with 28 vertices and 42 edges [151].
[10] with number of edges varying from 1 to 14.
[11] This includes graphs with partitions $(1, 14), (2, 13), \cdots, (7, 8)$.
[12] As per Vizing's theorem.

# of graphs of type	Up to order
Unlabelled simple	16
Unlabelled simple connected	16
Multigraphs[10]	15
Connected bipartite[11]	15
Trees	20
Class 1 and Class 2 [12]	9

Table 3.4: Royle's data on graph counting.

Royle also computed some interesting parameters for the graphs. This repository lists the chromatic numbers of all connected graphs on up to 11 vertices. A graph is said to be *vertex-critical* if its chromatic number drops whenever a vertex is deleted. Royle lists all such graphs up to 11 vertices. The graphs are stored in gzip-compressed files containing those graphs in `graph6` format. A graph is said to be *edge-critical* if its chromatic number drops whenever an edge is deleted. Every edge-critical graph is necessarily vertex-critical. Royle lists all such graphs up to 12 vertices, he also lists the number of all 4, 5, 6 and 7 edge-critical graphs up to 12 vertices, specifying the number of edges in each graph [127]. Each graph is stored in `graph6` format.

- **Cubic graphs**
  Exact numbers of cubic graphs are known by results of Robinson and Wormald for up to 40 vertices. The cubic graphs on up to 20 vertices, together with some smaller families of high girth cubic graphs on higher numbers of vertices, are available. The larger numbers in the table, other than the Robinson and Wormald results, are due to Gunnar Brinkmann. Each graph is stored in `graph6` format.

  One feature of Royle's repository is that he not only stores the graphs but also computes (stores) some interesting parameters of the graphs. In the case of cubic graphs, Royle has listed all graphs with up to 22 vertices with girth ranging from 2 to 14, he also listed all 3-connected cubic graphs (irrespective of girth) with up to 20 vertices.

  A *snark* is defined to be a cyclically 4-edge connected cubic graph with chromatic index 4 and girth at least 5 [136]. The really important part of this definition is that it can be used to show a *planar snark (a boojum)* would be a counterexample to the four-colour theorem [136]. Royle used Gunnar Brinkmann's cubic graph generation program to construct snarks of all orders up to 28, which are stored in the `graph6` format.

  We have discussed chromatic polynomials in Section 3.2.1. Royle used the tree representation to represent chromatic polynomials. Given a family of polynomials, it is natural to immediately think about computing their complex roots and to see whether any patterns arise that may lead to interesting theorems. When one does this with chromatic polynomials of graphs there are some quite striking patterns which are currently unexplained. A paper by Read and Royle [119] investigates some

of these patterns and have calculated the chromatic roots of many small graphs, and the structure of these roots is still elusive. One of the longstanding questions was whether it was possible for the chromatic polynomial of a graph to have roots whose real parts were negative [75, 139]. Royle discovered that the cubic graphs of large girth have some roots with quite significant negative real parts. Royle's repository stores the chromatic polynomial of all connected cubic graphs up to 30 vertices with girth ranging from 3 to 8.

- **Transitive graphs**
  Like McKay's repository, Royle's repository also contains transitive graphs. The data was prepared by McKay and Royle and thereby includes the various previous works of these authors in which the catalogue of transitive graphs was extended to 19 vertices [89], 24 vertices [126] and then up to 26 vertices [93]. The current extension has been made possible by the work of Alexander Hulpke [72] who has constructed all the transitive groups of degree up to 30. Using these groups Royle performed a complete re-computation of the graphs. The re-computation confirmed that the original numbers computed by McKay and Royle were correct.

  Royle's repository stores all transitive graphs up to 31 vertices. The correctness of the results has been checked up to 26 vertices. Royle also provided information on whether or not the following properties hold: Cayley, Non-Cayley, Connected Transitive, Connected Cayley, and Connected Non-Cayley hold on these graphs.

- **Cayley graphs**
  Let G be a group, and let $S \subseteq G$ be a set of group elements such that the identity element I is not in S. The *Cayley graph* associated with (G, S) is then defined as the directed graph having one vertex associated with each group element and directed edges (g, h) whenever gh^{-1} in S. The Cayley graph depends on the choice of a generating set, and is connected if and only if S generates G. Royle lists all the Cayley graphs on up to 31 vertices, but classified according to the group to which they belong.

  The first group of each order is the cyclic group, then the remaining groups are ordered according to the lists in the book Group Tables by Thomas and Wood [138]. They give some descriptive names, which Royle also uses (with the exception of using $D(2n)$ for the dihedral group of order $2n$, rather than $D(n)$). All graphs are stored in the `graph6` format.

- **Cubic transitive graphs**
  The combinatorial catalogue of cubic vertex-transitive graphs (prepared by McKay and Royle) contains graphs of order up to 256 vertices (inclusive). The graphs are stored in `graph6` format. Royle also specifies whether graphs are Cayley, non-Cayley and symmetric. However the list does not contain the information for some graphs with more than 96 vertices, if they are Cayley or non-Cayley.

- **Cubic cages and higher valency cages**
  A (k, g)-*cage* is a k-regular graph of girth g with the fewest possible number of vertices. Royle lists the currently known values for the sizes of a cubic cage. For certain small values of g the cages themselves are all known, and Royle lists them explicitly in the repository. For larger values of g Royle has given a range — the lower value is either the trivial bound $n(3, g)$ or a bound by extensive computation. Royle lists some cubic $(3, g)$ graphs for $g \leq 22$.

- **Cubic planar graphs**
  Royle lists 3-connected cubic planar graphs with up to 20 vertices. Royle constructed these by using Brinkmann and McKay's program *plantri* [24].

- **Cubic symmetric graphs (The Foster Census)**
  A graph is called *symmetric* if its automorphism group acts transitively on the set of arcs (directed edges) of the graph.

  If the graph is cubic, then by Tutte's theorem [144, 147] the automorphism group actually acts regularly on s-arcs for some value of s between 1 and 5, and we say that a graph is s-*arc transitive* if the group acts regularly on s-arcs but not transitively on $(s + 1)$-arcs. This repository lists the known cubic symmetric graphs with less than 1000 vertices. It is known to be complete for up to 768 vertices, but for 770 to 998 vertices it includes only the Cayley graphs.

- **Strongly regular graphs**
  Royle's repository stores strongly regular graphs up to 99 vertices and the graphs are stored in `graph6` format. Royle also computed some useful parameters for each of these graphs. He computed the eigenvalues of each of these graphs with their multiplicity.

Graph type	Order	Exhaustive list (Y/N)
Vertex-critical	1–11	Y
Edge-critical	1–12	Y
Class-2	1–9	Y
Cubic, diameter $\in [2, 14]$	1–22	N
3-connected cubic	10–20	Y
Snark	1–28	Y
Transitive	1–26 [13]	Y
Cubic transitive	1–256	Y
Cayley	1–31	Y
Cubic cages ((3,3)-cage, (3,4)-cage, (3,5)-cage, $\cdots$ (3,22)-cage)	–	N
3-connected cubic planar	10–20	Y
Cubic symmetric	1–1000	N [14]
Strongly regular graphs	1–99	Y

Table 3.5: A summary of graphs from Royle's repository.

Although Royle's repository contains fewer graphs than that of McKay's, it has a greater variety and contains information about many graph parameters. A quick summary on the graphs stored in Royle's repository is given in Tables 3.5. There is some duplication in Royle's repository, and just like McKay's repository, the data is represented in static form.

Other relevant repositories

A *torus* is an orientable surface that has one handle. A graph G is a *topological obstruction* for the torus if G has minimum degree at least three, and G does not embed on the torus but for all edges e in G, $G \setminus e$ embeds on the torus. A graph G is a *minor-order obstruction* for the torus if G is a topological obstruction for the torus and for all edges e in G, the graph resulting from contracting e embeds on the torus. Wendy Myrvold computed minor order obstructions for the torus with up to 26 vertices [102, 103]. The graphs are stored in terms of the upper triangular part of their adjacency matrix.

Edwin van Dam and Ted Spence worked on the classification of all regular graphs on at most 30 vertices that have four distinct eigenvalues [41]. They classified the graphs in two parts: graphs for which all four eigenvalues are integral, and the case when there are just two integral eigenvalues. Spence also listed all strongly regular graphs on at most 64 vertices. Spence contributed immensely to finding SRGs. Spence's findings with McKay are summarised in Section 3.2.2. Along with Coolsaet and Degraer, Spence has computed the (45,12,3,3) strongly regular graphs [38]. There are precisely 78 of these listed in the repository.

Markus Meginger focuses on regular graphs. Meginger lists simple connected k-regular graphs on n vertices and girth at least g with given parameters n, k, g [97]. Meginger lists these graphs by using a computer program GENREG. It does not only compute the number of regular graphs for the chosen parameters but even constructs the desired graphs. The large cases with $k = 3$ were solved by Gunnar Brinkmann, who implemented a very efficient algorithm for cubic graphs [22]. Meginger's collection of regular graphs are given in Table 3.6.

Brinkmann's repository contains numbers of connected regular graphs with given number of vertices (up to 26) and degree (up to degree 7, for graphs of order 17 and of degree 3 for the rest). Brinkmann's repository also contains connected regular graphs with girth at least 4 for graphs of order 26, connected regular graphs with girth at least 5 for graphs of order 32, connected regular graphs with girth at least 6 for graphs of order 34, connected regular graphs with girth at least 7 for graphs of order 32, connected regular graphs with girth at least 8 for graphs of order 40 and connected bipartite regular graphs up to 32 vertices. Brinkmann's repository also contains connected planar regular graphs, along with connected planar regular graphs with girth at least 4 up to 26 vertices (degree 3), and connected planar regular graphs with girth at least 5 of order 20–28 with degree 3.

[13] Royle has computed with up to 31 vertices, but correctness has been checked up to 26 vertices.
[14] The list is complete up to order 768, for order in the range 770 – 798 it includes only Cayley graphs.

Graph type	Order	Exhaustive list (Y/N)
Connected 3-regular	1–18	Y
Connected 4-regular	1–14	Y
Connected 5-regular	1–12	Y
Connected 6-regular	1–11	Y
Connected 7-regular	1–11	Y
Connected 3-regular with girth ≥ 4	1–20	Y
Connected 4-regular with girth ≥ 4	1–16	Y
Connected 5-regular with girth ≥ 4	1–16	Y
Connected 6-regular with girth ≥ 4	1–18	Y
Connected 7-regular with girth ≥ 4	1–8	Y
Connected 3-regular with girth ≥ 5	1–22	Y
Connected 4-regular with girth ≥ 5	1–23	Y
Connected 5-regular with girth ≥ 5	1–30	Y
Connected 3-regular with girth ≥ 6	1–24	Y
Connected 4-regular with girth ≥ 6	1–34	Y
Connected 3-regular with girth ≥ 7	1–32	Y
Connected 3-regular with girth ≥ 8	1–40	Y

Table 3.6: Meginger's repository.

Primož Potočnik provides censuses of cubic and 4-regular graphs having different degrees of symmetry. Potočnik's repository consists of the following graphs.

- **Census of rotary maps**: For definitions of chiral, orientable maps and non-orientable maps, refer to [19, 59]. Potočnik's census stores all non-orientable maps up to 1500 edges and all orientable maps (both chiral and regular) up to 3000 edges. The previously known census of Conder contained the maps up to 1000 edges.

- **Census of cubic vertex-transitive graphs**: A census (compiled by Pablo Spiga, Gabriel Verret and Primož Potočnik) of cubic vertex-transitive graphs on at most 1280 vertices is available at this link [`http://www.matapp.unimib.it/~spiga/census.html`].

- **Census of 2-regular arc-transitive digraphs**: Spiga, Verret and Potočnik have compiled a complete list of all connected arc-transitive digraphs on at most 1000 vertices [112]. As a byproduct, they have computed all connected 4-regular graphs with at most 1000 vertices that admit a half-arc-transitive action of a group of automorphisms. In particular, all 4-regular half-arc-transitive graphs are there.

- **Census of arc-transitive 4-regular graphs**: Based on some of their theoretical work, Spiga, Verret and Potočnik, were able to construct a complete list of all 4-regular arc-transitive graphs on at most 640 vertices [111, 113]. The MAGMA code which generates the sequence of these graphs can be found at `https://www.fmf.uni-lj.si/~potocnik/work_datoteke/Census4val-640.mgm`.

- **Census of 2-arc-transitive 4-regular graphs**: This repository contains a list of 2-arc-transitive 4-regular graphs on up to 2000 vertices (in MAGMA code). The

list is complete on up to 727 vertices, but misses some 7-arc-transitive graphs that admit no s-arc-transitive group for s less than 7 on more than 727 vertices, and also some 4-arc-transitive graphs that admit no s-arc-transitive group for s less than 4 on more than 1157 vertices [110].

G. Haggard [62, 63, 64] computed and stored chromatic polynomials and the Tutte polynomials for some special classes of graphs. We summarise the contents of Haggard's repository in Table 3.7.

Graph type	Chromatic polynomial	Tutte polynomial
9-cage-$k, k \in [1, 18]$	Y	N
Complete graphs of order k, $k \in [3, 20]$	Y	N
Complete graphs of order k, $k \in [6, 25]$	N	Y

Table 3.7: A summary of Haggard's repository.

Suppose G is a graph on n vertices with diameter d. For any vertex u and for any integer i with $0 \leq i \leq d$, let $G_i(u)$ denote the set of vertices at distance i from u. If $v \in G_i(u)$ and w is a neighbour of v, then w must be at distance $i-1$, i or $i+1$ from u. Let c_i, a_i and b_i denote the number of vertices whose distances from w are $i-1$, i and $i+1$ respectively. G is a *distance-regular graph* if and only if these parameters c_i, a_i, b_i depend only on the distance i, and not on the choice of u and v. R. Bailey, A. Jackson and C. Weir [6] developed a repository of distance-regular graphs. A summary of the repository is presented in Table 3.8. Bailey et al. also maintains an index for all the named graphs present in the repository.

Graph type	Exhaustive list (Y/N)
Special families graphs	N
Unlabelled graph with up to order 1416	N
Distance-regular graphs with degree k, $k \in [3, 13]$	Y
Distance-regular graphs with diameter k, $k \in [2, 10]$	Y
Special named graphs	N

Table 3.8: A summary of Bailey et al's repository.

All these repositories are rich in content and are useful resources in their own right, however they are not intended to be general graph repositories and do not cover many basic graph parameters. A summary of these repositories is given in Table 3.9. Moreover, these repositories are all static in nature, so they are not well suited to queries consisting of combinations of parameters.

[15] up to 1500 edges for non-orientable maps and up to 3000 edges for orientable maps
[16] Complete up to 727 vertices

Repository name	Graph type	Order	Exh. list (Y/N)
Myrvold	Minor order obstruction for torus	1–26	Y
Spence	Strongly regular	refer [38, 94]	N
Spence	Regular	1–30	N (refer to [41])
Brinkman	Connected regular, girth $\geq k$ ($k \in [4, 7]$)	1–32	N
Potočnik	Rotary maps	– [15]	Y
Potočnik et al.	2-regular arc-transitive digraph	1–1000	Y
Potočnik et al.	Arc-transitive 4-regular	1–640	Y
Potočnik et al.	2-arc-transitive 4-regular	1–2000	N [16]

Table 3.9: A summary of other useful repositories.

3.2.3 Interactive repositories

The resources discussed above are static in nature. Although these resources are very useful they are not flexible, as the user has to manually refine the data if the requirements of the user is different to the data presented in the repository.

Data stored in these repositories are mostly in the form of adjacency matrix, adjacency list or McKay's `graph6` and `sparse6` format [92]. To perform any tasks that involve these large data the user may need some built-in tools and computer(s). The advent of large scale computing clusters, GPUs, high performance computers enable storage of these graph-data in a more flexible and interactive way. In the following sections we list some relevant attempts at creating interactive graph repositories.

Wolfram Alpha

Wolfram Alpha (also styled WolframAlpha, and Wolfram—Alpha) is a computational knowledge engine or answer engine developed by Wolfram Alpha LLC, a subsidiary of Wolfram Research. It is an online service that answers factual queries directly by computing the answer from externally sourced "curated data" [74] rather than providing a list of documents or web pages that might contain the answer, as a search engine does.

Wolfram Alpha, which was released on May 18, 2009, is based on Wolfram's earlier flagship product Wolfram Mathematica, a computational platform or toolkit that encompasses computer algebra, symbolic and numerical computation, visualisation, and statistics capabilities. It contains definitions of more than 300 different types of undirected graphs, more than 20 different types of directed graphs and a good collection of problems related to graphs.

Although this is an informative repository, it is different from all other repositories we discuss. It has definitions and some theoretical information about graphs but it does not list or store graphs. It can be queried for gathering information about specific graphs, e.g. "cliques" or "Petersen graph". This repository does not filter graphs satisfying constraints like "listing all graphs with max degree k". The number of graphs listed in Wolfram Alpha is smaller than the number of graphs listed in "An Atlas of Graphs".

Encyclopedia of Graphs (2012–present)

Encyclopedia of Graphs (`http://atlas.gregas.eu/`) is an online encyclopedia of graph collections aiming to help researchers find and use data about various families of graphs. This repository was created by a Slovenian company, Abelium d.o.o.. This repository allows graphs to be stored in any of these formats: `graph6`, `sparse6`, adjacency matrix, adjacency list and edge list. Encyclopedia of Graphs lists cubic symmetric graphs, edge-transitive graphs, vertex-transitive graphs and other classes of graphs, some of them acquired from other graph repositories listed in Section 3.2.2. This repository contains the following graphs.

- **Symmetric cubic graphs (The Foster Census)**: This repository stores the 796 cubic connected symmetric graphs with up to 2048 vertices.
- **Edge-transitive 4-regular graphs**: The collection provides information about connected edge-transitive graphs of degree 4. The current edition has 793 graphs with up to 150 vertices, which turned out to be an incomplete list. The authors of this repository are aiming to expand the range up to 512 vertices.
- **Vertex-transitive graphs**: This page lists all the transitive graphs on up to 31 vertices. The data was prepared by McKay and Royle [91, 127]. The current extension has been made possible by the work of Alexander Hulpke who has constructed all the transitive groups of degree up to 31. The numbers in the current version are guaranteed to be correct only up to 26 vertices. The transitive groups on 24, 27, 28 and 30 vertices have not yet been checked. This list contains 100661 graphs.
- **Hexagonal capping of symmetric cubic graphs** : The *hexagonal capping* HC(G) of a graph G has four vertices $\{u_0, v_0\}, \{u_0, v_1\}, \{u_1, v_0\}, \{u_1, v_1\}$ for each edge $\{u, v\}$ of G, and each $\{u_i, v_j\}$ is joined to each $\{v_j, w_{1-i}\}$, where u and w are distinct neighbours of v in G [70]. The collection includes hexagonal cappings of cubic symmetric graphs up to 798 vertices. This list contains 284 graphs.
- **Line graphs of symmetric cubic graphs**: Given a graph G, its *line graph* $L(G)$ is a graph such that each vertex of $L(G)$ represents an edge of G; and two vertices of $L(G)$ are adjacent if and only if their corresponding edges share a common endpoint are incident in G. The collection includes line graphs of cubic symmetric graphs with up to 768 vertices and Cayley graphs in the range 770–998 vertices.
- **Arc-transitive 4-regular graphs**: This collection contains a complete census of all connected arc-transitive 4-regular graphs of order at most 640. The census is a joint project by Potočnik, Spiga, and Verret [113]. They stored 4820 graphs in this category.
- **Regular graphs**: This collection contains all connected regular graphs of girth 3 up to order 12, of girth 4 up to order 16, and of girth 5 up to order 23. The collection was prepared using the program `geng` which is part of the `nauty` software package of McKay. This list contains 140959 graphs.

- **Trees**: This collection contains all 522958 non-trivial trees with up to the order 19, as prepared by Royle [127].

- **Vertex-transitive cubic graphs**: This collection contains a complete census of all 111360 connected vertex-transitive cubic graphs of order at most 1280. The census is a joint project by Potočnik, Spiga, and Verret [113]. The properties of graphs were collected from the DiscreteZOO library [12].

- **Highly irregular graphs**: A connected graph is *highly irregular* if the neighbours of each vertex have distinct degrees. Such graphs exist on all orders except 3, 5 and 7. This collection lists all 21869 highly irregular graphs with up to the order 15 and was provided by McKay [91].

- **Snarks**: This lists all 153863 snarks with up to the order 30 and was provided by The House of Graphs [22].

- **Cubic graphs**: This collection contains a complete census of 556471 connected cubic graphs of order at most 20. The data was prepared by Royle [127] and later extended at the House of Graphs [22].

- **Strongly regular graphs**: We defined strongly regular graphs in the Section 3.2.1. A strongly regular graph is called *primitive* if both the graph and its complement are connected. The census lists 43679 primitive strongly regular graphs with up to order 40; however, the complete classification is still open for SRG(37,18,8,9). The census was provided by McKay and Spence [94].

- **Networks**: Network theory is the study of graphs as representations of either symmetric or asymmetric relations between discrete real-world objects. It studies technological networks (the internet, power grids, telephone networks, transportation networks), social networks (social graphs, affiliation networks), information networks (World Wide Web, citation graphs, patent networks), biological networks (biochemical networks, neural networks, food webs), and many more. Graphs provide a structural model that makes it possible to analyse and understand the ways in which many separate systems act together. This (expanding) collection houses some of the "interesting" networks that are used in research. This category contains 51 graphs of order ranging from 3 to 23219. Most of these graphs are of order ranging from 30 to 40.

- **Edge-critical graphs**: The collection was calculated by Royle [127] and lists 185844 edge-critical graphs with from 4 to 12 vertices.

- **Fullerenes**: A *fullerene* is a cubic planar graph having all faces 5- or 6-cycles [60]. Fullerenes are planar and hence polyhedral, and every fullerene has exactly twelve 5-cycles. They acquired this data from the House of Graphs [22] and listed all 467927 fullerenes on up to 90 vertices.

- **Maximal triangle-free graphs**: A *triangle-free* graph is an undirected connected graph in which no three vertices form a triangle. Triangle-free graphs may be equivalently defined as graphs with clique number ≤ 2, graphs with girth ≥ 4 (or 0), graphs with no induced 3-cycle, or locally independent graphs. In order to determine properties of all triangle-free graphs, it often suffices to investigate maximal triangle-free graphs. These are triangle-free graphs for which the insertion of any further edge would create a triangle. They acquired this from the House of Graphs [22] and listed 197396 maximal triangle-free graphs on up to 17 vertices.

- **Planar graphs**: A *planar graph* is a graph that can be embedded in the plane, i.e. it can be drawn on the plane in such a way that its edges intersect only at their endpoints. This collection was calculated by McKay [91] and he lists all 78633 non-isomorphic connected planar graphs on up to 9 vertices.

- **Vertex-critical graphs**: The collection was calculated by Royle [127] and lists 359787 vertex-critical graphs with from 4 to 11 vertices.

One can search the site by using graph names, Universal Graph Identifier (UGI), defined by the authors of the repository or by collection names. UGI is a string that uniquely identifies a graph and can be used to directly access its properties page. One can use filters to obtain specific graphs of interest, e.g. "bipartite = true, minimum degree > 3". After the relevant graphs and their properties have been selected, they can be exported. Alternatively, one can download the data of a specific graph. Its main focus is to list families of graphs. A summary of the graphs stored in this repository is listed in Table 3.10. Since most of its data has been collected from other sources, the correctness of the data is subject to the correctness of the various repositories used as sources.

House of graphs (2013–present)

A recent attempt to build an interactive repository "A House of Graphs" (`https://hog.grinvin.org/`) is by Brinkmann, Coolsaet, Goedgebeur and Mélot in 2013 [22]. A House of Graphs provides a searchable and downloadable graph database. Another functionality of the House of Graphs is a list of graphs which have been used as counterexamples. The authors of the database call these graphs "interesting/relevant" and they suggest that, for a new theorem or conjecture, the chance of finding a counterexample is higher among the "interesting" graphs. The authors acknowledge the difficulty in creating a database that contains all graphs up to a specific order, therefore they chose to create a database based on a paraphrase of Orwell's famous words: *all graphs are interesting, but some graphs are more interesting than others* [22].

Some graphs (e.g. the famous Petersen graph or the Heawood $(3, 6)$-cage on 14 vertices) or graph classes (e.g. snarks) appear repeatedly in the literature as counterexamples. In order to construct a rich source of possible counterexamples, they inserted 1570 graphs into the database at its inception. These graphs are either counterexamples to known

Graph type	Order	Exhaustive list (Y/N)
Connected Cubic	1–20	Y$^{*,\#}$
Symmetric cubic	1–2048	Y$^{*,\%}$
Transitive	1–31	Y$^{*,+}$
Vertex-transitive cubic	1–1280	Y$^{\&}$
Edge-transitive 4-regular	1–150	N
Arc-transitive 4-regular	1–640	Y$^{\&}$
Line graphs of symmetric cubic graphs	1–768	Y
Connected regular, girth 3	1–12	Y^{+}
Connected regular, girth 4	1–16	Y^{+}
Connected regular, girth 5	1–23	Y^{+}
Trees	1–19	Y*
Maximal triangle-free graphs	1–17	Y$^{\#}$
Planar	1–9	Y^{+}
Vertex-critical	4–11	Y*
Edge-critical	4–12	Y*
Strongly regular	1–40	Y^{+} [94]
Highly irregular graph	1–15	Y^{+}
Snark	1–30	Y$^{\#}$
Fullerenes	1–90	Y$^{\#}$
Network	–	N

Table 3.10: A summary of graphs from Encyclopedia of Graphs. +: data from McKay's repository; *: data from Royle's repository; %: data from Conder's repository; #: data from house of graphs; &: data from Potočnik's repository.

conjectures or extremal graphs. The authors explicitly consider extremal graphs "interesting". A large proportion of the 1570 graphs are extremal graphs found by GraPHedron [96].

The idea behind creating this repository is that if one wants to test a conjecture on a list of graphs, the ideal case would be if one could restrict the tests to graphs that are "interesting" or "relevant" for this conjecture [22]. Here, the meaning of "interesting" and "relevant" is vague, but this already shows how much the question of whether a graph is interesting or not depends on the question that one wants to study. As per the authors, if a specific graph has sufficient properties to get separated from the huge mass of other graphs, then the graph is interesting in some respect — and of course the database allows one to add that graph and also offers the possibility of saying for which invariants the graph is especially interesting. The House of Graphs offers (among others) the following lists:

- All graphs registered as interesting in the database. These interesting graphs plays a special role in this database. For graphs in this list, a lot of invariants (like the chromatic number, chromatic index, the clique number, the diameter, independence number, average degree, smallest eigenvalue, second largest eigenvalue, genus, etc.) and also embeddings (drawings) are precomputed and stored.
- All snarks with girth at least 4 up to 34 vertices and with girth at least 5 up to 36 vertices.
- All IPR-fullerenes[17] up to 160 vertices.
- Complete lists of regular graphs for various combinations of degree, vertex number and girth.
- Vertex-transitive graphs.
- Some classes of planar graphs.

Some of these lists are physically situated on the same server as the website itself, but others are just links to other people's websites, like those of McKay [91], Royle [127], Spence [26] and Meringer [97].

Although this is not a static repository of graphs and it provides the option of querying the database with a combination of multiple parameters, it does not list all graphs up to some order. Graphs like Petersen's graph are obvious inclusions for this repository but among special classes of graphs like trees, etc., very few will satisfy the "interesting" criterion. As a result one cannot get a comprehensive list of graphs up to some order as an outcome of queries. We list the graphs of this repository in Table 3.11 and the parameters in Table 3.12. The repository allows users to add graphs to the database that they themselves consider "interesting". This itself is an interesting way of building a searchable graph database.

[17] The *face-distance* between two pentagons is the distance between the corresponding vertices of degree 5 in the dual graph. We refer to the least face-distance between pentagons of a fullerene as the *pentagon separation* of the fullerene, denoted by d. Note that $d = 1$ gives the set of all fullerenes and $d = 2$ gives the set of all *IPR fullerenes*.

Graph type	Order	Exhaustive list (Y/N)
Snarks with girth ≥ 4	1–34	Y
Snarks with girth ≥ 5	1–36	Y
IPR-fullerenes	1–160	Y
Uniquely Hamilton with girth $\geq k$ ($k \in [3,5]$)	1–12	Y
Planar uniquely Hamilton with girth $\geq k$ ($k \in [3,5]$)	1–12	Y
Triangle-free k-chromatic ($k \in [4,5]$)	1–15, 1–23	Y
Cubic	1–22	N* [18]
Transitive	1–26	Y*
Cubic transitive	1–256	Y*
Connected Cubic	1–24	Y*,$
Connected regular (same as Table 3.6)	–	Y$,+
Strongly regular	–	N+,#

Table 3.11: A summary of graphs from House of Graphs. +: data from McKay's repository; *: data from Royle's repository; #: data from Spence's repository; $: data from Meginger's repository.

List of parameters
Algebraic connectivity, average degree, circumference
Chromatic number, chromatic index, clique number
Density, diameter, edge connectivity
Genus, girth, longest induced path
Longest induced cycle, matching number, minimum independent set
Number of components, number of triangles, radius
Second largest eigenvalue, smallest eigenvalue, vertex connectivity

Table 3.12: Parameters in House of Graphs.

Hoppe and Petrone's collection (2014)

In 2014, Hoppe and Petrone [71] exhaustively enumerated all simple, connected graphs of order ≤ 10 using `nauty` [90] and have computed the independence number, automorphism group size, chromatic number, girth, diameter and various properties like Hamiltonicity and Eulerianness over this set. Integer sequences were constructed from these invariants and checked against the Online Encyclopedia of Integer Sequences (OEIS). However they used brute force methods using Networkx [61], graph-tools [107] and PuLP [99] to compute the parameters. They presented the data in static form and stored it in a database. However this system is not interactive.

[18] All cubic graphs with diameter ranging from 2 to 14.

Discrete ZOO

Another notable repository is the Discrete ZOO (`https://discretezoo.xyz/`). This repository hosts 212269 graphs [12]. This repository mainly contains vertex transitive graphs (up to 31 vertices), cubic vertex transitive graphs (up to 1280 vertices) and cubic arc transitive graphs (up to 2048 vertices). One can filter its queries based on the graph parameters listed in Table 3.13.

Parameter	Type
Bipartite	Boolean
Cayley	Boolean
Clique number	Numeric
Degree	Numeric
Diameter	Numeric
Distance regular	Boolean
Distance transitive	Boolean
Edge transitive	Boolean
Girth	Numeric
Moebius ladder	Boolean
Strongly regular	Boolean
Triangles count	Numeric

Table 3.13: Parameter filter used in Discrete ZOO.

An online atlas of graphs (2010–present)

The repositories by McKay [91], Royle [127], Conder [36] are rich in content and are considered very good repositories. Any query on these data requires downloading the data first followed by manual compilation. For running queries consisting of multiple parameters on these data the process is complicated, which makes the usage of these repository little difficult. The House of Graphs on the other hand is extremely efficient for handling complex queries but the amount of data stored in this repository is sparse and incomplete. Although this is a good repository for some conjecture verification work, other problems may require more comprehensive data.

Another approach is to give comprehensive information on queries (conjectures), i.e. *if* the system can answer precisely whether the conjecture holds for all graphs of order $\leq k$, then this will at least become a lower bound on the order of graphs for which the conjecture holds, *else* the system will provide a counterexample for the conjecture. This prompted Paul Bonnington and Graham Farr to propose a repository of graphs which has the efficiency of handling complex queries (like House of Graphs) and completeness of data (like the repositories of McKay, Royle etc.).

In 2009 Nick Barnes [10] built a prototype of an Online Graph Atlas (OLGA) with the following invariants: degree sequences, connected components, girth, radius and diameter, independence number, clique number, vertex cover number, domination number, circumference, length of the longest path, size of the maximum matching. Barnes used

the Monash Grid cluster on a quad-core 2.5Ghz Intel Xeon L5420 processor, with 16GB of RAM, and a MySQL server hosted by Monash University Information Technology Services to develop the prototype. All of these invariants that Barnes used are easily computable or they follow a recursive rule, but there are some parameters, like the genus of a graph, which do not follow a recursive structure. C. Paul Bonnington and Graham Farr developed an approach to compute parameters using recursive lower and upper bounds. Man Son Sio (a BSE Honours student from Clayton School of IT, Monash University) extended Nick Barnes's system to include two new parameters: genus, as an example of a parameter that is hard and has no simple recursive rule; and the chromatic polynomial, as an example of a parameter that is a polynomial rather than a single number. Sio also improved the database query times, which were a problem in the first prototype, though there is scope of further improvement there [134].

The initial versions of OLGA used McKay's `nauty` [91] as the backbone to generate the graphs. The advantage of using `nauty` is the ease of use but it makes the system dependent on `nauty`. In 2015, the author (under the supervision of Farr, Bonnington and Morgan) changed the basic architecture of OLGA. This work uses the Schreier-Sims algorithm [133] for isomorphism checking and generated all graphs up to 12 vertices. We took advantage of parallel computing and cloud computing. We computed degree sequence, connected components, girth, radius, and diameter, independence number, clique number, vertex cover number, domination number, circumference, length of the longest path, size of the maximum matching, vertex connectivity, edge connectivity, chromatic number, chromatic index, treewidth, Tutte polynomial (with up to 7 vertices), automorphism group, genus, and eigenvalues. We also introduced a new graph parameter, the most frequent connected induced subgraph (MFCIS). The design of OLGA is scalable, however storage is a bottleneck for OLGA.

3.3 In a nutshell

In this section we summarise the contents of each repository in Table 3.14. In Figure 3.3, we illustrate the high level dependencies between the repositories using a simple directed labelled graph, with repositories as vertices and dependencies between repositories as edges. If repository "B" depends on repository "A" for some specific data "info", we depict this by a directed edge from "A" to "B" with the label "info" on it. If data is generated in collaboration between two repositories, we use a bidirection edge between them.

3.4 Conclusion

All these graph repositories are created to help researchers to get readily available graph data. If the need is to find a specific family of graphs like regular graphs, SRGs, Ramsey graphs etc., one can use the static repositories of McKay [91], Royle [127] and Conder [36]. To get an overall picture and comprehensive information about a graph-related problem on graphs with up to 11 vertices, one can use OLGA. To verify theorems and conjectures

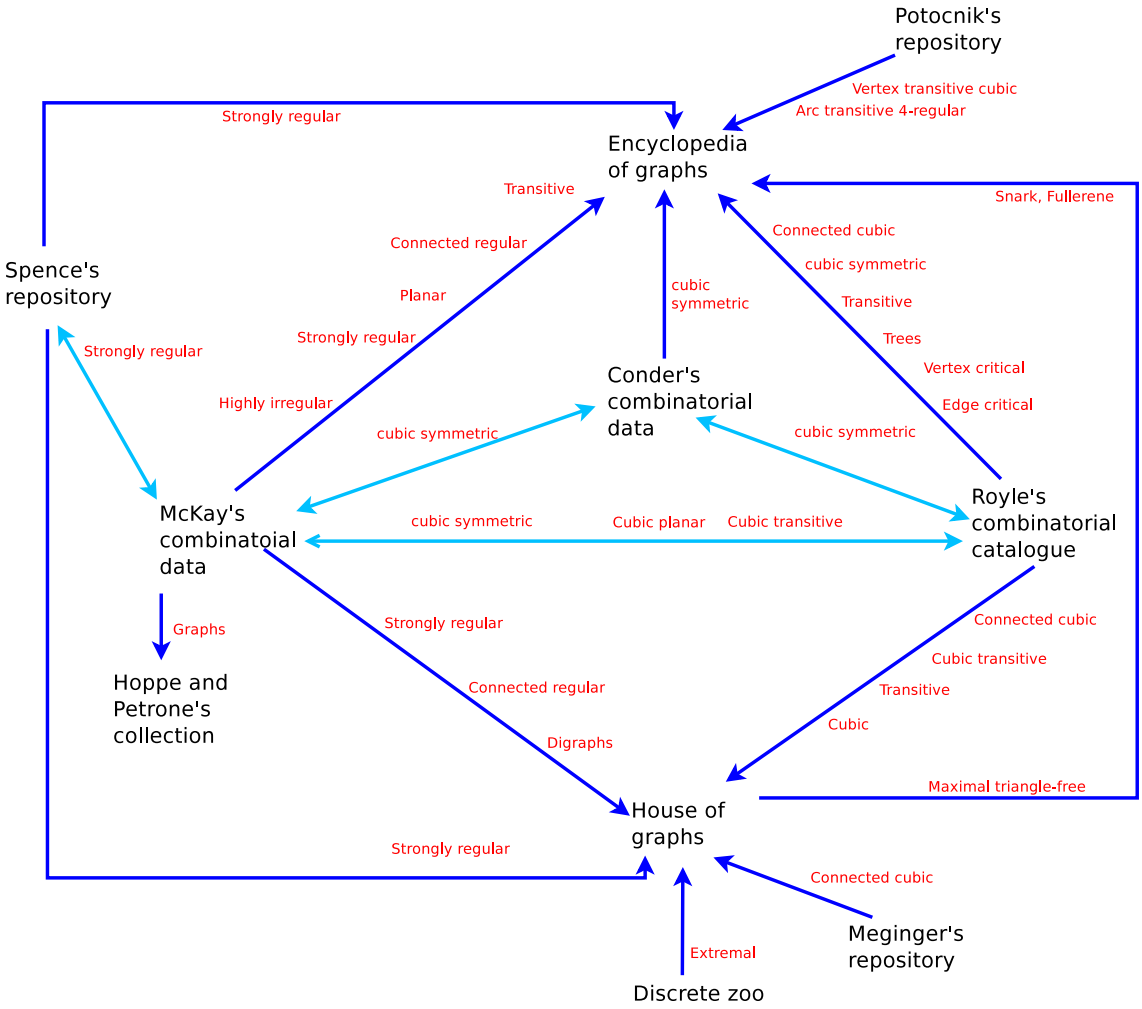


Figure 3.3: Data flow between graph repositories.

Graphs and parameters	Repositories
Unlabelled graphs with up to order 10	McKay, Royle
Strongly regular graphs	McKay, Spence, Royle, Encyclopedia of Graphs
Ramsey graphs	McKay
Trees	McKay, Royle
Digraphs	McKay, House of Graphs
Cubic graphs	Conder, Royle
Cubic symmetric graphs	Conder, Royle
Symmetric graphs	Conder
Snarks	Royle, House of Graphs, Encyclopedia of Graphs
Transitive graphs	Royle, Potočnik, Encyclopedia of Graphs
Cubic planar graphs	Royle, McKay
Fullerenes	House of Graphs, Encyclopedia of Graphs
Extremal graphs	House of Graphs, Discrete zoo
Edge-critical graphs	Royle, Encyclopedia of Graphs
Vertex-critical graphs	Royle, Encyclopedia of Graphs

Table 3.14: Summary of graph repositories.

on interesting graphs, A House of Graphs [22] can be used. None of these repositories guarantee a complete set of information about graphs or graph parameters, but they still offer useful information to understand many problems better and save a lot of time for researchers.

Chapter 4

Importance of OLGA

In Chapter 1 we introduced OLGA and briefly outlined its importance. The usage of OLGA is not limited to the role of a search engine for graphs. In this chapter, we demonstrate the utility of OLGA as a theorem/conjecture verifier (extender), generator and a data analytics tool. We also highlight an interesting complexity class inspired by the OLGA setup which is analogous to the notion of self-reducibility introduced by Schnorr [130]. We list some interesting questions that can be answered using OLGA (or an OLGA-like framework).

4.1 OLGA as a conjecture (dis)prover

The usefulness of OLGA lies in its completeness, that is, when OLGA answers a query it not only provides an answer but also gives the complete information on the query over the number of vertices specified in the query.

For example, suppose the query is to find *a* graph on 8 vertices which is 4-connected and 3-colourable and with domination number between 3 to 5. OLGA interprets the query as find *all* unlabelled graphs on 8 vertices that are 4-connected, 3-colourable and with domination number between 3 to 5. As a result the user gets the complete information on all graphs of order 8 satisfying the specified criteria.

Consider the following open question and conjectures:

Open question 4.1.1 (Harary-Schwenk, 1974 [67])**.** *Which graphs have distinct eigenvalues?*

Figure 4.1 gives some examples of graphs that have distinct eigenvalues and were found using OLGA. The graphs A, B, C have the following eigenvalues:

$$A : \{2.3027, 0.6180, 0, -1.6180, -1.3027\},$$
$$B : \{2.4811, -1.1700, 0.6888, 0, -2.0\},$$
$$C : \{-1.8136, 2.3429, 0.4706, 0, -1.0\}.$$

A complete list of graphs with up to 10 vertices satisfying the criterion of Open question 4.1.1 is given in Chapter 8. Although this data does not answer the question in its

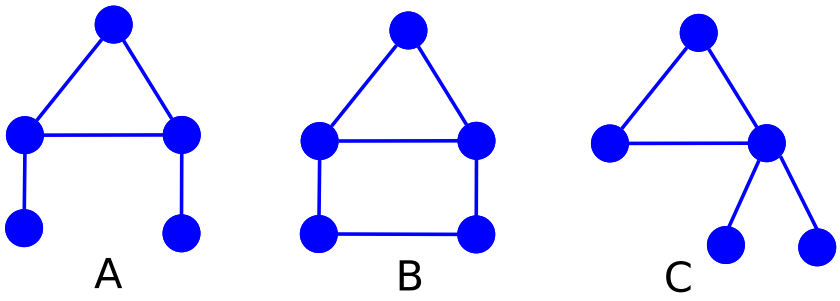


Figure 4.1: Some graphs with unique eigenvalues.

entirety, it answers the question for all graphs with fewer than 11 vertices. It also provides information that may be useful in finding a general solution.

Conjecture 4.1.2 (B. Reed, 1996 [121]). *Every 3-connected cubic graph G has domination number at most $\lceil |V(G)|/3 \rceil$.*

We used OLGA to compute $\gamma(G)$ of all the graphs with up to 10 vertices. Based on our computation we have Theorem 4.1.3.

Theorem 4.1.3. *Conjecture 4.1.2 holds for all graphs with up to 10 vertices.* □

However Kostochka and Stodolsky (2005) disproved this conjecture by giving a counterexample, a connected cubic graph on 60 vertices with domination number 21 [82]. Although OLGA did not disprove the conjecture in this case, it provided a lower bound on the number of vertices for which Conjecture 4.1.2 holds.

Conjecture 4.1.4 (B. Reed, 1998 [122]). *For every graph G, $\chi(G) \leq \lceil \frac{1}{2}(\Delta(G)+1) + \frac{1}{2}\omega(G) \rceil$.*

Rabern (2008) proved that Conjecture 4.1.4 holds for graphs satisfying $\Delta(G) \geq n + 2 - (\alpha(G) + \sqrt{n+5-\alpha(G)})$ [114]. We used OLGA to compute $\Delta(G)$, $\omega(G)$ and $\chi(G)$ for all graphs with up to 10 vertices. Based on our computation we have:

Theorem 4.1.5. *Conjecture 4.1.4 hold for all graphs with up to 8 vertices.* □

Open question 4.1.1 and Conjectures 4.1.2 and 4.1.4 are cases where OLGA can be used in a direct manner. However we may gain more information from OLGA by modifying some portion of its code. We demonstrate an indirect use of OLGA in Theorem 4.1.7.

Conjecture 4.1.6 (C. Thomassen, 1976 [17]). *Every longest cycle in a 3-connected graph has a chord.*

Thomassen (2018) proved that every longest cycle in a 2-connected cubic graph contains a chord [140]. We used OLGA to list all 3-connected graphs with up to 10 vertices. There are 3968929 such graphs. For each of these graphs, we listed the longest cycles by manipulating the circumference function in OLGA's code so that it also searched for a chord in each of the listed cycles. Based on our computation, we have:

Theorem 4.1.7. *For all 3-connected graphs with up to* 10 *vertices, Conjecture 4.1.6 holds.* □

Conjecture 4.1.8 (S. Smith, 1984 [17])**.** *In a k-connected graph, where $k \geq 2$, any two longest cycles have at least k vertices in common.*

Using an approach similar to the one we used earlier, the validity of Conjecture 4.1.8 can be checked for graphs with up to 10 vertices. In the literature there are instances where graph repositories played an important role either to improve on known results or disprove conjectures. We listed such instances in Section 3.2.2.

4.2 OLGA as a conjecture (theorem) generator

In Section 4.1 we highlighted the importance of OLGA in addressing existing open questions and conjectures. In this section we outline the importance of OLGA in finding new information on parameters. The current version of OLGA computes 25 different parameters for all unlabelled graphs with up to 10 vertices. One can do the following with the data present in OLGA:

- Find a relationship (if one exists) between every pair of parameters. Some results on these types of relations are already known. In these cases they act as a verifier for the correctness of OLGA. This kind of correctness checking was employed by Barnes [10] in the earlier versions of OLGA. However, there is still scope to find more of these relations.
- In Chapter 8 we list some graphs that have some unique properties in the context of graph colouring and MFCIS. We also present some conjectures about them in Chapter 8.
- As mentioned in Chapter 6, we computed most of the parameters present in OLGA using recursion. We list some new interesting results on these parameters with respect to their respective recurrence relations in Chapter 6.
- We report some new integer sequences in Chapter 8 based on the properties that graphs satisfy.

4.3 OLGA as an assistant for inductive proofs

A tool like OLGA may be helpful in inductive proofs where the base case consists of many small graphs and the inductive step is simple.

A *spindle surface* is formed from the sphere by identifying two distinct points, commonly considered as the north and south poles N and S. A graph G is *nearly planar* if there exists an edge e such that G is nonplanar and $G-e$ is planar. Archdeacon and Bonnington (2004) found the complete list of 21 cubic graphs that are topological obstructions for cubic graphs that embed on the spindle surface [2]. They also gave the topological obstruction set for cubic nearly planar graphs. In their proof of [2, Proposition 4.5], the base case consisted of many small graphs and the inductive step is simpler.

4.4 Scope of OLGA in machine learning

In this section we propose how machine learning concepts may be used to predict the parameter value of a graph that is not present in OLGA.

Machine learning is an important branch of computer science and specifically of artificial intelligence. In the OLGA setup, we use the graphs or a combination of graphs and parameters as *input variables* while a parameter value p (not part of the input variables) is an *output variable*. The input variables are typically denoted using the symbol X, with a subscript to distinguish them. The input variables go by different names, such as *predictors, independent variables, features* or sometimes just *variables*. The output variable is often called the *response* or *dependent variable*, and is typically denoted by the symbol Y.

In *supervised learning* [15], we assume there is some relationship between Y and X which can be represented as $Y = f(X) + \epsilon$. Here f is some fixed but unknown function of X and ϵ is a random *error term*, which is independent of X and has mean zero. In this formulation, f represents the *systematic* information that X provides about Y.

In the OLGA setup we define a *data set* $D = (D_{\text{train}}, D_{\text{test}})$, where $D_{\text{train}} = \{(G, p(G)) : G \text{ is in OLGA}\}$ and D_{test} is a finite set of graphs not in OLGA for which we want to compute parameter p. Figure 4.2 illustrates how a supervised learning model M can be trained with data generated from OLGA to compute the approximate value of a parameter p of a graph which is not present in OLGA.

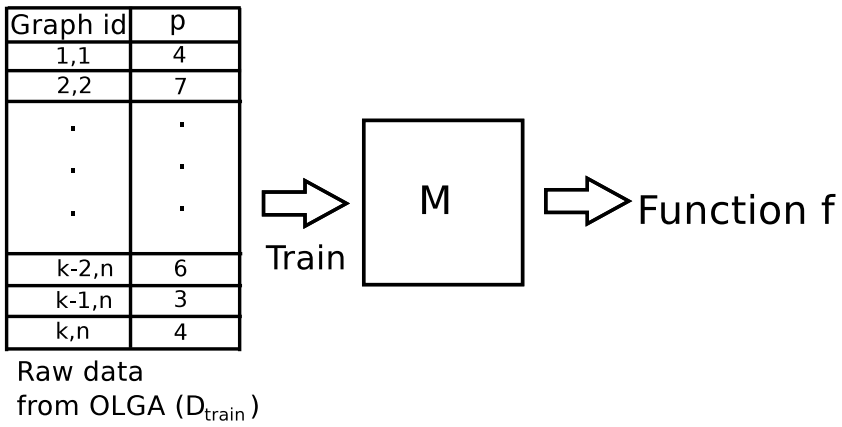


Figure 4.2: Supervised learning on OLGA data.

4.5 Self-reducibility

Schnorr introduced self-reducibility [130]. He tried to correlate the decision and function versions of NP-complete problems using self-reducibility. In this section we define different variations of self-reducibility. We also relate self-reducibility to the computation method adopted by OLGA for parameter computation.

Definition 4.5.1. *The language accepted by an oracle Turing machine M with oracle A is denoted by $L(M, A)$.*

Definition 4.5.2. *The function computed by an oracle Turing machine M with oracle f is denoted by $F(M, f)$.*

Definition 4.5.3. *[130] A set A is ldq-self-reducible if there is a polynomial time deterministic oracle Turing machine such that $A = L(M, A)$, and on each input of length n every word queried to the oracle has length less than n.*

The prefix "ldq" in ldq-self-reducible stands for length decreasing query.

Definition 4.5.4. *[130] A set A is self-reducible if there is a strict partial order $<$ on Σ^* and a polynomial time deterministic oracle Turing machine M such that $A = L(M, A)$, satisfying:*

1. *On each input x every word queried to the oracle is $< x$.*
2. *There exists a polynomial p such that if $x < y$ then $|x| \leq p(|y|)$.*
3. *It is decidable in polynomial time whether $x < y$.*
4. *For every decreasing chain there exists a polynomial q such that the length of the chain is $\leq q(|e|)$, where e is the maximum element of the chain.*

Definition 4.5.5. *[130] A function f is self-reducible if there is a strict partial order $<$ on Σ^* and a polynomial time deterministic oracle Turing machine M such that $f = F(M, f)$, satisfying:*

1. *On each input x every word queried to the oracle to compute $f(x)$ is $< x$.*
2. *There exists a polynomial p such that if $x < y$ then $|x| \leq p(|y|)$.*
3. *It is decidable in polynomial time whether $x < y$.*
4. *For every decreasing chain there exists a polynomial q such that the length of the chain is $\leq q(|e|)$, where e is the maximum element of the chain.*

Theorem 4.5.6. *[9] Every NP-complete language L is self-reducible.*

In Definition 4.5.7, we define a partial order on graphs, which we use in the definition of OLGA-t-self-reducibility. Then we will show directly that if a parameter p is OLGA-t-self-reducible then p is self-reducible.

Definition 4.5.7. *The order relations $\prec_1, \prec_2$ and $\prec_3$ on the set of graphs are defined by:*

- *$H \prec_1 G$ if $|V(H)| < |V(G)|$.*
- *$H \prec_2 G$ if either $H \prec_1 G$ or $|V(H)| = |V(G)|$ and $|E(H)| < |E(G)|$.*
- *$H \prec_3 G$ if either $H \prec_1 G$ or $|V(H)| = |V(G)|$ and $|E(H)| > |E(G)|$.*

Note: $(\{graphs\}, \prec_t)$ is a strict partial order, for each $t \in \{1, 2, 3\}$. We define self-reducibility in the OLGA setup based on these order relations between graphs.

Definition 4.5.8. *A graph parameter p is OLGA-t-self-reducible for $t \in \{1, 2, 3\}$ if there is a polynomial time oracle Turing machine M such that $p = F(M, p)$, where M computes p for input graph G by querying the oracle and the queries to the oracle satisfy the following:*

1. *On each input G every word queried to oracle is $\prec_t G$.*
2. *There exists a polynomial q such that if $G \prec_t H$ then $|G| \leq q(|H|)$.*
3. *For every decreasing chain there exists a polynomial q such that the length of the chain is $\leq q(|B|)$, where B is the maximum element of chain.*

Note: If a parameter p is OLGA-t-self-reducible for $t \in \{2, 3\}$ then p is OLGA-1-self-reducible. For if a parameter p is OLGA-t-self-reducible for $t \in \{2, 3\}$ then each input word is $\prec_t$ the queried word, therefore is $\prec_1$ the queried word.

An example of a OLGA-1-self-reducible parameter is the independence number. OLGA uses the following recurrence relation to compute the independence number of a graph:

$$\alpha(G) = \max\{\alpha(G \setminus v), \alpha(G \setminus v \setminus N[v]) + 1 : v \in V(G)\}.$$

Here computing $\alpha(G)$ depends on knowing the independence numbers of the graphs $G \setminus v$ and $G \setminus v \setminus N[v]$ which have fewer vertices than G. We give a detailed discussion on computing the independence number in Section 6.3.3.

An example of a OLGA-2-self-reducible parameter is the circumference. OLGA used the following recurrence relation to compute the circumference of a graph:

$$\text{circ}(G) = \max\{\text{circ}(G \setminus e) : e \in E(G) \wedge \text{circ}(G) < \infty\}.$$

Finding the circ(G) depends on the circumference of the graphs $G \setminus e$ which have the same number of vertices as G and fewer edges than G. We give a detailed discussion on computing the circumference in Section 6.1.3.

An example of a OLGA-3-self-reducible parameter is the chromatic number. We have the following recurrence relation for the chromatic number of a graph:

$$\chi(G) = \min\{\chi(G/uv), \chi(G + uv), u, v \in V(G).$$

Finding the $\chi(G)$ depends on the chromatic numbers of the graphs G/uv and $G + uv$. In this case $G \prec_1 G/uv$ and $G \prec_3 G + uv$. We give a detailed discussion on computing the chromatic number in Section 6.4.2.

In Table 4.1 we list parameters and its OLGA-self-reducible type, based on the method used to compute the parameters.

Lemma 4.5.9. *If parameter p is OLGA-t-self reducible for $t \in \{1, 2, 3\}$ and C is a decreasing chain for $\prec_t$ with the maximum element B of order n, then the length of C is at most n^3.*

Parameters	OLGA-self-reducible type
Independence number, clique number, vertex cover number, treewidth, degeneracy	OLGA-1-self-reducible
Circumference, maximum matching number, domination number, chromatic index	OLGA-2-self-reducible
Chromatic number	OLGA-3-self-reducible

Table 4.1: Parameters and OLGA-self-reducibility.

Proof. We prove it by induction on the order of maximum element in the chain.
Base case:
If $n = |B| = 1$, then C has only one element. Therefore $|C| = n^3$.
Inductive step:
Suppose the lemma holds when $n = k-1$ for $k > 1$, that is, $|C| \leq (k-1)^3$ for all descending chains C with maximum element of order $k-1$.

Consider a decreasing chain C' for $\prec_t$ such that the maximum element B' has order k. Consider the following cases:

- C' is a chain for $\prec_1$: The length of C' is the number of elements in C' without B' plus 1 for the element B'. Since the order relation is $\prec_1$ and by the inductive hypothesis every decreasing chain for $\prec_1$ with maximum element of order $k-1$ has length at most $(k-1)^3$, therefore the length of C' is at most $(k-1)^3 + 1 < k^3$, as $k > 1$ as per assumption.

- C' is a chain for $\prec_2$: The length of C' is the number of elements in C' without the elements of order k plus the elements of order k. The order relation is $\prec_2$, therefore by the inductive hypothesis every decreasing chain for $\prec_2$ with the maximum element of order $k-1$ has length at most $(k-1)^3$. Since B' is of order k, the number elements of order k that are $\prec_2 B'$ in C' is at most $\binom{k}{2}$. Therefore, the length of C' is $\leq (k-1)^3 + \binom{k}{2} = k^3 - 5k/2(k-1) - 1 < k^3$.

- C' is a chain for $\prec_3$: The length of C' is the number of elements in C' without the elements of order k plus the elements of order k. The order relation is $\prec_3$, therefore by the inductive hypothesis every decreasing chain for $\prec_3$ with the maximum element of order $k-1$ has length at most $(k-1)^3$. Since B' is of order k, so the number elements of order k that are $\prec_3 B'$ in C' is at most $\binom{k}{2}$. Therefore, the length of C' is at most $(k-1)^3 + \binom{k}{2} + 1 = k^3 - 5k/2(k-1) < k^3$.

The lemma therefore holds, by the principle of mathematical induction. □

Theorem 4.5.10. *If a parameter p is OLGA-t-self-reducible for $t \in \{1, 2, 3\}$ then p is self-reducible.*

Proof. Let $t \in \{1, 2, 3\}$. $(\{graphs\}, \prec_t)$ is a strict partial order. Let us consider the following scenarios:

- p is OLGA-1-self-reducible: Let G be an input graph and H be a queried graph corresponding to G. By Definition 4.5.8, $H \prec_1 G$. Therefore condition 1 of Definition 4.5.5 is satisfied. By Definition 4.5.7 if $H \prec_1 G$, then $|H| < |G|$, this satisfies the second condition of self-reducibility. It takes linear time to decide $H \prec_1 G$, which satisfies condition 3 of Definition 4.5.5. By Definition 4.5.8, for every decreasing chain C, there exists a polynomial q such that $|C| \leq q(|B|)$, where B is the maximum element of C. Lemma 4.5.9 shows any decreasing chain under $\prec_1$ has length at most n^3, where n is the order of the maximum element. Therefore, the fourth condition of Definition 4.5.5 is satisfied.

  Therefore, all four conditions of Definition 4.5.5 are satisfied, implying that p is self-reducible.

- p is OLGA-2-self-reducible: Let the input graph be G and H be a queried graph. By Definition 4.5.8, $H \prec_2 G$. Therefore condition 1 of Definition 4.5.5 is satisfied. By Definition 4.5.7 if $H \prec_2 G$, then number of edges in H is less than the number of edges in G, this satisfies the second condition of self-reducibility. It takes linear time to decide if $H \prec_2 G$, as the process only involves a comparison of the number of vertices and edges between G and H. Therefore condition 3 of Definition 4.5.5 is satisfied. Lemma 4.5.9 shows any decreasing chain under $\prec_2$ has length at most n^3, where n is the order of the maximum element in the chain. Therefore, the fourth condition of Definition 4.5.5 is satisfied.

  Therefore, all four conditions of Definition 4.5.5 are satisfied, implying that p is self-reducible.

- p is OLGA-3-self-reducible: Let the input graph be G and H be a queried graph. By Definition 4.5.8, $H \prec_3 G$, satisfying the first condition of self-reducibility. Deciding $G \prec_3 H$ takes linear time, as the process only involves a comparison of the number of vertices and edges between G and H. Therefore, condition 3 of Definition 4.5.5 is satisfied. Lemma 4.5.9 shows any decreasing chain under $\prec_3$ has length at most n^3, where n is the order of the maximum element of the chain. Therefore, the fourth condition of Definition 4.5.5 is satisfied. Since $H \prec_3 G$, one of the following holds.
  – $|V(H)| < |V(G)|$: Condition 2 of self-reducibility Definition 4.5.5 holds.
  – $|V(H)| = |V(G)|$ and $|E(G)| < |E(H)|$: The maximum number of edges possible in a graph of order $|V(G)|$ is $\binom{|V(G)|}{2}$. Since both G and H have same order, $|E(H)| \leq \binom{|V(G)|}{2}$. Therefore $|H| \leq |V(G)|^2|G| = |V(G)|^3$. Hence the condition 2 of self-reducibility holds.
  Therefore p is self-reducible.

□

Chapter 5

Design and implementation

In this chapter we outline the design and system specifications used in the online graph atlas. We present the challenges faced during OLGA development and our approaches to work around it. We also specify various implementations used in OLGA.

5.1 Challenges in OLGA development

Graphs form an integral part of any graph repository. The usefulness of a repository enhances if information contained in it is correct and complete. The major challenges faced in OLGA development are the following:

- Graph generation,
- Isomorphism checking,
- Implementation priorities,
- Efficient storage,
- Query optimization.

5.1.1 Graph generation

In a *labelled* graph of order n, the integers from 1 to n are assigned to its vertices. The most commonly used methods of generating unlabelled graphs is to generate all labelled graphs and filter them up to isomorphism. Two natural questions arise in unlabelled graph generation. First, one asks: how many labelled graphs of order n are there? The second is: how many unlabelled graphs of order n are there? The first question is easier among the two to answer.

Pólya [66] found a generating function which counts the total number of labelled graphs over n vertices and m edges. Subsequently Palmer, Read [66] found generating functions to count the number of ways of labelling a graph of order n and the number of ways to count the number of connected unlabelled graphs. Details about these generating functions is given in [66].

Graph generation in OLGA

The first step for creating OLGA is to generate an exhaustive list of graphs of "small" order. The rate of growth of number of graphs is exponential with respect to the order of graph. This makes the task of graph generation extremely challenging. As we have shown in Table 1.1 number of unlabelled graphs on 12 vertices is already huge. So generating all graphs beyond 12 vertices will need more sophisticated computation methods. This is also an indication that it is highly unlikely that any repository of graphs will ever be able to store all graphs beyond some order.

The simple solution to this problem of graph generation is to generate graphs as required but this approach is not practical as the number of graphs on n vertices for $n \geq 11$ is so large that it will take a long time to generate the graphs. However there are some efficient graph generation softwares, one such software is McKay's `nauty` [90]. However the problem of using these software is the unwanted dependency on the software and moreover the user is restricted by the limitations of the software.

We used a simple brute-force algorithm to generate all connected graphs of order n, unique up to isomorphism. The algorithm is outlined in Algorithm 1.

Algorithm 1: Generate all unlabelled graphs of order n.

```
/* ************************************************
Input:  A positiver integer n
Output:  All unlabelled graphs of order n
Notation:  The vertices of a graph are numbered from 1,··· ,n.
   ************************************************          */
1  def generateGraphs(n):
2  temp=[ ]
3  final=[ ]
4  F = K_1
5  for i = 1 to n − 1 do
6  |  for each graph H of order i in F do
7  |  |  G = H ∪ (i + 1)
8  |  |  temp.append(all graphs resulted from all unique ways to join i + 1 with
   |  |   vertices of H ) check_Isomorphism(temp)      // This function keeps
   |  |   one graph per isomorphism class of graphs and removes the
   |  |   rest.
9  |  |  F.append(temp)
10 |  end
11 |  final.append(F)
12 end
13 return final
```

We parallelise Algorithm 1 by allocating a thread to check isomorphism of graphs with equal numbers of edges. We check the correctness of our algorithm by verifying the number of unlabelled graphs of order n generated by Algorithm 1 with the total number of unlabelled graphs given by the generating functions by Read and Palmer [66].

In terms of computation time McKay's `nauty` [90] takes less time to generate all unlabelled graphs of order n than Algorithm 1. We initially used Algorithm 1 in OLGA, as

the *check_Isomorphism*() is an integral part of OLGA and it can be parallelised further in future by considering more invariants before using the Schreier-Sims algorithm [133] discussed in Section 5.2. However, the current version of OLGA uses `nauty` only for graph generation and it uses *check_Isomorphism*() of Algorithm 1 for isomorphism checking.

5.2 Isomorphism checking

The graph isomorphism problem (GI) is in NP, although it is not known thus far whether it is in co-NP or NP-complete. Complexity theory results show if GI were NP-complete, then the polynomial time hierarchy would collapse to its second level ($\Sigma_p^2 = \Pi_p^2 = \text{AM}$) [20, 135]. Karp [80] in his seminal paper conjectured that GI might lie strictly between P and NP-complete. In 1977, Reed and Corneil wrote an interesting survey on the GI problem, called "The Graph Isomorphism Disease" [118] and new software still appears regularly to address the issue.

It is clear that the total number of connected unlabelled graphs of order n is much less than the number of connected labelled graphs of order n. We generate all connected labelled graphs using the methods discussed in Section 5.1.1. Since parameters stored in OLGA are invariant under isomorphism, it is sufficient to store the parameters only for the unlabelled graphs (rather than for all labelled graphs). Therefore, isomorphism checking plays an important role in the development of OLGA. After generating all connected labelled graphs, we choose one graph of each isomorphism class and refer it as the representative of the class. A *canonical label*, for a family $\mathcal{F}$ of isomorphic graphs is a function such that for any $G_1, G_2 \in \mathcal{F}$, canonical label(G_1) = canonical label(G_2). Since identity of graphs is easier to check than isomorphism of graphs, canonical labelling is our preferred mechanism to store the representative graphs of isomorphism classes.

The most commonly used software for GI is `nauty` [90], `VF2` [39], `saucy` [40], `bliss` [77], `trace` [108] and `canauto` [86]. The mechanisms used in these programs are very similar and all of them uses canonical labelling. Most of these programs started with the reimplementation of `nauty` and diverged afterwards. In OLGA we developed our own software to check graph isomorphism based on the canonical labelling using the Schreier-Sims algorithm [132, 133]. We implemented the algorithm from [83] and parallelised the process of partition refinements. The canonical labelling algorithm takes $2^{O(\sqrt{n \log n})}$ time. Our python implementation for computing the canonical label of a graph from [83] is listed here. We use the same variable names as in [83] for consistency.

```
class Canon1:
    """
    Implementation of algorithm 7.7 of Kreher, Stinson.
    """
    @classmethod
    def do(cls, graph: UGraph, blocks: Blocks, best: Vertices = None,
logger: logging.Logger = None, depth: int=0) -> Vertices:

        q: Blocks = Refine.do(graph, blocks)
```

```
        logger.debug('{} q: {}'.format('  '*depth, q))
        result: Result = Result.BETTER
        if best is not None:
            pi: Vertices = cls.get_partition(blocks)
            result = Compare.do(graph, best, pi)
            logger.debug('{} result: {}'.format('  '*depth, result))

        if len(q) == graph.num_vertices:
            logger.debug('{} discrete q: {}'.format('  '*depth, q))
            if (best is None) or (result == Result.BETTER):
                best = cls.get_partition(q)
                logger.debug('{} best: {}'.format('  '*depth, best))
            return best

        if result == Result.WORSE:
            return best

        return cls.handle_non_discrete(graph, q, best, logger, depth)

    @classmethod
    def handle_non_discrete(cls, graph: UGraph, q: Blocks,
    best: Vertices, logger: logging.Logger, depth: int) -> Vertices:

        ix_non_discrete = cls.get_ix_non_discrete(q)
        block_non_discrete: VertexSet = q[ix_non_discrete].copy()
        logger.debug
        ('{} non-discrete: {}'.format('  '*depth, block_non_discrete))
        r: Blocks = [block.copy() for block in q[:ix_non_discrete]]
        r.extend([{}, {}])
        r.extend([block.copy() for block in q[ix_non_discrete+1:]])
        for u in block_non_discrete:
            r[ix_non_discrete] = {u}
            r[ix_non_discrete+1] = block_non_discrete.difference({u})
            logger.debug('{} r: {}'.format('  '*depth, r))
            best = cls.do(graph, r, best, logger, depth+1)
        return best

    @classmethod
    def get_partition(cls, blocks: Blocks) -> Vertices:

        partition: Vertices = []
        for block in blocks:
            if len(block) > 1:
                break
            partition.append(list(block)[0])
        return partition

    @classmethod
    def get_ix_non_discrete(cls, blocks: Blocks) -> int:

        for ix_block, block in enumerate(blocks):
            if len(block) > 1:
                return ix_block
        return len(blocks)
```

5.3 Implementation priorities

In OLGA, we only consider computable parameters. Some graph parameters need super-exponential time to compute even when using the state of the art exact algorithms. Therefore, it becomes difficult to use them when the size of the input increases. This prompted us to explore the recursiveness of graph parameters and devise new algorithms based on it. In OLGA, we store the graphs and graph parameters progressively in order of number of vertices. This enables us to use this information and compute graph parameters recursively, which are otherwise extremely hard to compute. Recursive algorithms used in OLGA mainly use the following operations:

- Vertex deletion,
- Edge deletion,
- Edge contraction,
- Vertex identification.

Vertex deletion

We first compute parameter values for graphs of order ≤ 3 and store them. We then use these values to compute the parameter values of graphs of order > 3 recursively via vertex deletion. The vertex deletion process is parallelised by using different threads for each vertex to be deleted in the graph. Each thread then fetches the value of the parameter of the graph resulting from vertex deletion from the already stored parameter value in the OLGA. The threads communicate with each other to compute the minimum or maximum value fetched by the threads, as specified by the recurrence.

Consider the following example. Let P be the parameter defined for any graph G by the recurrence relation $P(G) = \max\{P(G \setminus v)$ [1]$: v \in V(G)\}, P(K_1) = 1, P(K_2) = 2$. Table 5.1 contains the value of P for all unlabelled connected graphs with up to 4 vertices. Figure 5.1 shows how OLGA evaluates $P(G)$ using look-ups in the database (in this case Table 5.1), when it encounters a graph with 5 vertices. Let G be the graph shown on the left of Figure 5.1 for which we want to compute $P(G)$. Since the recurrence relation uses vertex deletion, there are 5 graphs resulting after deleting a vertex. We use parallelisation to speed-up the process, therefore the graphs resulting due to vertex deletion get uniformly distributed among all available threads (in this case there are prescribed 2 threads) for computation. Note that these graphs resulting from vertex deletion are all isomorphic to some graph in Table 5.1. Therefore, we compute the canonical label for each of these graphs. We then fetch the values of P for these graphs using their already computed canonical label from Table 5.1 via look-ups. Each thread returns the maximum value of the parameter among the graphs it processed; in this case Thread 1 returns 3 and Thread 2 returns 3. Finally the threads interact among themselves to obtain the maximum value across all threads; in this case the maximum value is 3. Note that the number of look-ups needed in this case is 5.

[1] if $G \setminus v$ is disconnected then $P(G) = \max P(C_i)$, where $\cup_i C_i = G$, $P(C_i)$ is already computed as $|V(C_i)| < |V(G)|$.

Canonical label	# edges	Parameter value
1	3	2
2	3	2
3	4	3
4	4	2
5	5	3
6	6	4

Table 5.1: $P(G)$ for all unlabelled connected graphs of 4 vertices.

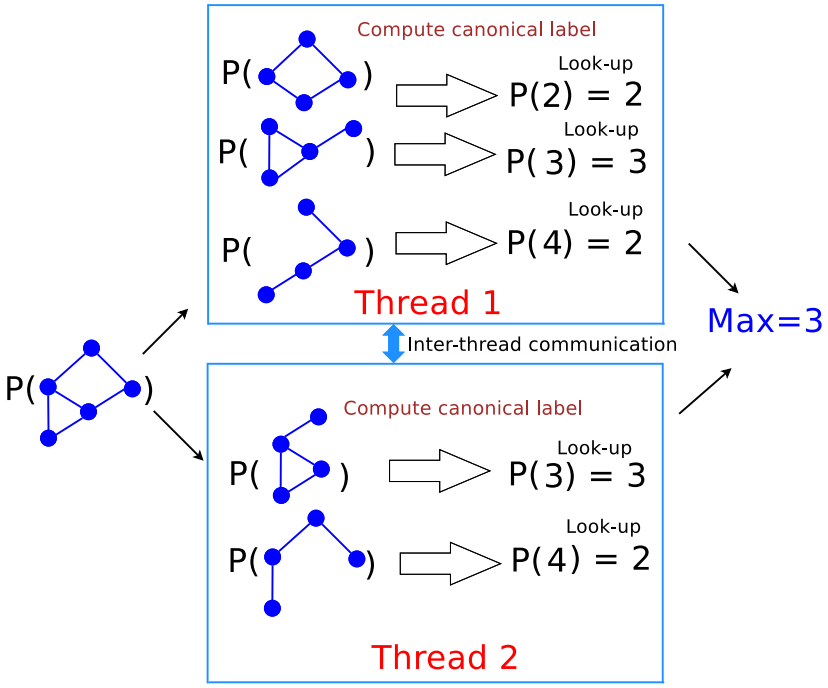


Figure 5.1: Vertex deletion in OLGA.

Edge deletion

The value of the graph parameters for small graphs (based on the base case of the recursion) first gets stored in OLGA. For graphs of higher order, the edge deletion process is parallelised by using different threads for each graph generated by deleting an edge. Each thread then fetches the value of the parameter of the graph resulting from edge deletion from the already-stored parameter value in OLGA. This step is tricky as the graphs were generated in order of increasing number of vertices, followed by increasing number of edges.

Both ordering by number of vertices and ordering by number of edges are important for OLGA, as without the ordering of the number of edges, edge deletion might result in a graph for which the parameter value is not yet computed as the order of the graph is same. In this case the recursion would keep on using the edge deletion process until it

finds a graph whose parameter value is already computed (known), which in the worst case would take an exponential number of operations.

The threads communicate among each other to compute the minimum or maximum value fetched by the threads, specified by the recurrence. Vertex deletion always needs $O(n)$ look-ups to compute the value of a parameter under the OLGA setup. Edge deletion is more look-up heavy than vertex deletion, it takes $O(m)$ look-ups in the average case. Edge deletion in the worst case can take $O(n^2)$ look-ups, we will discuss this in more detail in Section 6.4.2. Therefore, more memory is needed for edge deletion as deleting an edge does not always guarantee reduction in the number of vertices. The inter-communication between threads requires a careful implementation as threads fetch the values from OLGA at different times.

Consider the following example. Let P be the parameter defined for any given graph G by the recurrence relation $P(G) = \max\{P(G \setminus e)$ [2]: $\forall e \in E(G)\}, P(K_1) = 1, P(K_2) = 2$. Tables 5.1 and 5.2 contain the values of P for all unlabelled connected graphs with up to 4 vertices and some graphs of 5 vertices whose number of edges is less than 6. Figure 5.2 shows how OLGA evaluates $P(G)$ when it encounters a graph with 5 vertices and more edges than those graphs already consulted by looking-up in the database (in this case Table 5.1, Table 5.2). Since the recurrence relation uses edge deletion, there are 6 graphs remaining after deleting an edge. The graphs resulting from edge deletion get uniformly distributed among all available threads (in this case there are prescribed 2 threads) for faster computation via parallelisation. Note that the graphs resulting from edge deletion are all isomorphic to some graph in Table 5.1 or in Table 5.2. Therefore, we compute the canonical label for each of these graphs and use it to get the value of P for the corresponding graph via look-ups. Each thread returns the maximum value of the parameter among the graphs it processed; in this case Thread 1 returns 3 and Thread 2 returns 2. Finally the threads interact among themselves to obtain the maximum value across all threads; in this case the maximum value is 3. Note that the number of look-ups needed in this case is 6. Furthermore the look-ups used in this example successfully fetch the value from the database as the graphs are generated in order of increasing number of edges.

Parallel implementation of **edge contraction** and **vertex identification** are similar to that of vertex deletion.

Not all graph parameters follow a nice recursive structure. In these cases one may still be able to get a recursive lower bound and upper bound for the parameter. If the bounds match, then that value is the exact value, otherwise one has to find the exact solution by another method. Since OLGA stores many graph parameters and most of them are computed using recursion (which need look-ups, which is a costly operation), a natural question arises: does the recursive strategy guarantee that the parameter value is computed in the most efficient computation time?

Consider the data represented in Table 1.1. The look-ups are carried out seamlessly for computing graph parameters for graphs up to 11 vertices, as the size of the RAM (4

[2] if $G \setminus e$ is disconnected then $P(G) = \max P(C_i)$, where $\cup_i C_i = G$ and $P(C_i)$ is already computed as $|V(C_i)| < |V(G)|$ and $E(C_i) \subset E(G)$.

Canonical label	# edges	Parameter value
1	4	2
2	5	2
3	5	3
4	5	2
⋮	⋮	⋮
12	5	3
13	5	3
⋮	⋮	⋮
17	5	2
18	5	2
19	5	2

Table 5.2: $P(G)$ for some unlabelled connected graphs of 5 vertices.

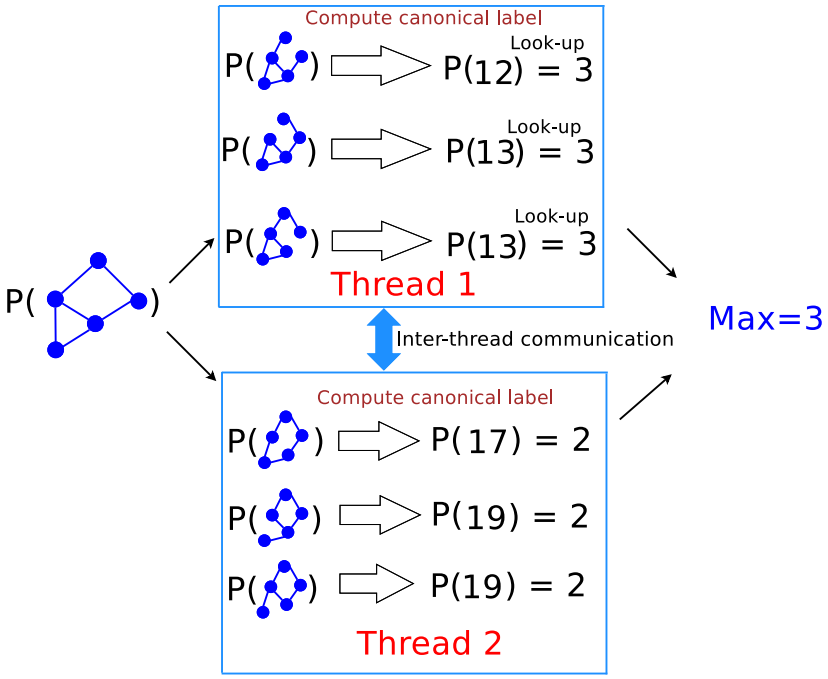


Figure 5.2: Edge deletion in OLGA.

GB in a standard PC) is more than the number of connected unlabelled graphs of order 11. The scenario changes when the order of the graph is more than 11, as the number of graphs exceeds the size of the RAM. Therefore, the graphs and their related data must be obtained by look-ups from the secondary memory. This results in page faults and increased computation time. Exponential-time exact algorithms, on the other hand, can compute the graph parameters without any page faults when the space requirement is polynomial. The time complexity of many exponential time search tree algorithms can be reduced at the cost of an exponential space complexity via *memoization* which was introduced by Robson [125]. Memoization works as follows: the solutions of all the subproblems solved are stored in an (exponential-size in terms of size of input) database. If the same subproblem turns up more than once, the algorithm is not run a second time, but the previously computed result is looked up. A list of algorithms using exponential space to improve time complexity is given in [49]. Since most of the parameters in OLGA are computed using recursion, we use memoization for their efficient implementation.

If the exact algorithm takes exponential space, it results in page faults while computing parameters of graphs of higher order. As a result we have employed a balanced approach between *recursive* computation and *exact* computation of parameters.

To investigate this balance we introduced a new parameter, *most frequent connected induced subgraph (MFCIS)*, defined as follows:

> The *frequency* of an unlabelled graph H in a graph G is the number of induced subgraphs of G that are isomorphic to H. The graph H is a *Most Frequent Connected Induced Subgraph (MFCIS)* of G if the frequency of H in G is maximum among all unlabelled graphs occurring as induced subgraphs of G.

We will discuss MFCIS further in Chapter 7. Its exact computation is super-exponential in both time and space. We compare the time complexity of this parameter with a parameter which requires a lot of look-ups during its recursive computation in Section 5.3.1.

5.3.1 I/O-bound jobs and CPU-bound algorithms in OLGA

In any operating system the physical memory is partitioned into small logical addressable portions called *pages*. The operating system uses these pages to transfer data between primary storage and secondary storage.

When a program gets executed the operating system allocates a fixed memory for its execution. If the program needs some data that is not present in its allocated memory, it is called *page fault*. If the program's requested data is in the primary storage but not accessible to the program's allocated memory then it is a *minor page fault*. If the program's requested data is in the secondary storage it is a *major page fault*. Note that page fault is not an error in the program, it is only a way to indicate the program will need some extra time as the memory needed to serve the requests of the program is not locally available. Clearly a major page fault results in more waiting time for the program, as it involves i/o-lookups.

To compare the time-complexity of a parameter with lots of look-ups as a parameter that takes exponential time and space, we chose chromatic index and MFCIS. We run this

experiment keeping OLGA as our focus, so these results cannot be generalised. We run this experiment in the following machine.

System specification:

Hardware: OptiPlex 9020 (OptiPlex 9020) (Dell Inc.)
Width: 64 bits
Product: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
Clock: 100MHz
RAM: 8GB
O/S: Ubuntu 16.04.5 LTS

We first list the numbers of page faults (minor and major) and run time for computing the MFCISs of all unlabelled graphs with order up to 10.

```
$ THREADS=2 OPTION=1 START=3 STOP=10 ./welcome
*** This is for computing MFCIS up to 10 vertices ***
$ pgrep welcome
32453     >>>>>>>> This is the process id
$ top|grep 32453
32453 sswa9 20 0 7865284 6.330g 720 D 0.3 82.2 791:12.16 welcome

$ ps -o min_flt,maj_flt 32453
MINFL   MAJFL    >>>> Minor fault  Major fault
4939961 1880075

Run time: ~8hr 45 mins

$ THREADS=2 OPTION=1 START=3 STOP=9 ./welcome
*** This is for computing MFCIS up to 9 vertices***
$ pgrep welcome
11136     >>>>>>>> This is the process id

$ ps -o min_flt,maj_flt 11136
MINFL   MAJFL
34262    14

Run time: ~21 mins
============================================================
```

Now we list the numbers of page faults (minor and major) and run time for computing the chromatic index of all unlabelled graphs with order up to 10. We chose chromatic index for our experiment, as the recursive bounds used for computing chromatic index are based on edge deletion. This results in more look-ups than computing any other parameter based on vertex deletion or edge contraction.

```
Chromatic index computation
$ THREADS=2 OPTION=3 START=3 STOP=10 ./welcome
*** This command is for computing chromatic index up to 10 vertices ***
$ pgrep welcome
3919
$ ps -o min_flt,maj_flt 3919
MINFL    MAJFL
2162278  43189

Run time: ~4hrs
```

```
$ THREADS=2 OPTION=3 START=3 STOP=9 ./welcome
*** This command is for computing chromatic index up to 9 vertices ***
$ pgrep welcome
10993
$ ps -o min_flt,maj_flt 10993
MINFL    MAJFL
33283     0

Run time: ~7 mins
============================================================
```

The running time for chromatic index is significantly less than the running time of MFCIS when the input is of order ≤ 10. Therefore under the OLGA setup recursive algorithms are expected to perform better when the corresponding exact algorithm requires exponential space.

5.4 Efficient storage

The number of graphs grows super-exponentially with respect to the number of vertices [66, 120] (see Table 1.1). Storing these graphs is a challenging task when the number of vertices is more than 10. The most commonly used method to store graphs is to store them in an adjacency list or adjacency matrix. However, these methods require lot of space to store all connected unlabelled graphs up to 13 vertices. OLGA stores the graphs in graph6 format, which is a compressed form of storing graphs, introduced by McKay [92]. Table 5.3 lists the amount of space needed for storing all graphs up to 13 vertices in graph6 format.

In OLGA, for each graph we also store its canonical label (also known as certificate, see [90]). OLGA also stores the value of various parameters which plays an important role in making OLGA interactive. A natural storage solution is to store the data using MySQL database engine [105], which is more useful in dealing with high volumes of data. However, in the current setup of OLGA, we are storing graphs and parameters up to 10 vertices. Therefore, we use *SQLite*, which is not only a very powerful, embedded relational database management system [137] but also easy to manage and implement. When an application uses SQLite, the complications of communicating with interface such as ports, sockets is completely eliminated. This makes SQLite extremely fast and efficient thanks to the library's underlying technology [137].

We store all the parameters in a single database, except for eigenvalues and the Tutte polynomial. These two parameters are stored in two separate databases as these parameters take approximately polynomial space for a graph with n vertices. When OLGA gets a query from a user, appropriate joins are performed between these databases. Details of query processing are given in Section 5.5.

5.5 Query optimization

OLGA is a queryable database of graphs, and therefore an efficient query execution plan will reduce the turn-around time of the query. Consider the following example: suppose a

# vertices	Memory required
1	2B
2	3B
3	6B
4	18B
5	84B
6	560B
7	5.1KB
8	77.8KB
9	2.1MB
10	117.2MB
11	12.1GB
12	≈ 2.13TB
13	≈ 760TB

Table 5.3: Space requirements to store graphs in graph6 format.

user query is of the form $c_1 \wedge c_2 \wedge \cdots \wedge c_k$, where each c_i is a condition on some parameter. Let $n_i(c_i)$ denote the number of graphs satisfying c_i. The execution time of the query depends on the sequence in which the query is executed; that is, if $n_1(c_1) \ll n_2(c_2)$, then executing c_1 before c_2 will make the execution of the query faster as the input for c_2 is $n_1(c_1)$ graphs, whereas reversing the order of execution will require more time as c_1 is applied over $n_2(c_2)$ graphs.

For example, consider the following query: list all 4 vertex graphs with vertex connectivity 2 and chromatic number 4. There are two ways to carry out this query: *either* find all 4 vertex graphs that are 2-connected and among them find those which have chromatic number 4 *or* execute in the reverse order. The first approach is shown in Table 5.4 and the second approach is shown in Table 5.5. Clearly the second approach is faster than the first approach.

Can_id	# edges	Ver_conn	Chr_no
7	3	1	2
13	3	1	2
15	4	1	3
30	4	2	2
31	5	2	3
63	6	3	4

Step 1 →

Can_id	# edges	Ver_conn	Chr_no
30	4	2	2
31	5	2	3
63	6	3	4

Step 2 ↓

Can_id	# edges	Ver_conn	Chr_no
63	6	3	4

Table 5.4: Query execution using the first approach.

Can_id	# edges	Ver_conn	Chr_no
7	3	1	2
13	3	1	2
15	4	1	3
30	4	2	2
31	5	2	3
63	6	3	4

Step 1 →

Can_id	# edges	Ver_conn	Chr_no
63	6	3	4

Step 2 ↓

Can_id	# edges	Ver_conn	Chr_no
63	6	3	4

Table 5.5: Query execution using the second approach.

Finding the best sequence of execution for queries is difficult as the sequence is query dependent. The order of execution which gives better turn-around time for one query can perform poorly for another query if the parameters change in the query. Moreover, if the query is of the form $c_1 \vee c_2 \vee \cdots \vee c_k$ then the sequence of execution has no impact on the execution time. For example, if the query is to list all 4 vertex graphs with vertex connectivity 2 *or* chromatic number 4. Then the order of execution has no impact on the turn-around time. An example is shown in Table 5.6.

Can_id	# edges	Ver_conn	Chr_no
7	3	1	2
13	3	1	2
15	4	1	3
30	4	2	2
31	5	2	3
63	6	3	4

Step 1 →

Can_id	# edges	Ver_conn	Chr_no
30	4	2	2
31	5	2	3
63	6	3	4

Step 2 ↓

Can_id	# edges	Ver_conn	Chr_no
30	4	2	2
31	5	2	3
63	6	3	4

Table 5.6: Query execution independent of execution sequence.

In order to develop a general strategy that will make query execution faster, we took advantage of the database. While storing the data in SQLite we used (SQL) *indices* for

each parameter, except for the Tutte polynomial. *Indices* are special lookup tables that the database search engine can use to speed up data retrieval [106]. An index is a pointer to data in a database table. An index in a database is very similar to an index of a book. Since OLGA has only few parameters (< 30), the indices take little space. This technique improved the turn-around time for any kind of search in OLGA.

5.6 OLGA architecture

We use the client-server architecture [13] for OLGA. Since the contents of the OLGA do not change, we store them in the server database. OLGA treats all user systems as clients which send queries to the server.

An important part of the design is efficient query execution. Since OLGA is designed to facilitate user queries on graphs and parameters, we needed a lightweight, scalable and simple web framework to enable users to query from anywhere (geographically). We used *django* as our web framework. Django's primary goal is to ease the creation of complex, database-driven websites. Django emphasises reusability and "pluggability" of components, less code, low coupling, rapid development, and the principle of no redundancy in action [52].

Chapter 6

Graph parameters

In this chapter we list the graph parameters present in OLGA. For most of the listed parameters, we present the recursion used for its computation. Some parameters do not have a proper recursive structure so, for them we list the recursive bounds and compute their values through exact algorithms. We also report our findings on the recursive bounds.

The recursions used in this chapter are mostly elementary and presumably old. It is hard to pinpoint the origins for these recursions, however we give the proofs for completeness.

6.1 Metric parameters

In this section we discuss various parameters that measure the distances between vertices and lengths of cycles in a graph.

6.1.1 Radius and diameter

The *distance* between a pair of vertices u and v in a graph G, denoted by $\text{dist}(u, v)$, is the length of the shortest path joining u and v. If G has no uv-path, the distance between u and v is defined to be ∞. The *eccentricity* $\text{ecc}(v)$ of a vertex v in a graph G is the maximum distance between v and any vertex of G. That is,

$$\text{ecc}(v) = \max\{\text{dist}(v, u) : u \in V(G)\}.$$

The *radius* of G is the minimum eccentricity, i.e.,

$$\text{rad}(G) = \begin{cases} \min\{\text{ecc}(v) : v \in V(G)\}, & \text{if } G \text{ is connected,} \\ \infty, & \text{otherwise.} \end{cases}$$

The *diameter* of G is the maximum eccentricity, i.e.,

$$\text{diam}(G) = \begin{cases} \max\{\text{ecc}(v) : v \in V(G)\}, & \text{if } G \text{ is connected,} \\ \infty, & \text{otherwise.} \end{cases}$$

Alternatively, $\text{diam}(G) = \max\{\text{dist}(u, v) : u, v \in V(G)\}$.

We use the Floyd-Warshall algorithm [42, 152], which is based on a simple dynamic programming approach, to calculate distance between every pair of vertices in $O(n^3)$ time. The heart of the Floyd-Warshall algorithm is

$$\text{dist}(i, j) = \min\{\text{dist}(i, j), \text{dist}(i, k) + \text{dist}(k, j), \forall k \in V(G)\}.$$

We compute the radius and diameter values from the output of the Floyd-Warshall algorithm.

Recursion for diameter and radius

Theorem 6.1.1. *For a connected graph G, $diam(G/e) \leq diam(G) \leq diam(G/e) + 1$.*

Proof. Consider the following cases.

- Contracting an edge will not increase the distance between any two vertices.
- Contracting an edge will not disconnect the graph. For any two vertices $u, v \in V(G)$, suppose $\text{dist}_G(u, v) = d$. Contracting an edge $e \in E(G)$ will result in the following.
  - Suppose e is not in any shortest path between u and v. In G/e, $\text{dist}_{G/e}(u, v) = d$.
  - Suppose e is in a shortest path between u and v. In G/e, $\text{dist}_{G/e}(u, v) = d - 1$.

Therefore, for a connected graph G, $\text{diam}(G/e) \leq \text{diam}(G) \leq \text{diam}(G/e) + 1$. □

Theorem 6.1.2. *For a connected graph G, $rad(G/e) \leq rad(G) \leq rad(G/e) + 1$.*

OLGA implementation for diameter and radius

Diameter and radius are also part of Barnes's version of OLGA [10]. We used the Floyd-Warshall algorithm to compute diameter and radius. More information on the implementation is given in Section A.3.

6.1.2 Longest path

The *max-length* between a pair of vertices u and v in a graph G, denoted by $\text{max-dist}(u, v)$, is the maximum length of all (u, v)-paths in G. A (u, v)-path in G with length $\text{max-dist}(u, v)$ is called a *longest path* from u to v. The *longest path* of G, denoted by $\text{lp}(G)$, is a (u, v)-path with maximum $\text{max-dist}(u, v)$ for all $u, v \in V(G)$, i.e.,

$$\text{lp}(G) = \begin{cases} \max\{\text{max-dist}(u, v) : u, v \in V(G)\}, & \text{if } G \text{ is connected,} \\ \max\{\text{lp}(C_1), \text{lp}(C_2), \cdots, \text{lp}(C_k)\}, & \text{otherwise.} \end{cases}$$

where $C_1, C_2, \cdots, C_k$ are the components of G.
Finding a longest path in a graph is NP-hard [55].

Recursion for longest path

Fact 6.1.3. *If G is a connected graph of order n, with $\delta(G) = 1$ and $\Delta(G) \leq 2$, then $lp(G) = n - 1$.*

Lemma 6.1.4. *For any connected non-path graph G there exists an edge $e \in E(G)$, such that $G \setminus e$ is connected and $lp(G \setminus e) = lp(G)$.*

Proof. Suppose G is a non-path connected graph and P is a longest path in G with endpoints u, v, for some $u, v \in V(G)$. There is an edge $e \in E(G)$ such that $e \notin E(P)$, since G is a connected non-path graph. Therefore, $\text{lp}(G \setminus e) = \text{lp}(G)$. □

OLGA implementation for longest path

The recursion for longest path in Lemma 6.1.4 is part of Barnes's version of OLGA [10]. We used this recurrence for computing longest path of a graph. However we also implemented a brute force exact algorithm to compute longest path of a graph to confirm our results from the recursive algorithm. More information on the implementation is given in Section A.3.

6.1.3 Girth and circumference

The *girth* of a graph is the length of the shortest cycle. If the graph is acyclic or disconnected, the girth is defined to be infinite. For a disconnected graph with components $C_1, C_2, \cdots, C_k$ the girth is the smallest girth among the components:

$$\text{girth}(G) = \min\{\text{girth}(C_1), \text{girth}(C_2), \cdots, \text{girth}(C_k)\}.$$

In the connected case, we compute the girth in polynomial time through a series of breadth-first traversals using Algorithm 2. Clearly, the running time of this algorithm is $O(|V|(|V|+|E|)) = O(|V||E|)$.
The *circumference* of a graph, denoted by $\text{circ}(G)$ is the length of the longest cycle. If the graph is acyclic the circumference is infinite. Unlike the girth, which is computed in polynomial time, finding the circumference is NP-hard [55].

Fact 6.1.5. *If G is connected and 2-regular, then the circumference of G is n.*

Fact 6.1.6. *If G is acyclic, then the circumference of G is ∞.*

If G is disconnected, with components $C_1, C_2, \cdots C_k$ such that

$$\text{circ}(G) = \max\{\text{circ}(C_i) : i \in [1, k], \text{circ}(C_i) < \infty\}.$$

If G has a cut vertex, then the longest cycle in G must be contained within a block. Therefore, if $B_1, B_2, \cdots, B_k$ are the blocks in G, then

$$\text{circ}(G) = \max\{\text{circ}(B_i) : i \in [1, k], \text{circ}(B_i) < \infty\}.$$

For an efficient implementation of circumference we follow the following strategy. We search for a cut-vertex using depth-first search and partition the graph into two parts based

Algorithm 2: Find the girth of a graph

```
1  girth(G) = ∞ ;                          /* Size of the smallest cycle found.  */
2  for every v ∈ V(G) do
3      S = ∅
4      R = {v}
5      Parent(v) = NULL
6      D(v) = 0 ;                                   /* D(w) = dist(v, w) */
7      while R ≠ ∅ do
8          choose x ∈ R
9          S = S ∪ {x}
10         R = R \ {x}
11         for every y ∈ Neighbour(x) \ {Parent(x)} do
12             if y ∉ S then
13                 Parent(y) = x
14                 D(y) = D(x) + 1
15                 R = R ∪ {y}
16             else
17                 girth(G) = min{girth(G), D(x) + D(y) + 1}
18 return girth(G)
```

on the cut-vertex. We compute the circumference of each part to find the circumference of the graph.

Lemma 6.1.7. *If G has connectivity ≥ 3, then $circ(G) = \max\{circ(G \setminus e) : e \in E \wedge circ(G \setminus e) < \infty\}$.*

Proof. Let G be a k-connected graph with $k \geq 3$. Let C be a longest cycle in G. Let $e \in E(G)$ such that $e \notin C$. Such an edge always exists in G, as G is 3-connected (which implies each edge in G is not part of every cycle of G). Therefore, $\text{circ}(G) = \text{circ}(G \setminus e)$. If $e \in C$, then $\text{circ}(G) \geq \text{circ}(G \setminus e)$.

Hence $\text{circ}(G) = \max\{\text{circ}(G \setminus e) : e \in E \wedge \text{circ}(G \setminus e) < \infty\}$. □

OLGA implementation for girth and circumference

Both girth and circumference were part of Barnes's version of OLGA [10]. The recursion for circumference in Lemma 6.1.7 is part of Barnes's honours thesis [10]. We used this recurrence for computing the circumference of a graph. However our implementation is a parallel implementation of the recursion using Lemma 6.1.7. More information on the implementation is given in Section A.3.

6.2 Algebraic parameters

In this section we discuss parameters related to the adjacency matrix of a graph and graph polynomials.

6.2.1 Tutte polynomial

Graph polynomials can contain useful information about graphs. One such polynomial is the Tutte polynomial, a renowned tool for capturing properties of graphs and networks. This is a two-variable polynomial, discovered by W. T. Tutte [145, 146]. This polynomial encapsulates some other useful graph polynomials including the chromatic polynomial, flow polynomial and reliability polynomial [131]. The Tutte polynomial not only captures combinatorial information about graphs but also encodes information relevant to physical applications. The Tutte polynomial encapsulates other important parameters in effect, e.g. number of colourings, number of non-zero flows, number of spanning trees, number of forests, number of spanning subgraphs, number of acyclic orientations (see [58]). The Tutte polynomial of a graph G is defined as

$$T(G;x,y) = \sum_{X \subseteq E(G)} (x-1)^{r(E)-r(X)}(y-1)^{|X|-r(X)},$$

where $r(X)$ is the rank of X. Let the edges of G be linearly ordered. Let T be a spanning tree of G. For an edge e outside T, the cycle created by adding e to T is called the *fundamental cycle* of e with respect to T. This edge is called *externally active* if e is less than all other edges on its fundamental cycle under the order on the edges. For an edge e inside T, the *fundamental cutset* of e with respect to T is the cutset of G containing e for which all edges in the set except e are outside T. If e is less than all other edges on its fundamental cutset, e is *internally active* [146]. The Tutte polynomial can be represented as a generating function where the coefficient counts the number of spanning trees having given numbers of internally and externally active edges,

$$T(G;x,y) = \sum_{i,j} t_{ij} x^i y^j,$$

where t_{ij} counts the number of spanning trees with i internally active edges and j externally active edges [146].

Recursive properties of Tutte polynomial

The Tutte polynomial can also be defined by deletion and contraction relations. If e is an edge of G and it is not a bridge or a loop, then

$$\begin{aligned} T(G;x,y) &= T(G \setminus e;x,y) + T(G/e;x,y). \\ T(K_1) &= 1. \\ T(K_2) &= x. \\ T(K_1^{\odot}) &= y, \text{ where } K_1^{\odot} \text{ is a self loop.} \end{aligned}$$

If G is a graph with i bridges and j loops, then

$$T(G;x,y) = x^i y^j.$$

It is routine to show that $T(G \cup H; x, y) = T(G; x, y) \cdot T(H; x, y)$, where $\cup$ is disjoint union of graphs [146]. The Tutte polynomial is multiplicative over blocks.

OLGA implementation for Tutte polynomial

Two of the major challenges in implementing the Tutte polynomial are listed below.

1. Storing the polynomial: The data structure must be efficient for polynomial addition and multiplication.

2. Data retrieval: Each graph has a polynomial with $O(m)$ terms that needs to be stored. An efficient storage and retrieval mechanism is needed for reducing the query response time.

We implemented the recursive relation of the Tutte polynomial. In OLGA, we consider only simple graphs, however, we computed the Tutte polynomial as an exception. The Tutte polynomial takes more space than any other parameter stored in OLGA. So in the current version we store Tutte polynomials up to 7 vertices. For ease of implementation we use arrays for storage. However data retrieval is still a challenge. In the current version of OLGA we store Tutte polynomials in a separate database and we do not provide querying choice on Tutte polynomial. However we provide the option of displaying the Tutte polynomial for any graph of order ≤ 7.

6.2.2 Eigenvalues

Let A be an $n \times n$ real matrix. An *eigenvector* of A is a vector such that Ax is parallel to x; in other words, $Ax = \lambda x$ for some real or complex number λ. This number λ is called the *eigenvalue* of A belonging to eigenvector x. Clearly λ is an eigenvalue if and only if the matrix $A - \lambda I$ is singular, i.e., $\det(A - \lambda I) = 0$. This is a polynomial equation of degree n for λ, and hence has n roots (with multiplicity). The *trace* of the square matrix $A = (A_{ij})$ is defined as

$$\mathrm{tr}(A) = \sum_{i=1}^{n} A_{ii}$$

The trace of A is the sum of the eigenvalues of A, each taken with the same multiplicity as it occurs among the roots of the equation $\det(A - \lambda I) = 0$.

If the matrix A is *symmetric* (i.e., $A_{ij} = A_{ji}\ \forall i, j$), then its eigenvalues and eigenvectors are particularly well behaved. All the eigenvalues are real.

Let G be a (finite, undirected, simple) graph with vertex set $V(G) = \{1, \cdots, n\}$. The *eigenvalues of a graph* G are defined to be the eigenvalues of its adjacency matrix $A(G)$. The collection of the eigenvalues of G is called the *spectrum* of G. Since $A(G)$ is a real symmetric matrix, the eigenvalues of G, written as $\lambda_i(G), i = 1, 2, \cdots, n$, are real numbers. Therefore they can be ordered so that $\lambda_1(G) \geq \lambda_2(G) \geq \cdots \geq \lambda_n(G)$. The *i-th eigenvalue* of G is $\lambda_i(G)$. In particular $\lambda_2(G)$ is the second largest eigenvalue of G.

Let $D = \mathrm{diag}(d_1, \cdots, d_n)$ be the diagonal matrix where d_j is the degree of vertex v_j. The eigenvalues of $D - A(G)$ are called the *Laplacian eigenvalues* of G [32].

Importance of eigenvalues

The eigenvalues of a graph characterise some important properties of the graphs [32]. For example,

- if $\lambda_1(G) = -\lambda_n(G)$, then G is bipartite.
- if $\lambda_2(G) = 0$, then G is complete multi-partite.
- if $\lambda_2(G) = -1$, then G is a complete graph.

An interesting open question is to find all graphs whose eigenvalues are all distinct [67] (we discussed this in Chapter 4).

OLGA implementation for eigenvalues

We use the basic matrix operations to compute the eigenvalues of a graph. In OLGA, we use the default equation solvers (that come with the library of the programming language) to compute the eigenvalues. Therefore, the correctness of the eigenvalues depends on the precision of the eigenvalue solver used.

Storing eigenvalues for graphs require more space due to their multiplicity and their type. In the current version of OLGA, we store the eigenvalues in the same database where we store the Tutte polynomials.

6.2.3 Automorphism group

GI is in NP, but it is not known whether it is NP-complete or in P. It is one of the very few problems in NP whose complexity remains unknown [55]. Babai proposed a quasi polynomial time algorithm to check isomorphism between graphs [5]. Helfgott found an error in the timing analysis in Babai's work [4]. However Babai published another note with a new algorithm which takes quasi polynomial time [4].

The *subgraph isomorphism problem*, is the problem of finding a subgraph in G that is isomorphic to a different given graph H. This problem is known to be NP-complete [55].

Computing the automorphism group $\mathrm{Aut}(G)$ of a graph G is more specialised version of GI, as the goal is to find isomorphism of the graph with itself. A brute force approach for computing $\mathrm{Aut}(G)$ is to list out all permutations on $V(G)$ and filter out the ones that are automorphisms. However, Unger [148] showed that with a brute force approach even a modern-day computer may not work beyond graphs of order 10.

We used the Schreier-Sims algorithm [132, 133] from the textbook [83] to compute the order of the automorphism group.

OLGA implementation for automorphism group

In OLGA we stored the vectors resulting from the Schreier-Sims representation and computed the size of the automorphism group using the vectors. We verified our results by comparing with [88].

6.3 Structural parameters

In this section we discuss parameters revealing structural properties of the graph.

6.3.1 Graph degeneracy

Degeneracy, introduced by Lick and White, is useful in measuring the sparseness of a graph [85]. A *d-degenerate* graph is an undirected graph in which every subgraph has a vertex of degree at most d. The *degeneracy* $\mathrm{dgn}(G)$ of a graph G is the smallest value d for which it is d-degenerate.

The degeneracy of a graph can be computed in linear time [11]. Degeneracy plays an important role in graph mining, network analysis [56], graph colouring and sensitivity analysis [54].

In OLGA, we use some elementary recursive properties of degeneracy which suit our framework for its computation. We list these properties with proof for completeness.

Theorem 6.3.1. *For every $v \in V(G)$, $dgn(G \setminus v) \leq dgn(G) \leq dgn(G \setminus v) + 1$*

Proof. Consider the following scenarios.

- Let $v \in V(G)$. As every subgraph of a graph with degeneracy $\mathrm{dgn}(G) = d$ has a vertex of degree at most d, the subgraph $G \setminus v$ has a vertex of degree at most d. So $\mathrm{dgn}(G \setminus v) \leq d$. Hence $\mathrm{dgn}(G \setminus v) \leq \mathrm{dgn}(G)$.

- Let $v \in V(G)$ and $\mathrm{dgn}(G \setminus v) = d$.

  A subgraph of G either has v in it or it does not.

  Every subgraph of G that does not contain v is a subgraph of $G \setminus v$. Therefore every such subgraph of G has a vertex of degree at most d in it.

  Every subgraph of G that contains v is a subgraph of $G \setminus v$ together with the vertex v and possibly some edges incident with v. Let G' be a subgraph of G that contains v. Then $G' \setminus v$ is a subgraph of $G \setminus v$. So it has a vertex u of degree $\leq \mathrm{dgn}(G \setminus v)$. Therefore, $\deg_{G'} u \leq \deg_{G' \setminus v} u + 1 \leq d + 1$.

  Therefore, every subgraph of G has a vertex of degree at most $d + 1$.

Therefore $\mathrm{dgn}(G \setminus v) \leq \mathrm{dgn}(G) \leq \mathrm{dgn}(G \setminus v) + 1$ □

OLGA implementation for degeneracy

We computed the degeneracy of a graph using the recurrence defined in Theorem 6.3.1. If the minimum upper bound and maximum lower bound of the graph (obtained from Theorem 6.3.1) coincide we report the value as the degeneracy. We verify the correctness of the value obtained from recurrence by comparing with the value obtained from exact computation of degeneracy. Our results gave us the following theorem.

Theorem 6.3.2. *Let G be a graph of order $n \leq 12$. For all $v \in V(G)$, $\max(dgn(G \setminus v)) = \min(dgn(G \setminus v) + 1)$.*

Proof. We proved Theorem 6.3.2 by exhaustively considering all possible graphs up to 12 vertices. □

However, the problem of finding a short(er) proof for the theorem is still open.

6.3.2 Maximum matching

A set $M \subseteq E$ is a matching if no two edges in M have a common vertex. A vertex v is *matched* by M if it is contained in an edge of M, and *unmatched* otherwise. The *maximum matching size* of a graph G, denoted by $\mathrm{mm}(G)$, is the size of a matching M of maximum size in G. The maximum matching problem in bipartite graphs can be easily solved in $O(|E|\sqrt{|V|})$ time using Dinic's algorithm [43]. Edmonds's algorithm [47] takes $O(|V|^2|E|)$ time to find any maximum matching in a graph (not necessarily bipartite). Micali and Vazirani subsequently improved Edmonds's algorithm to run in time $O(\sqrt{|V|}|E|)$ time [98].

Fact 6.3.3. *If G is disconnected, with components $C_1, C_2, \cdots, C_k$, the maximum matching size is given by the sum of the maximum matching size of each component. Therefore*

$$mm(G) = \sum_{i=1}^{k} mm(C_i).$$

Recursion for maximum matching

Theorem 6.3.4. *For any connected graph $G \ncong K_2$, $mm(G) = max\{mm(G \setminus e) : e \in E\}$*

Proof. Consider the following cases:

- Removing an edge e from a graph G will not increase the size of any matching.
- Suppose G is a connected graph and M is a maximum matching in G. There is an edge $e \in E(G)$ such that $e \notin M$, since G is connected and $G \ncong K_2$. Therefore, $\mathrm{mm}(G \setminus e) = \mathrm{mm}(G)$.

□

OLGA implementation for maximum matching

Maximum matching and Theorem 6.3.4 were part of Barnes's prototype of OLGA [10]. In OLGA we use the Edmonds algorithm to compute the maximum matching. Although we have Theorem 6.3.4, we preferred the Edmonds algorithm due to its polynomial running time.

6.3.3 Independence number, vertex cover number, and clique number

Independence number

The maximum independent set problem is useful in solving interval scheduling problems and graph colouring problems [81]. The independent set decision problem is as follows: Given a graph G and a positive integer q, is there an independent set of size q or more?

This was among the first few problems to be proven NP-complete [55]. If G is disconnected, with components $C_1, C_2, \ldots C_k$, a maximum independent set of G is the union of maximum independent sets of $C_1, C_2, \ldots C_k$; it follows that $\alpha(G) = \sum_i \alpha(C_i)$ [42].

We use some known recursive properties of independent sets to compute the independence number. We did an efficient implementation of the recursion to compute independence number by taking advantage of the OLGA architecture. We used the independence number to compute the clique number and vertex cover number of a graph. Some of the elementary results that we have used in computing the values of independence number, clique number and vertex cover number are listed below.

Lemma 6.3.5. *Let G be a graph and v be a vertex in $V(G)$, then* $\alpha(G) = \max\{\alpha(G \setminus v), \alpha(G \setminus v \setminus N[v]) + 1\}$.

Proof. Let G be a graph and I be a maximum independent set of G. If we delete any vertex $v \in V(G)$ then the following cases can arise:

- $v \notin I$: In this case deleting v has no impact on I, hence $\alpha(G) = \alpha(G \setminus v)$. However there must be a vertex $u \in N[v]$ such that $u \in I$, otherwise $I \cup \{v\}$ is independent, so that I is not a maximum independent set. Hence $\alpha(G) = \alpha(G \setminus v) \geq \alpha(G \setminus v \setminus N[v]) + 1$.
- $v \in I$: This implies $N[v] \cap I = \emptyset$, otherwise I is not an independent set. Therefore, $\alpha(G) = \alpha(G \setminus v) + 1 = \alpha(G \setminus v \setminus N[v]) + 1$.

Combining both the cases, $\alpha(G) = \max\{\alpha(G \setminus v), \alpha(G \setminus v \setminus N[v]) + 1\}$. □

Vertex cover

The vertex cover problem is well studied in [42, 152]. The decision version of a vertex cover problem is as follows: Given a graph G and a positive integer q, is there a vertex cover of size q or less? This problem is a well known NP-complete problem [55]. If G is disconnected, with components $C_1, C_2, \ldots C_k$, a minimum vertex cover of G is the union of minimum vertex covers of $C_1, C_2, \ldots C_k$; therefore $\mathrm{vc}(G) = \sum_i \mathrm{vc}(C_i)$ [42].

There is a well known relationship between vertex covers and independent sets [55]: $I \subseteq V(G)$ is an independent set if and only if the set $V(G) \setminus I$ is a vertex cover. It follows that if I is a maximum independent set, then $V \setminus I$ is a minimum vertex cover, and therefore $\mathrm{vc}(G) = n - \alpha(G)$. Hence we can use independence number to compute the vertex cover number.

Clique number

The problem of finding the size of a maximum clique in a graph is well studied [42, 152]. The decision version of the maximum clique problem is as follows: Given a graph G and a positive integer q, is there a clique of size q or more in G? This is an NP-complete problem [55]. Cliques are, in a sense, very similar to independent sets —while a clique refers to a set of mutually adjacent vertices, an independent set contains only mutually non-adjacent vertices. $C \subseteq V(G)$ is a clique in G if and only if C is an independent set

in $\overline{G}$; in particular, a maximum clique in $\overline{G}$ becomes the maximum independent set in G and vice versa. Hence $\omega(G) = \alpha(\overline{G})$ [42].

OLGA implementation for independence number, vertex cover number, and clique number

Independence number, vertex cover number, and clique number were part of Barnes's version of OLGA [10]. We used the recurrence defined in Lemma 6.3.5 for computing the independence number. We computed the vertex cover number and clique number using the relationships between independence number, clique number and vertex cover number.

6.3.4 Edge-connectivity

An *edge cutset* of G is a subset U of $E(G)$ such that $G - U$ is disconnected. Let k be a positive integer. A graph G is *k-edge-connected* if G has more than one vertex and every edge cutset has at least k edges. The *edge-connectivity* of a graph G, denoted by $\lambda(G)$, is the maximum integer k such that G is k-edge-connected. Note that $\lambda(G)$ equals the minimum number of edges whose removal results in a disconnected graph.

Fact 6.3.6. *1. $\lambda(G) = 0$ if and only if $G = K_1$ or G is disconnected.*

2. $\lambda(G) = 1$ if and only if G is connected and G contains a bridge.

3. $\lambda(K_n) = n - 1$ for all $n \geq 1$.

4. $\lambda(C_n) = 2$.

5. If T is a tree with at least one edge then $\lambda(T) = 1$.

Two paths in a graph G are *edge-disjoint* if they have no edges in common.

Recursion for edge connectivity

Theorem 6.3.7. *If e is an edge in any bridge-less graph G, $\lambda(G \backslash e) \leq \lambda(G) \leq \lambda(G \backslash e) + 1$.*

Proof. Consider the following cases:

- Removing an edge in a graph will not increase its edge-connectivity.
- Let k be the edge-connectivity of G. Since G is bridge-less, removing e will not disconnect G. If e is in any edge cutset of size k then $\lambda(G \setminus e) = k - 1$, otherwise $\lambda(G \setminus e) = k$.

□

OLGA implementation for edge-connectivity

We use the Ford-Fulkerson algorithm to compute the edge-connectivity [50]. Although we found a recursion on edge-connectivity, we chose the exact algorithm for ease of computation.

6.3.5 Vertex-connectivity

For a non-negative integer k, a graph G is *(vertex) k-connected* if G has more than k vertices, and $G - X$ is connected for every set $X \subseteq V(G)$ with $|X| \leq k$. That is, no two vertices of G are separated by fewer than k other vertices. Every (non-empty) graph is 0-connected. The *vertex-connectivity*, or simply *connectivity* of a graph G, denoted by $\kappa(G)$, is the maximum k such that G is k-connected.

Fact 6.3.8. *1.* $\kappa(K_n) = n - 1$.

2. $\kappa(G) = 0$ *if and only if* $G \cong K_1$ *or* G *is disconnected.*

3. $\kappa(G) = 1$ *if and only if* $G \cong K_2$ *or* G *contains a cut-vertex.*

4. If C_n *is the* n*-vertex cycle then* $\kappa(C_n) = 2$.

5. If T *is a tree with at least* 2 *vertices the* $\kappa(T) = 1$.

Recursion on vertex connectivity

Theorem 6.3.9. *If* v *is a vertex in any* 2*-connected graph* G, $\kappa(G \setminus v) \leq \kappa(G) \leq \kappa(G \setminus v) + 1$.

Proof. Consider the following scenarios:

- Removing a vertex in a graph will not increase its vertex-connectivity.
- Let k be the vertex-connectivity of G. Since G is 2-connected, removing v will not disconnect G. If v is in any vertex cut of size k then $\kappa(G \setminus v) = k - 1$, otherwise $\kappa(G \setminus v) = k$.

□

OLGA implementation for vertex connectivity

We used the Ford-Fulkerson algorithm to compute the vertex-connectivity [50]. Although we found a recursion on vertex-connectivity, we chose the exact algorithm for ease of computation.

6.3.6 Treewidth

Treewidth was introduced by Robertson and Seymour in 1983 [124]. Treewidth is discussed in detail in [46]. Treewidth is defined as follows. A *tree decomposition* of a graph G is a pair (T, X), where T is a tree and, $X = \{X_i : i \in V(T)\}$ is a family of subsets of V indexed by $V(T)$, satisfying the following properties:

- $\bigcup_i X_i = V(G)$. That is, each graph vertex is associated with at least one tree node. Each X_i is also called a *bag*. A *trivial bag* in a tree decomposition is a bag with exactly one vertex in it.

- For every $xy \in E(G)$, there exists $X_i \in X$ that contains both x and y.
- Let $1, 2$ and 3 be three nodes of T such that 2 lies on the path from 1 to 3. Then, if a vertex v of G belongs to both X_1 and X_3, it also belongs to X_2.

The *width* of a tree decomposition (T, X) is $\max_i(|X_i| - 1)$. The *treewidth* $TW(G)$ of a graph G is the minimum width over all tree decompositions of G.

Treewidth plays an important role in parameterized complexity [46]. Many graph theoretic problems which are NP-complete are polynomial time tractable when restricted to bounded treewidth [46]. The problem of finding the treewidth of a graph is NP-complete [55].

There are some exact algorithms for computing the treewidth of a graph [16]. Most of these algorithms take exponential time and exponential space, which makes the implementation difficult. In OLGA we compute the treewidth of a graph using treewidths of subgraphs of the graph. In OLGA, we use some elementary recursive properties of treewidth which suit our framework for its computation. These properties are listed below with proofs for completeness.

Lemma 6.3.10. *Let G be a graph and $v \in V(G)$, v is not a cutvertex, then $TW(G) \leq TW(G \setminus v) + 1$.*

Proof. Let (T, X) be a minimum width tree decomposition of $G \setminus v$. Let us add v to every bag to obtain (T', X'), where $X' = \{X'_i : i \in V(T)\}$ and $X'_i = X_i \cup \{v\}$ for all $i \in V(T)$. Then the following holds.

- $\bigcup_i X'_i = \bigcup_i (X_i \cup \{v\}) = (\bigcup_i X_i) \cup \{v\} = (V(G) \setminus \{v\}) \cup \{v\} = V(G)$.
- Every every edge $xy \in E(G)$ with $x \neq v \neq y$ is also an edge in $G \setminus v$. Since (T, X) is a tree decomposition of $G \setminus v$, there is a bag $X_i \in X$ that contains both x and y. Since $X'_i = X_i \cup \{v\}$, the bag $X'_i \in X'$ contains both x and y.

  Let one of x or y be v, without loss of generality, let $v = y$. Since (T, X) is a tree decomposition of $G \setminus v$, there is a bag $X_i \in X$ that contains x. By construction of X' we know that $X'_i = X_i \cup \{v\}$. So the bag $X'_i \in X'$ contains both x and $v(= y)$.

  Therefore for every edge $xy \in E(G)$ there is a bag $X'_i \in X'$ containing both x and y.

- Suppose $X'_i = X_i \cup \{v\}$, $X'_j = X_j \cup \{v\}$, for $i, j \in V(T)$. There exists a unique path in T between i and j. Let k be a node in the path between i and j. Let $u \in X'_i \cap X'_j$. We will show $u \in X'_k$. Since (T, X) is a tree decomposition of $G \setminus v$, if $X_i \in X$ and $X_j \in X$ both contain a vertex u, then all nodes $k \in V(T)$ of the tree in the (unique) path between i and j contain u as well. So $u \in X_k$, hence $u \in X'_k$. That is, for every $X'_i, X'_j, X'_k \in X'$ such that k is in the path from i to j, then $X'_i \cap X'_j \subseteq X'_k$.

Hence (T, X') is a tree decomposition of G. Moreover $\forall i, |X'_i| = |X_i| + 1$. So the width of (T, X') is $TW(G \setminus v) + 1$. Hence $TW(G) \leq TW(G \setminus v) + 1$. □

Lemma 6.3.11. *Let G be a graph and $v \in V(G)$ is not a cut-vertex, then $TW(G \setminus v) \leq TW(G)$.*

Proof. Let (T, X) be a minimum width tree decomposition of G with no trivial bags. Let us remove v from every bag to obtain (T, X'), where $X' = \{X_i \setminus \{v\} : X_i \in X\}$. Then the following properties are satisfied.

- $\bigcup_i X'_i = \bigcup_i (X_i \setminus \{v\}) = V(G) \setminus v$.
- Every $xy \in E(G \setminus v)$ is also an edge in G. Since (T, X) is a tree decomposition of G there exists $X_i \in X$ such that $x, y \in X_i$. This implies $x, y \in X_i \setminus v$. That is for every edge $xy \in E(G \setminus v)$ there is a bag $X'_j = X_i \setminus \{v\} \in X'$ that contains both x and y.
- Since (T, X) is connected with no trivial bags, (T, X') is also connected. Let $X'_i, X'_j, X'_k \in X'$ be such that j lies on the path from i to k. Since each $X'_i \subseteq X_i$, j lies on the path from i to k. As (T, X) is a tree decomposition of G, if a vertex $u \neq v$ of G belongs to both X_i and X_k, it also belongs to X_j. So if a vertex u of G belongs to both X'_i and X'_k, it also belongs to X'_j.

Hence (T, X') is a tree decomposition of $G \setminus v$. Moreover $\forall i, |X'_i| \leq |X_i|$. So the width of $(T, X') \leq TW(G)$. Hence $TW(G \setminus v) \leq TW(G)$. □

Proposition 6.3.12. *Let G be a graph and $v \in V(G)$, v is not a cut-vertex, then $TW(G \setminus v) \leq TW(G) \leq TW(G \setminus v) + 1$.*

Lemma 6.3.13. *Let $H \leq G$, then $TW(H) \leq TW(G)$.*

Proof. Let (T, X) be a minimum width tree decomposition of G. If $H \leq G$, then the following cases arise:

- If $E(H) \subseteq E(G)$ and $V(H) = V(G)$ then (T, X) is a tree decomposition of H.
- If $v \in V(G) \setminus V(H)$, then (T, X') is a tree decomposition of H, where $X' = \{X'_i : X'_i = X_i \setminus \{v\}, i \in V(T)\}$.

Therefore $TW(H) \leq TW(G)$.

□

Lemma 6.3.14. *If $S \subseteq V(G)$ is a clique of G, then in any tree decomposition (T, X) of G, there exists a node $X_i \in V(T)$ with $S \subseteq X_i$.*

Proof. We will prove it by induction on the size of the clique.

Induction Basis: Let the size of the clique is 2, the proposition is true for cliques of size 2 by the definition of tree decomposition. So the Lemma holds in this case.

Inductive step: Suppose this is true for cliques of size k. Let us consider a clique C of size $k + 1$. Suppose the proposition does not hold for C. Then there exists a tree decomposition (T, X) of G such that $C \nsubseteq X_i$, for each $X_i \in X$. Let us pick three distinct vertices $u, v, w \in C$. By the inductive hypothesis, the cliques $C \setminus u, C \setminus v$ and $C \setminus w$ must each occur in at least one bag. Let us denote these bags by X_u, X_v and X_w. Bags

Number of vertices	Unmatched Cases	
	Number	Percentage
3	1	50.00
4	2	33.00
5	4	19.04
6	19	16.96
7	109	12.77
8	1149	10.33
9	28980	11.10
10	1371471	11.70

Table 6.1: Treewidth bound unmatched cases.

X_u, X_v, X_w are distinct as if any two of them are same then C is contained in a bag and we are done. Otherwise, there exists a bag $X_p \in X$ from which each of the three bags can be reached via a path that does not go through the other bag. Then the following holds:

- $C \setminus \{u, v, w\} \subseteq X_p$, as these vertices $\{u, v, w\}$ are in the common intersection of all three bags.
- $u \in X_p$ as X_p lies between X_v and X_w which contains u. Similarly $v, w \in X_p$

Hence $C \subseteq X_p$, which is a contradiction to our assumption. Hence by induction the proposition holds. □

OLGA computation for treewidth

We computed treewidth using the recurrence defined in Theorem 6.3.12. If the minimum upper bound and maximum lower bound of the graph (obtained from Theorem 6.3.12) coincide we report the value as the treewidth. If the bounds do not match, we report such cases as unmatched. The number of unmatched cases are reported in Table 6.1. In the unmatched cases we used the exact algorithm of [16] to compute the treewith.

Data analysis

There are two questions arise from the data reported in Table 6.1:

- What is the closest lower and upper bounds that can be established for the proportion of unmatched cases, for sufficiently large n?
- Does the proportion of unmatched cases converge to a limit? It is unclear from the whether there is a limit point or not.

6.3.7 Domination number

A set of vertices D is a *dominating set* for a graph G, if every vertex of G is either in D or adjacent to a vertex which is in D. A *minimal dominating set* D of a graph G is a dominating set such that every proper subset of D is non-dominating. A smallest of all the

minimal dominating sets is a *minimum dominating set*. The cardinality of the minimum dominating set is known as the *domination number*, denoted by $\text{dom}(G)$.

The task of finding a minimum dominating set is a classical problem in computational complexity theory, known to be NP-hard [55]. The dominating set problem has many applications in networking, geographical surveillance and in the classical N-queens problem [68].

A minimum dominating set of a disconnected graph is a union of minimum dominating sets of each component. So if G has components $C_1, C_2, \cdots, C_k$, then

$$\text{dom}(G) = \sum_{i=1}^{k} \text{dom}(C_i).$$

Efficient algorithms exist for special classes of graphs; for example trees and interval graphs. The domination number in these cases can be found in linear time [33].

Proposition 6.3.15. *For any graph G, $dom(G \setminus e) - 1 \leq dom(G) \leq dom(G \setminus e)$.*

Proof. An edge in a graph joins two vertices, so adding an edge can decrease the domination number at most by 1, hence $\text{dom}(G \setminus e) - 1 \leq \text{dom}(G)$. Similarly, removing an edge will not decrease the domination number, so $\text{dom}(G) \leq \text{dom}(G \setminus e)$. □

Lemma 6.3.16. *For any connected non-tree graph G there exists an edge $e \in E(G)$, such that $G \setminus e$ is connected and $dom(G \setminus e) = dom(G)$.*

Proof. Suppose G is a non-tree connected graph and D_G is a minimum dominating set of G. There is an edge $e = (u, v) \in E(G)$ such that $G \setminus e$ is connected, since G is a non-tree graph. One of the following cases holds.

1. If $u, v \in D_G$, then D_G is also a minimum dominating set for $D_{G \setminus e}$, as presence or absence of e has no effect on $\text{dom}(G \setminus e)$. Therefore, $\text{dom}(G) = \text{dom}(G \setminus e)$.

2. If $u, v \in V(G) \setminus D_G$, then $\text{dom}(G) = \text{dom}(G \setminus e)$, as removing e has no effect on $\text{dom}(G)$.

3. Now suppose $u \in D_G$ and $v \notin D_G$. The degree of v is at least 2, else $G \setminus e$ is disconnected. Therefore there is another vertex w adjacent to v. If $w \in V(G) \setminus D_G$ then removal of the edge (w, v) has no effect on $\text{dom}(G)$, by Case 2. Otherwise, removing the edge e has no impact on $\text{dom}(G)$, since v is still dominated by w.

□

Fact 6.3.17. *For any connected graph $G \leq K_3$, the domination number $dom(G) = 1$.*

Theorem 6.3.18. *For any non-tree connected graph G, the domination number $dom(G) = \min\{dom(G \setminus e) : e \in E(G) \wedge G \setminus e \text{ is connected}\}$.*

Proof. Suppose G is a non-tree connected graph and D_G be a minimum dominating set of G. By Proposition 6.3.15, removing an edge from G will not decrease the domination

number of the resulting graph. Therefore for all $e \in E(G), \text{dom}(G \setminus e) \geq \text{dom}(G)$. From Lemma 6.3.16 we know that there exists an $e \in E(G)$ such that $\text{dom}(G \setminus e) = \text{dom}(G)$. Therefore $\text{dom}(G) = \min\{\text{dom}(G \setminus e) : e \in E(G) \wedge G \setminus e \text{ is connected}\}$. □

OLGA implementation for domination number

Domination number was a part of Barnes's version of OLGA [10]. In OLGA we implement domination number using Theorem 6.3.18, which is a slight improvement on the recursion used by Barnes. We verified our results using a brute force exact algorithm.

6.4 Graph colouring

The study of graph colourings has historically been linked to the four colour theorem [42]. We will discuss chromatic number and chromatic index in this section.

6.4.1 Chromatic index

Finding the chromatic index is known to be NP-hard [55]. Chromatic index is heavily used in scheduling problems [76], routing problems [30].

We derive recursive bounds on chromatic index. Using these bounds we compute the chromatic index of a graph. We use some of the existing results to give a recursive bound on the chromatic index.

Fact 6.4.1. *If H is a subgraph of G, then $\chi'(H) \leq \chi'(G)$.*

Fact 6.4.2. *For a disconnected graph with components $C_1, C_2, \ldots, C_k$, the chromatic index is given by the maximum chromatic index among all the components:*

$$\chi'(G) = \max\{\chi'(C_1), \chi'(C_2), \ldots, \chi'(C_k)\}.$$

Theorem 6.4.3 (Vizing [42])**.** *For any simple graph G, $\chi'(G)$ is either $\Delta(G)$ or $\Delta(G)+1$.*

Theorem 6.4.4. *Let G be a connected graph. Let $v \in V(G)$ be such that $\Delta(G) = \Delta(G-v)$, then*

$$\chi'(G-v) \leq \chi'(G) \leq \chi'(G-v)+1$$

Proof. By Theorem 6.4.3, $\Delta(G) \leq \chi'(G) \leq \Delta(G)+1$. Since $\Delta(G) = \Delta(G-v)$, so $\Delta(G) \leq \chi'(G-v) \leq \Delta(G)+1$. We have the following cases:

- If $\chi'(G) = \Delta(G)$ and $\chi'(G-v) = \Delta(G)$ then $\chi'(G) = \chi'(G-v)$.
- If $\chi'(G) = \Delta(G)+1$ and $\chi'(G-v) = \Delta(G)$ then $\chi'(G-v) < \chi'(G-v)+1 = \chi'(G)$.
- If $\chi'(G) = \Delta(G)+1$ and $\chi'(G-v) = \Delta(G)+1$ then $\chi'(G) = \chi'(G-v)$.

It is impossible for $\chi'(G) = \Delta(G)$ and $\chi'(G-v) = \Delta(G)+1$, due to Fact 6.4.1. Hence $\chi'(G-v) \leq \chi'(G) \leq \chi'(G-v)+1$.

□

Number of vertices	Unmatched Cases	
	Number	Percentage
3	0	100.00
4	2	33.33
5	7	33.33
6	42	37.50
7	206	24.15
8	2041	18.35
9	35882	13.74
10	1310394	11.18

Table 6.2: Chromatic index bound unmatched cases.

Theorem 6.4.5. *Let G be a connected graph such that G has a unique maximum degree vertex w. Let $v \in N(w)$. Then*

$$\chi'(G-v) \leq \chi'(G) \leq \chi'(G-v)+2$$

Proof. We know from Theorem 6.4.3, $\Delta(G) \leq \chi'(G) \leq \Delta(G)+1$. Removing the vertex v from G will reduce the maximum degree of G by 1, as G is a simple graph and $v \in N(w)$, by uniqueness of W. So the maximum degree of $G-v$ is $\Delta(G)-1$. By Theorem 6.4.3, $\Delta(G)-1 \leq \chi'(G-v) \leq (\Delta(G)-1)+1 = \Delta(G)$. So we have the following cases:

- If $\chi'(G) = \Delta(G)$ and $\chi'(G-v) = \Delta(G)$ then $\chi'(G) = \chi'(G-v)$.
- If $\chi'(G) = \Delta(G)$ and $\chi'(G-v) = \Delta(G)-1$ then $\chi'(G) = \chi'(G-v)+1$.
- If $\chi'(G) = \Delta(G)+1$ and $\chi'(G-v) = \Delta(G)$ then $\chi'(G) = \chi'(G-v)+1$.
- If $\chi'(G) = \Delta(G)+1$ and $\chi'(G-v) = \Delta(G)-1$ then $\chi'(G) = \chi'(G-v)+2$.

We cannot have any other inequalities between $\chi'(G)$ and $\chi'(G-v)$, due to Fact 6.4.1. Hence $\chi'(G-v) \leq \chi'(G) \leq \chi'(G-v)+2$. □

Theorems 6.4.4 and 6.4.5 exhibit that a simple algorithm can be devised using recursive bounds to compute bounds on the chromatic index of a graph. The bounds for chromatic index computed by this approach may not coincide but should still give some insights.

We computed bounds on the chromatic index of a graph using the recurrence defined in Theorems 6.4.4 and 6.4.5. If the minimum upper bound and maximum lower bound for the graph (obtained from Theorem 6.4.4 and 6.4.5) coincide we report the value as the chromatic index. If the bounds do not match we choose the minimum upper bound and validate it through Vizing's Theorem 6.4.3. We report such cases as unmatched cases. The number of unmatched cases are reported in Table 6.2.

Data analysis

There are two questions arise from the data reported in Table 6.2:

- What is the closest lower and upper bounds that can be established for the proportion of unmatched cases, for sufficiently large n?
- Does the proportion of unmatched cases converge to a limit?

OLGA implementation for chromatic index

We use the recursions in Theorems 6.4.4 and 6.4.5 to compute chromatic index. We verified the correctness of our data by comparing it with the output of an exact exponential algorithm.

6.4.2 Chromatic number

Finding the chromatic number is known to be NP-hard [55]. The chromatic number is heavily used in scheduling problems [76].

Theorem 6.4.6 (Brooks' Theorem 1941)**.** *Every connected graph G with maximum degree Δ is Δ-colourable, unless G is an odd cycle or a complete graph.*

Recursion on chromatic number

In this section we give recursive bounds on chromatic numbers and prove them for completeness.

Lemma 6.4.7. *For all $e \in E(G)$, $\chi(G \setminus e) \leq \chi(G)$.*

Proof. Removing an edge will not increase the chromatic number of a graph. □

Lemma 6.4.8. *For all $u, v \in V(G)$ such that $uv \notin E(G), \chi(G) \leq \chi(G/uv)$.*

Proof. Let G' be the graph obtained from G by identifying u and v. Let w be the new vertex in G' obtained by identifying u and v. Let f' be a colouring of G'. Let f be a colouring of G obtained by modifying the colouring f' as follows:

$$f(t) = \begin{cases} f'(t), & \text{if } t \notin \{u, v\}, \\ f'(w) & \text{otherwise.} \end{cases}$$

Clearly f is a colouring of G. This implies any minimal colouring of G' yields a colouring of G. Therefore, $\chi(G) \leq \chi(G/uv)$. □

Lemma 6.4.9. *For all $u, v \in V(G)$ such that $uv \notin E(G), \chi(G) \leq \chi(G + uv)$.*

Proof. Let G' be the graph obtained from G by adding the edge uv. If f' be a colouring of G' then f' is also a colouring of G. This implies any optimal colouring of G' yields a colouring of G. Therefore, $\chi(G) \leq \chi(G + uv)$. □

Corollary 6.4.10. *For any graph G, $\forall e \in E(G)$, and $\forall u, v \in V(G)$ such that $uv \notin E(G), \chi(G \setminus e) \leq \chi(G)$.*

Proposition 6.4.11. *For any graph G, $\forall e \in E(G)$, and $\forall u, v \in V(G)$ such that $uv \notin E(G), \chi(G \setminus e) \leq \chi(G) \leq \chi(G/uv)$.*

Proposition 6.4.12. *For a given graph G and $u, v \in V(G)$, $\chi(G) = \min\{\chi(G/uv), \chi(G+uv)\}$.*

Another way to find the chromatic number of a graph is through the chromatic polynomial. The *chromatic polynomial* $P(G, x)$ counts the number of x-colourings of a graph G [14]. The chromatic polynomial of a graph can be computed using the deletion-contraction method [14]; for any graph G, with $e \in E(G)$,

$$P(G, x) = P(G \setminus e, x) - P(G/e, x).$$

OLGA implementation for chromatic number

We used a brute force exact exponential-time algorithm to compute the chromatic number. The implementation of the recurrence to compute the chromatic number is not straightforward due to the following reason. Let G' be the new graph resulted from by adding a new edge to G while computing the chromatic number using the recursion in Proposition 6.4.12. Since G' has more edges than G the value of $\chi(G')$ is not yet computed (known). Therefore, we have to first compute the value of $\chi(G')$, using the recursion in Proposition 6.4.12 which will introduce another new graph with one more edges than G'. We have to continue this process of new edge addition till we reach the complete graph on $|V(G)|$ vertices. Then we have to backtrack computing the chromatic number of all the graphs generated in the process of reaching the complete graph.

6.4.3 Genus

The genus of a graph $\gamma(G)$ is the smallest $i \in \mathbb{N} \cup \{0\}$ such that G can be drawn on a sphere with i handles without edge crossings. For example, a planar graph has genus 0, as it can be drawn on a sphere. We denote an orientable surface obtained from sphere by attaching k handles to it by S_k. Examples of different surfaces are given Figure 6.1. In this section we will use Γ_k as an alternate to S_k.

Let $G(V, E)$ be a graph. Let F be the set of faces of G when embedded on a surface. Let girth(G) and circ(G) be the girth and circumference of G respectively.

Fact 6.4.13 (Euler's characteristic)**.** $|V| - |E| + |F| = 2 - 2\gamma(G)$.

Fact 6.4.14. *For an embedded graph G of order n*

(a) The number of edges in any face of $G \geq$ girth(G).

(b) The number of edges in any face of $G \leq$ circ(G).

(c) $\gamma(G) \leq \gamma(K_n)$.

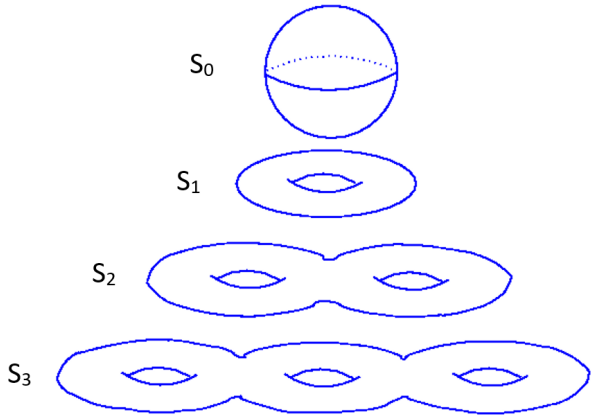


Figure 6.1: Sphere and Tori.

Non recursive bounds on genus

Most of the results presented in this section are elementary results on genus and can be found in [65].

Lemma 6.4.15.

$$\gamma(G) \geq 1 + \frac{|E|}{2}\Big(1 - \frac{2}{girth(G)}\Big) - \frac{|V|}{2}.$$

Proof. We know that

$$\sum_{f_i \in F} |f_i| = 2|E| \Rightarrow \text{girth}(G)|F| \leq 2|E| \Rightarrow |F| \leq \frac{2|E|}{\text{girth}(G)}.$$

Using this inequality and Fact 6.4.13 we get

$$\gamma(G) \geq 1 + \frac{|E|}{2}\Big(1 - \frac{2}{\text{girth}(G)}\Big) - \frac{|V|}{2}.$$

□

Lemma 6.4.16.

$$\gamma(G) \geq 1 + \frac{|E|}{2}\Big(1 - \frac{2}{circ(G)}\Big) - \frac{|V|}{2}.$$

Lemma 6.4.17.

$$\gamma(G) \geq \left\lceil \frac{1}{2}\Big(2 - \Big(|V| - \frac{|E|}{3}\Big)\Big) \right\rceil.$$

Proof. By Fact 6.4.14 the genus of a graph of order n is $\leq \gamma(K_n)$. The number of faces of K_n is $\leq \frac{2|E|}{3}$. So the number of faces of a graph of order n is $\leq \frac{2|E|}{3}$. Applying this to Fact 6.4.13 we get

$$|V| - |E| + \frac{2|E|}{3} \geq 2 - 2\gamma(G) \Rightarrow |V| - \frac{|E|}{3} \geq 2 - 2\gamma(G) \Rightarrow \gamma(G) \geq \left\lceil \frac{1}{2}\Big(2 - \Big(|V| - \frac{|E|}{3}\Big)\Big) \right\rceil.$$

Hence the claim. □

Recursive bounds on genus

Lemma 6.4.18.

$$\gamma(G) \leq \min\{\gamma(G \setminus e) : e \in E\} + 1.$$

Proof. Let $\gamma(G \setminus e) = k$ where $uv = e$. Let ϕ be an embedding of $G \setminus e$ on Γ_k. We can construct an embedding of G from the embedding of $G \setminus e$ as follows:

- Case 1: Suppose there exists a face f such that both u and v are incident with it. We can add an edge (u, v) in the interior of f without introducing any edge crossings. So $\gamma(G) = k$.
- Case 2: Suppose there is no face in G such that both u and v are incident to the same face. Let f_1 be the face containing u and f_2 be the face containing v. Suppose adding an edge (u, v) crosses other edges in $G \setminus e$. By adding an extra handle h from the interior of face f_1 to the interior of face f_2, we create a new surface Γ_{k+1}. We create a new embedding ϕ' by extending ϕ with the new edge e embedded on h. Thus there exists an embedding of G on Γ_{k+1}. So $\gamma(G) \leq k + 1$. Therefore the genus of the graph $\gamma(G)$ is at most the genus of the graph $\gamma(G \setminus e) + 1$. Hence

$$\gamma(G) \leq \min\{\gamma(G \setminus e) : e \in E\} + 1.$$

□

Lemma 6.4.19.

$$\gamma(G) \geq \max\{\gamma(G \setminus e) : e \in E\}.$$

Proof. Let $\gamma(G) = k$. Let ϕ be an embedding of G on Γ_k. Let $e \in E(G)$ be any edge removed from G. The embedding ϕ will still be an embedding of $G \setminus e$ on Γ_k, as removing an edge will not introduce new edge crossings in the embedding. Therefore $\gamma(G) \geq \gamma(G \setminus e)$. Hence $\max\{\gamma(G \setminus e) : e \in E\}$ is at most $\gamma(G)$. □

We computed bounds on the genus of a graph using the recurrence defined in Lemmas 6.4.18 and 6.4.19. If the minimum upper bound and maximum lower bound for the graph coincide we report the value as the genus. If the bounds do not match we choose the minimum upper bound. We report such cases as unmatched cases. The numbers of unmatched cases are reported in Table 6.3.

Data analysis

There are two questions arise from the data reported in Table 6.3:

- What are the sharpest lower and upper bounds that can be established for the proportion of unmatched cases, for sufficiently large n?
- Does the proportion of unmatched cases converge to a limit?

Number of vertices	Unmatched Cases	
	Number	Percentage
3	0	100.00
4	0	100.00
5	0	100.00
6	34	30.35
7	149	17.45
8	2648	23.81
9	115465	44.22
10	530010	45.23

Table 6.3: Genus bound unmatched cases.

OLGA implementation for genus

Genus was part of Sio's version of OLGA, where Sio used a different recursion to compute genus [134]. We implemented genus using the recurrences in Lemma 6.4.18 and Lemma 6.4.19 and we hard coded the genus of all graphs of order ≤ 5, based on Kuratowski's theorem [84]. We did not verify the correctness of genus data reported in OLGA. We applied different base cases for the recurrence used for genus and coincidentally the number of unmatched cases are always same. This makes recurrence of genus different than other recurrences used in OLGA.

Chapter 7

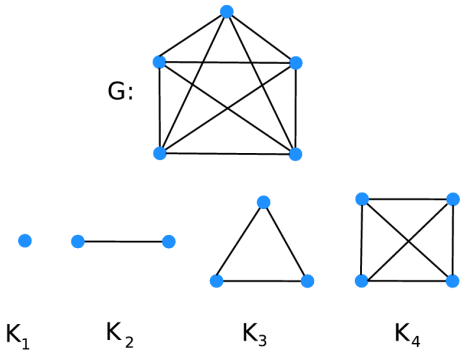
Most Frequent Connected Induced Subgraph (MFCIS)

In this chapter we introduce a new graph parameter as mentioned in Section 5.3. The initial motivation behind introducing this parameter was to help compare *recursive* computation and *exact* computation of parameters for OLGA as discussed in Section 5.3.1, however this parameter is also very interesting on its own right. We first discuss the MFCISs of special classes of graphs. Later in the chapter we prove that finding a MFCIS of a graph is #P-hard. We also give an infinite class of graphs whose order of the MFCIS contains a large portion of the order of the graph. We chracterised the most frequent induced path for those special class of graphs. We conclude the chapter with some interesting questions related to it.

7.1 Introduction

Enumerative combinatorics is an area of combinatorics that deals with the number of ways that certain patterns can be formed. Two examples of this type of problem are counting combinations and counting permutations [3]. The enumeration theorems by Pólya [109], Cayley [31] and Redfield [66] are widely used for combinatorial enumeration problems.

Given two graphs, determining whether they are identical, determining whether one input graph is a part of the other input graph, finding a maximum common part of the two input graphs and enumerating common substructures of different graphs are some questions of interest in a wide range of fields of study, like computer networks, bioinformatics and chemoinformatics [48, 129]. Some variations of finding the common substructure between two different graphs include the maximum common induced subgraph, the maximum common partial subgraph, the maximum common connected subgraph [37, 115, 150]. One of the most extensively studied problems is the maximum common induced subgraph problem (MCIS) [37, 150]. Finding a MCIS is NP-hard [55]. Finding a MCIS is about finding the largest *size* of common subgraph of two graphs whereas finding a Most Frequent Connected Induced Subgraph (MFCIS) is about finding a subgraph with largest *frequency* in a single graph. Formally, we define the MFCIS as:



Subgraph	Frequency in G
K_1	5
K_2	10
K_3	10
K_4	5
G	1

Figure 7.1: K_5 and its connected induced subgraphs.

Definition 7.1.1. *Let G be a connected simple graph. Let H be a connected induced subgraph of G. Let $freq_G(H) = |\{H'|H'$ is an induced connected subgraph of G and $H' \cong H\}|$. The graph H is a most frequent connected induced subgraph (MFCIS) of a graph G, if $freq_G H$ is the maximum among all connected induced subgraphs of G.*

For example in Figure 7.1 we show the MFCISs in red.

7.2 MFCIS of special classes of Graphs

In this section, we will characterise the MFCISs of some special class of graphs.

Lemma 7.2.1. *Let P_n be a path with n edges, then K_1 is the MFCIS of P_n.*

Proof. The induced subgraphs of P_n are P_k, $k \leq n$. It is clear that $\text{freq}_{P_n}(P_k) = n-k+1$. The value of $\text{freq}_{P_n}(P_k)$ is maximum when $k = 0$. □

Lemma 7.2.2. *Let C_n be a cycle with n edges, then for $k \leq n-2, P_k$ are the MFCISs of C_n.*

Proof. The induced subgraphs of C_n are P_k, $k \leq n-2$ or C_n. It is clear that $\text{freq}_{C_n}(P_k) = n-k+k = n$ and $\text{freq}_{C_n}(C_n) = 1$. Therefore, P_k are the MFCISs of C_n. □

Note: C_n is the only class of graphs for which every non-trivial induced subgraph (all induced subgraphs except the graph itself and $\emptyset$) is a MFCIS.

In order to obtain our results on some more special classes of graphs, we use the following binomial distribution property.

Fact 7.2.3. *For given n, the value of $\binom{n}{k}$ is maximum when*

$$k = \begin{cases} \lceil \frac{n}{2} \rceil or \lfloor \frac{n}{2} \rfloor & \text{if } n \text{ is odd,} \\ \frac{n}{2} & \text{otherwise.} \end{cases}$$

Note that the second largest value of $\binom{n}{k}$ occurs when $k = \lfloor \frac{n}{2} \rfloor - 1$.

Lemma 7.2.4. *For $n > 2$, $\binom{n}{\frac{n}{2}} / \binom{n}{\frac{n}{2}-1} = 1 + \frac{2}{n}$.*

Proof. $\frac{n!}{\frac{n}{2}!\frac{n}{2}!} \times \frac{((\frac{n}{2}+1))!(\frac{n}{2}-1)!}{n!} = \frac{\frac{n}{2}+1}{\frac{n}{2}} = 1 + \frac{2}{n}$ □

Fact 7.2.5. *If $n > 2$ is an even number, then $\binom{n}{\frac{n}{2}}^2 < 2\binom{n}{\frac{n}{2}}\binom{n}{\frac{n}{2}+1}$.*

Proof. Using Lemma 7.2.4, $\frac{\binom{n}{\frac{n}{2}}^2}{2\binom{n}{\frac{n}{2}+1}\cdot\binom{n}{\frac{n}{2}}} = \frac{\binom{n}{\frac{n}{2}}}{2\binom{n}{\frac{n}{2}+1}} = \frac{\frac{n}{2}+1}{n} = \frac{1}{2} + \frac{1}{n} < 1, \forall n > 2.$ □

Similarly, if n is an odd number, then $\binom{n}{\lceil \frac{n}{2} \rceil}^2 < 2\binom{n}{\lceil \frac{n}{2} \rceil}\binom{n}{\lfloor \frac{n}{2} \rfloor}$.

Lemma 7.2.6. *K_t is the MFCIS of K_{2t}.*

Proof. Any induced subgraph of a complete graph is also a complete graph. We know that $\binom{2m}{y}$ is maximum when $y = 2m/2 = m$. Here $\text{freq}_{K_t}(K_{2t}) = \binom{2t}{t}$. So $\text{freq}_{K_t}(K_{2t})$ is maximum among all induced subgraphs of K_{2t}. Therefore, K_t is the MFCIS of K_{2t}. □

Corollary 7.2.7. *K_t and K_{t+1} are the MFCISs of K_{2t+1}.*

Any induced subgraph of a complete bipartite graph $K_{m,n}$ is a complete bipartite graph because the induced subgraph contains all possible edges between vertices across the bipartition of the induced subgraph, that are present in $K_{m,n}$. Let $K_{a,b}$ be an induced subgraph of $K_{m,n}$. Let $a, b < m, n$.

- If $a = b$, $\text{freq}_{K_{a,b}}\ K_{m,n}$ = (Number of ways to choose a vertices from m vertices)(Number of ways to choose b vertices from n vertices) = $\binom{m}{a}\binom{n}{b}$.

- If $a \neq b$, $\text{freq}_{K_{a,b}} K_{m,n}$ = (Number of ways to choose a vertices from m vertices)(Number of ways to choose b vertices from n vertices) + (Number of ways to choose b vertices from m vertices)(Number of ways to choose a vertices from n vertices) = $\binom{m}{a}\binom{n}{b} + \binom{m}{b}\binom{n}{a}$.

Lemma 7.2.8. *The MFCIS of $K_{n,n}$ is*

$$\begin{cases} K_{\lceil \frac{n}{2} \rceil, \lfloor \frac{n}{2} \rfloor} & \text{if } n \text{ is odd,} \\ K_{\frac{n}{2}, \frac{n}{2}+1} & \text{otherwise.} \end{cases}$$

The induced subgraph $K_{a,b}$ of $K_{m,n}$ is a MFCIS when the values a, b ($a \neq b$) are selected to maximise the expression $\binom{m}{a}\binom{n}{b} + \binom{m}{b}\binom{n}{a}$.

We computed the values of a, b ($a \neq b$) which maximises $\binom{m}{a}\binom{n}{b} + \binom{m}{b}\binom{n}{a}$, for all combinations of values of m and n such that $m \leq 25$ and $n \leq 25$. The following question arise from the computational result.

Problem

Given $m, n \in \mathbb{N}$, find a, b that maximises the value of $\binom{m}{a}\binom{n}{b} + \binom{m}{b}\binom{n}{a}$ such that $1 \leq a \leq m, 1 \leq b \leq n$ and $a \neq b$.

7.3 Finding a MFCIS is #P-hard

We first define the problems that we need to prove the #P-hardness of finding a MFCIS.

#$\lfloor \frac{n}{2} \rfloor$-CLIQUE:
Input: A graph G of order n.
Output: Number of complete subgraphs of order $\lfloor \frac{n}{2} \rfloor$ in G.

The **#k-CLIQUE** is defined as:

#k-CLIQUE:
Input: A graph G and an integer $k > 0$.
Output: Number of complete subgraphs of order k in G.

Finding a MFCIS is defined as:

MFCIS:
Input: A graph G.
Output: An induced subgraph H of G such that $\text{freq}_G(H)$ is the maximum among all connected induced subgraphs of G.

Theorem 7.3.1. *#$\lfloor \frac{n}{2} \rfloor$-CLIQUE is #P-complete.*

Proof. We will prove #k-CLIQUE $\propto$ #$\lfloor n/2 \rfloor$-CLIQUE. Let M be an oracle that solves #$\lfloor n/2 \rfloor$-CLIQUE, returning the number $M(G)$ of $\lfloor n/2 \rfloor$-cliques in G. Given any graph G and an integer k such that $0 < k \leq n$, we will show how to compute the numbers of k-cliques of G in polynomial time using M. Note that the size of largest clique in G is at most n, therefore $k \leq n$. If $n \leq 2$ or $k \leq 2$ the numbers of k-cliques of G can be counted by direct computation. Otherwise, consider the following cases.

- $1 < \lfloor n/2 \rfloor < k$: Let us construct G' by adding $2k-n$ isolated vertices to G. Note that the number of k-cliques in G and G' are the same. Construction of G' takes linear time as $k \leq n$, so the number of isolated vertices added is $\leq n$. By construction, $M(G')$ = the number of $\lfloor 2k-n+n \rfloor/2$-cliques in G', which is same as the number of k-cliques in G, as adding isolated vertices does not increase the number of k-cliques for $k > 1$.

- $2 < k \leq \lfloor n/2 \rfloor$: If $k = \lfloor n/2 \rfloor$, then $M(G)$ gives the desired answer. So assume $k < \lfloor n/2 \rfloor$. For each i such that $1 \leq i \leq n - 2k - 1$, let $G_i = G + K_i$ and $n_i = |V(G_i)| = n + i$. By construction, $G_{i+1} = G_i + K_1$. Therefore, each clique of size t in G_i yields a clique of size $t + 1$ in G_{i+1} as the new vertex added to G_i is adjacent to every other vertex of G_i. For each G_i, note that for all j such that $0 \leq j \leq i$, each $(\lfloor n_i/2 \rfloor - j)$-clique in G corresponds to a $\lfloor n_i/2 \rfloor$-clique in G_i. Let x_t denote the number of cliques of size t in G. Therefore,

$$\text{number of } \lfloor n_i/2 \rfloor \text{ cliques in } G_i = \sum_{j=0}^{i} \binom{i}{j} x_{\lfloor n_i/2 \rfloor - j} \tag{7.1}$$

Let A be the matrix formed by the coefficients of x_t from (7.1), i.e.,

$$
\begin{array}{c}
\\
Graphs \\
G \\
G_1 \\
G_2 \\
\vdots \\
G_{n-2k-1}
\end{array}
\begin{array}{c}
\text{Number of cliques of size } \ldots \\
\begin{array}{ccccccccc}
k & k+1 & \ldots & \lfloor n/2 \rfloor - 1 & \lfloor n/2 \rfloor & \lfloor n/2 \rfloor + 1 & \ldots & n-k-1 \\
\end{array} \\
\begin{pmatrix}
0 & 0 & \ldots & 0 & 1 & 0 & \ldots & 0 \\
0 & 0 & \ldots & \binom{1}{0} & \binom{1}{1} & 0 & \ldots & 0 \\
0 & 0 & \ldots & \binom{2}{0} & \binom{2}{1} & \binom{2}{2} & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
\binom{n-2k-1}{0} & \binom{n-2k-1}{1} & \ldots & \binom{n-2k-1}{\lfloor n/2 \rfloor - k - 1} & \binom{n-2k-1}{\lfloor n/2 \rfloor - k} & \binom{n-2k-1}{\lfloor n/2 \rfloor - k + 1} & \ldots & 1
\end{pmatrix}
\end{array}.
$$

Note that if n is odd, the entries of the matrix A will be different. Let X be the vector formed by the x_t and B be the vector containing the output $M(G_i)$, i.e.,

$$
X = \begin{pmatrix} x_k \\ x_{k+1} \\ \vdots \\ \vdots \\ x_{n-k} \end{pmatrix}, \quad B = \begin{pmatrix} M(G_1) \\ M(G_2) \\ \vdots \\ \vdots \\ M(G_{n-2k}) \end{pmatrix}.
$$

Clearly, A, X and B satisfy the equation

$$
AX = B. \tag{7.2}
$$

We will compute the value of x_k using Algorithm 3.

Algorithm 3: Count the number of k-cliques in graph G

```
/* ************************************************
   Input:  A graph G of order n and 2 < k ≤ ⌊n/2⌋.
   Output:  The number of k-cliques in G.
   Oracle:  M, returns the number of ⌊n/2⌋-cliques in G.
      ************************************************ */
1  def k-cliqueEnumerator(G):
2  i = 1
3  while i ≤ n − 2k − 1 do
4  |  i = i + 1
5  |  G_i = G + K_i
6  |  compute M(G_i)
7  end
8  Set up equations AX = B, as defined in (7.2)
9  X = A^{-1}B
10 return value of x_k
```

The construction of each G_i takes polynomial time, as we add $n-2k-1$ vertices and a polynomially bounded number of edges. There are $n-2k$ calls made to the oracle and each call takes $O(1)$ time. The matrix A is a triangular matrix (after rearrangement of columns) with non-zero diagonal entries. Therefore, A is invertible. The value of X is computed using Step 11 of Algorithm 3, which takes polynomial time. Therefore, the value of each x_t, in particular x_k can be computed in polynomial time.

The computation of the number of k-cliques takes polynomial time in both the cases, therefore $\#\lfloor n/2 \rfloor$-CLIQUE is #P-hard. □

Fact 7.3.2. *If H is a MFCIS of a simple graph G of order n, then $\text{freq}_G(H) \leq \binom{n}{\lceil n/2 \rceil}$.*

Fact 7.3.2 holds as $\text{freq}_G(H)$ is at most $\binom{n}{k}$ where $k = |V(H)|$.

Lemma 7.3.3. *For $n > 0$, $\frac{\binom{n}{\lfloor \frac{n}{2} \rfloor}}{\binom{n+1}{\lfloor \frac{n+1}{2} \rfloor}} \geq 1/2$.*

Proof. Consider the following cases.

- $n = 2m$: $\frac{\binom{n}{\lfloor \frac{n}{2} \rfloor}}{\binom{n+1}{\lfloor \frac{n+1}{2} \rfloor}} = \frac{\binom{2m}{m}}{\binom{2m+1}{\lfloor \frac{2m+1}{2} \rfloor}} = \frac{2m!}{m!} \times \frac{m!(m+1)!}{(2m+1)!} = \frac{m+1}{2m+1} > 1/2.$
- $n = 2m+1$: $\frac{\binom{n}{\lfloor \frac{n}{2} \rfloor}}{\binom{n+1}{\lfloor \frac{n+1}{2} \rfloor}} = \frac{\binom{2m+1}{\lfloor \frac{2m+1}{2} \rfloor}}{\binom{2m+2}{m+1}} = \frac{\binom{2m+1}{m}}{\binom{2m+2}{m+1}} = \frac{(2m+1)!}{(m+1)!m!} \times \frac{(m+1)!^2}{(2m+2)!} = \frac{m+1}{2m+2} = 1/2.$

□

Lemma 7.3.4. *For $n > 0$, if $\binom{t}{\lfloor \frac{t}{2} \rfloor} > n$ and $\binom{t-1}{\lfloor \frac{t-1}{2} \rfloor} < n$, then $\binom{t-1}{\lfloor \frac{t-1}{2} \rfloor} > n/2$.*

Proof. If $\binom{t}{\lfloor \frac{t}{2} \rfloor} > n$, then $\frac{1}{2}\binom{t}{\lfloor \frac{t}{2} \rfloor} > n/2$. By Lemma 7.3.3, $\binom{t-1}{\lfloor \frac{t-1}{2} \rfloor} \geq \frac{1}{2}\binom{t}{\lfloor \frac{t}{2} \rfloor} > n/2$. □

Lemma 7.3.5. *Given $k > 0$ and $t - k \geq 1$, $\binom{t-1}{k}/\binom{t}{k} \geq 1/t$.*

Proof. $\binom{t-1}{k}/\binom{t}{k} = \frac{(t-1)!}{k!(t-1-k)!} \times \frac{k!(t-k)!}{t!} = \frac{t-k}{t} \geq 1/t,$ for $t - k \geq 1$. □

Note that if $t \geq 2k$, then $\binom{t-1}{k}/\binom{t}{k} \geq 1/2$.

$\#\lfloor \frac{n}{2} \rfloor$-CLIQUE $\propto$ MFCIS

In order to prove finding a MFCIS is #P-hard we first define the following terms. Let M be an oracle Turing machine such that $M(G)$ outputs a MFCIS of G. We assume that in the case of multiple MFCISs, M will output the smaller clique (if one exists as a MFCIS), and otherwise will output a MFCIS of smaller order of the graph. We define a *loopy clique* $K_t^{\odot}$ to be a clique K_t where every vertex has a self loop. The oracle M will output a simple graph as the MFCIS, if it has to choose between a simple graph and a multigraph. We use $\oplus$ to denote a disjoint union of graphs.

If $n \leq 6$, we find the MFCIS by direct computation. Otherwise we reduce from $\#\lfloor \frac{n}{2} \rfloor$-CLIQUE to finding a MFCIS in two steps. In the first step we compute the frequency of the MFCIS and in the second step we compute the number of $\#\lfloor \frac{n}{2} \rfloor$-cliques .

Given any graph G of order n we will compute the frequency of the MFCIS(G) using M. Let H be a MFCIS of G and $f = \text{freq}_G(H)$. If G is a complete graph then by Lemma 7.2.6 and Fact 7.2.3, the frequency is $\binom{n}{\lfloor n/2 \rfloor}$. Otherwise, we find the smallest order $K_t^\odot$ such that $M(G \oplus K_t^\odot)$ returns a loopy clique. Such $K_t^\odot$ exists as by Fact 7.3.2, $f \leq \binom{t}{\lfloor t/2 \rfloor} = \text{freq}_{K_t^\odot}(K^\odot_{\lfloor t/2 \rfloor})$ for sufficiently large t.

Observation 7.3.6. *If G is a simple graph and H is a MFCIS of G, then for any loopy clique $K_t^\odot$, we have $freq_G(H) = freq_{G \oplus K_t^\odot}(H)$.*

Fact 7.3.7. *For any j such that $1 \leq j \leq i$, we have $freq_{K_i^\odot}(K_j^\odot) = \binom{i}{j}$.*

Let p be the smallest integer such that $M(G \oplus K_t^\odot) = K_p^\odot$. Then the following hold:

- $\binom{t}{\lfloor t/2 \rfloor} > f$, as $\text{freq}_{K_t^\odot}(K_p^\odot) > f$.
- $\binom{t-1}{\lfloor \frac{t-1}{2} \rfloor} < f$.

By Fact 7.3.4,

$$\binom{t-1}{\lfloor \frac{t-1}{2} \rfloor} > f/2. \tag{7.3}$$

We first add the loopy clique $K_{t-1}^\odot$ to G, i.e., $G := G \oplus K_{t-1}^\odot$. Note that we use G to denote the current graph, including loopy cliques that have already been added to it. By Observation 7.3.6, a MFCIS($G \oplus K_{t-1}^\odot$) is H and the $\text{freq}_{G \oplus K_{t-1}^\odot}(H)$ is still f.

Note that $f > n$. Otherwise K_1 is a MFCIS as the oracle choses the graph with lower order as the MFCIS when it has to make a choice between multiple graphs with the same frequency as the MFCIS. Also, for $n > 6$, since we do direct computation. Therefore $f \geq 8$.

After adding i loopy cliques to G, let p_i denotes the order of the loopy clique which has the highest frequency in G and d_i denotes the difference between f and the frequency of $K_{p_i}^\odot$ in G.

At present $G = G \oplus K_{t-1}^\odot$ still. Remember G contains a $K_{t-1}^\odot$ component, so by Fact 7.2.3, the loopy clique with maximum frequency in G is $K^\odot_{\lfloor \frac{t-1}{2} \rfloor}$. Therefore $p_1 = \lfloor \frac{t-1}{2} \rfloor$ and $d_1 = f - \binom{t-1}{p_1}$. Therefore, by (7.3),

$$d_1 \leq f(1 - \frac{1}{2}). \tag{7.4}$$

To find the exact value of f we iteratively add more loopy cliques by following the following process.

We find the smallest number t_1 such that $\binom{t_1}{p_1} > d_1$. Note that now, G is $G \oplus K_{t-1}^\odot$. By Observation 7.3.6, $M(G \oplus K_{t_1}^\odot) = K_{p_1}^\odot$. Therefore $\binom{t_1-1}{p_1} \leq d_1$. By Lemma 7.3.5 and the definition of t_1, we have

$$\binom{t_1 - 1}{p_1} > \frac{1}{t}\binom{t_1}{p_1} > \frac{d_1}{t_1}. \tag{7.5}$$

Now we add $K^{\odot}_{t_1-1}$ to G. So G now contains two components that are loopy cliques, $K^{\odot}_{t-1}$ and $K^{\odot}_{t_1-1}$, therefore,

$$\begin{aligned} d_2 &= d_1 - \binom{t_1-1}{p_1} \\ &\leq d_1(1-\frac{1}{t_1}) && \text{by (7.4)} \\ &\leq f(1-\frac{1}{2})(1-\frac{1}{t_1}) && \text{by (7.3).} \end{aligned} \tag{7.6}$$

Observation 7.3.8. $t \geq t_1$.

Proof. Remember before adding $K^{\odot}_{t-1}$ to G it had no loopy cliques, whereas after adding $K^{\odot}_{t-1}$ to G it has $\binom{t-1}{i}$ loopy cliques $K^{\odot}_i$, $i \leq t-1$. Since t_1 is the smallest number satisfying $M(G \oplus K^{\odot}_{t_1}) = K^{\odot}_{p_1}$, it follows that $t \geq t_1$. □

We keep repeating the above process of adding loopy clique components. After repeating the process r times, $d_r \leq f(1-\frac{1}{2})(1-\frac{1}{t_1})\cdots(1-\frac{1}{t_r})$. Also observe $d_1 > d_2 > \cdots > d_r$, so that $d_r < n$ for sufficiently large n. Let us put $r = n^2$.

$$\begin{aligned} & f(1-1/2)(1-1/t_1)\cdots(1-1/t_r) \\ & < f(1-1/n)^{n^2} \\ & < f \cdot c \cdot e^{-n} && \text{for sufficiently large } n \text{ and some constant } c > 1 \\ & < n && \text{for sufficiently large } n. \end{aligned} \tag{7.7}$$

From (7.7) after $O(n^2)$ iterations $d_r < n$. So we add d_r components of $K^{\odot}_{p_r}$ to match the value of f.

We use Algorithm 4 to compute f.

Adding a loopy clique takes polynomial time, as we add at most n vertices and $\binom{n}{2}$ edges and n loops. We have shown that by adding $O(n^2)$ components of loopy cliques we find the value of f. A loopy clique gets added to G after $O(n)$ oracle calls. There are $O(n(O(n^2))) + O(n)$ calls to the oracle and each call takes $O(1)$ time. The value of f from the loopy cliques can be computed in $O(1)$ time. Therefore the value of f can be computed in polynomial time.

Now we will use f to compute the the number of $\lfloor n/2 \rfloor$-cliques in G. If G is a complete graph we return $\binom{n}{\lfloor n/2 \rfloor}$. Otherwise we add (disjoint union) K_n to G. By Fact 7.3.2, $f \leq \binom{n}{\lfloor n/2 \rfloor}$, so adding multiple (say $c \geq 1$) K_n to G will make $K_{\lfloor n/2 \rfloor}$ the MFCIS of $G \oplus K_n \oplus K_n \cdots (c \text{ times}) \cdots \oplus K_n$. Using Algorithm 4, we compute $\text{freq}_{G \oplus K_n} K_{\lfloor n/2 \rfloor}$ and subtracting $c\binom{n}{\lfloor n/2 \rfloor}$ gives the desired result.

Algorithm 4: Find the frequency of MFCIS in G

```
/* ************************************************
   Input:  A graph G.
   Output:  The frequency of a MFCIS of G of order n.
   Oracle:  M, returns a MFCIS of the graph G.
   Notation:  ⊕ represents the disjoint union of graphs.
   Assumption:  n is fixed throughout the execution .
   ************************************************ */
1  LC = [ ] // LC is a list to store the orders of the loopy cliques
     added to G.
2  def frequencyFinder(G):
3  if G is a complete graph then
4  |  frequency = (n choose ⌊n/2⌋)
5  |  return frequency
6  for i = 1 to n do
7  |  if M(G ⊕ K_i^⊙) ≅ K_p^⊙, for some p then
8  |  |  if p == i then
9  |  |  |  frequency = Σ_{j=0}^{|LC|} (LC[j] choose p)
10 |  |  |  return frequency
11 |  |  else
12 |  |  |  LC·append(i − 1)
13 |  |  |  G = G ⊕ K_{i−1}^⊙
14 |  |  |  return frequencyFinder(G)
15 |  |  end
16 end
```

We use Algorithm 5 to compute the number of $\lfloor n/2 \rfloor-$cliques in G.

Algorithm 5: Count the number of $K_{\lfloor n/2 \rfloor}$ in graph G

```
/* ****************************************************
Input:  A graph G of order n.
Output:  The frequency of K_{⌊n/2⌋} in G.
Oracle:  M, returns the MFCIS of the graph G.
Notation:  ⊕ represents the disjoint union of graphs.
   ****************************************************      */
1  def cliqueEnumerator(G):
2  if G is a complete graph then
3  |  return (n choose ⌊n/2⌋)
4  i = 1
5  G = G ⊕ K_n
6  while M(G) ≇ K_{⌊n/2⌋} do
7  |  i = i + 1
8  |  G = G ⊕ K_n
9  end
10 freq = frequencyFinder(G)
11 return (freq − i ∗ (n choose ⌊n/2⌋))
```

Adding K_n to G takes polynomial time, as we add n vertices and $\binom{n}{2}$ edges. Suppose before adding any K_n to G, $\text{freq}_G(H) - \text{freq}_G(K_{\lfloor n/2 \rfloor}) = t$. By Observation 7.3.2, $\text{freq}_G(H) \leq \binom{n}{\lfloor n/2 \rfloor}$. By adding a K_n to G we are increasing the number of $K_{\lfloor n/2 \rfloor}$ in G by $\binom{n}{\lfloor n/2 \rfloor}$. Then by Lemma 7.2.4, after adding at most n copies of K_n to G, $K_{\lfloor n/2 \rfloor}$ will become the MFCIS of G. Note that, G now has n components of K_n. Algorithm 4 is called once, which takes polynomial time. There are $O(n)$ oracle calls and each call takes $O(1)$ time, therefore Algorithm 5 takes polynomial time.

Theorem 7.3.9. *The problem of finding a MFCIS of a graph is #P-hard.*

Proof. Let G be a graph. Algorithm 4 and Algorithm 5 compute the number of $K_{\lfloor n/2 \rfloor}$ in G using an oracle Turing machine M which finds a MFCIS of G. Since both Algorithm 4 and Algorithm 5 take polynomial time, the reduction from $\#\lfloor \frac{n}{2} \rfloor$-CLIQUE to MFCIS takes polynomial time. Hence computing a MFCIS is $\#P$-hard. □

We found by computation in all but a few cases the MFCISs of all non-clique graphs of up to 10 vertices is one of the following graphs: K_2, K_3, P_2, P_3. These graphs are of order at most 4, however we give an infinite class of graphs whose MFCISs contain approximately 60% of the vertices of the graph.

We support the following options in OLGA.

- Given a range of graphs, we generate the MFCISs for each. The current version of the code can generate the MFCISs of all graphs up to 12 vertices.
- Given a graph as input, it generates the MFCISs of the graph.

7.4 MFCIS of perfect binary tree

We analysed the MFCISs of graphs of up to 10 vertices. The set of graphs that appear as the MFCISs of non-clique graphs of order at most 10 are mostly P_2, P_3, K_2 and K_3. These graphs have order at most 4. This observation motivated us to investigate the question; do all the MFCISs have small order? In this section, we prove that this is not the case by presenting an infinite class of graphs whose order of the MFCISs contains a large portion of the order of the graph.

7.4.1 Definition and notation

A *binary tree* is a tree with a designated *root* node where each node has at most two children. The subtree rooted at the left (right) child of the root is called the *left (right) subtree*.

The *level* of a node v in a binary tree is the length of the path from the root node to v. The level of the root node is 0. The *height* of a binary tree is the maximum level among all the nodes in the tree. We say a node is *full* if it has a non-empty left child and a non-empty right child.

A *perfect binary tree* T_N of height $N \geq 0$ with root r is a binary tree where

- all leaf nodes have the same level, N and
- all other nodes are full.

The perfect binary tree T_N has order $2^{N+1} - 1$. We use T_N^L and T_N^R to denote the left and right subtree of T_N respectively. We use T_N^U to represent the subtree of T_N induced by all vertices from level 0 up to level $N - 1$.

Let S be a connected induced subgraph of T_N. We use $\mathscr{C}(T_i, S)$ to denote the set of connected induced subgraphs H of T_i satisfying the following conditions:

- $H \cong S$,
- $V(H)$ includes the root of T_i,
- $V(H)$ includes at least one leaf of T_i.

For example, Figure 7.2 shows $\mathscr{C}(T_2, T_1)$. In this example $H_1 \cong H_2 \cong H_3 \cong H_4 \cong J \cong T_1$, but $r \notin V(J)$, so $J \notin \mathscr{C}(T_2, T_1)$. Therefore $\mathscr{C}(T_2, T_1) = \{H_1, H_2, H_3, H_4\}$.

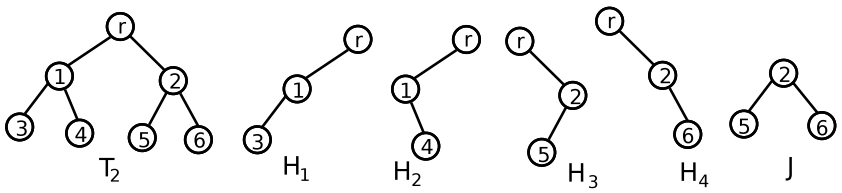


Figure 7.2: $\mathscr{C}(T_2, T_1) = \{H_1, H_2, H_3, H_4\}$ and $J \notin \mathscr{C}(T_2, T_1)$.

In Section 7.4.3, we will use $\mathscr{C}(T_i, S)$ to calculate the frequency of an induced subgraph in $T_N, N \geq i$.

7.4.2 Observations and facts

In this section, we list some facts and basic results, which we will use to count the frequency of a MFCIS in T_N.

Fact 7.4.1. $T_N^L \cong T_N^R \cong T_N^U \cong T_{N-1}$.

Fact 7.4.2. *Let r be the root of an induced subgraph S of T_N. If S^L and S^R denote the left and right subtrees of S respectively, then*

$$freq_{T_N}(S) = \begin{cases} 2\ freq_{T_N^L}(S^L) freq_{T_N^R}(S^R) & \text{if } S^L \ncong S^R, \\ freq_{T_N^L}(S^L) freq_{T_N^R}(S^R) & \text{otherwise.} \end{cases}$$

Observation 7.4.3. *If $S < T_k$, is not a perfect binary tree then $|\mathscr{C}(T_i, S)|$ is either ≥ 2 or 0 for $i \geq 1$.*

Proof. Since $S \subset T_k$, there exists a vertex $u \in V(T_k)$ that is not in $V(S)$. If $S \notin \mathscr{C}(T_i, S)$ then $|\mathscr{C}(T_i, S)| = 0$. Otherwise, there exists some $v \in V(S)$ whose left and right subtrees are not isomorphic, therefore $|\mathscr{C}(T_i, S)| \geq 2$. □

7.4.3 Properties of the MFCIS(T_N)

Lemma 7.4.4. *MFCIS$(T_N) \not\leq T_{N-1}, for N \geq 4$.*

Proof. On the contrary, let $S \leq T_{N-1}$ be a MFCIS of T_N. Let us consider the following cases.

- Case 1: $S \cong T_k$, where $k \in [1, N-1]$: Clearly

  $$\text{freq}_{T_N}(T_k) < \text{freq}_{T_N}(T_k \setminus v) = 2\ \text{freq}_{T_N}(T_k), \text{ for any leaf node } v \text{ of } T_k.$$

  Therefore, S is not a MFCIS of T_N.

- Case 2: $S \ncong T_k$, where $k \in [1, N-1]$: Let h be the smallest number such that $|\mathscr{C}(T_h, S)| > 0$. This implies the length of the longest path in S is at most $2h$. Therefore, the frequency of S in T_N is,

  $$\text{freq}_{T_N}(S) = \sum_{i=0}^{\min(N-h,h)} |\mathscr{C}(T_{h+i}, S)|\ (2^{N-(h+i)+1} - 1). \tag{7.8}$$

  The second term in (7.8) counts the number of occurrences of T_{h+i} in T_N.

  Let S' be a binary tree such that its left and right subtrees are isomorphic to S. By the construction of S', the smallest number for which $|\mathscr{C}(T_h + 1, S')| > 0$ is $h+1$. Therefore the length of the longest path in S' is at most $2h+2$. Consider the following cases:

– Case $(1 < 2h < N)$: The frequency of S' in T_N is

$$\begin{aligned}
\text{freq}_{T_N}(S') &= \sum_{i=1}^{h+2} |\mathscr{C}(T_{h+i}, S')| \ (2^{N-(h+i)+1} - 1) \\
&> \sum_{i=1}^{h+1} |\mathscr{C}(T_{h+i}, S')| \ (2^{N-(h+i)+1} - 1) \\
&> \sum_{i=1}^{h+1} (|\mathscr{C}(T_{h+i-1}, S)|)^2 \ (2^{N-(h+i)+1} - 1) \quad \text{using Fact 7.4.2.} \\
&> \sum_{i=1}^{h+1} 2(|\mathscr{C}(T_{h+i-1}, S)|) \ (2^{N-(h+i)+1} - 1) \quad \text{using Observation 7.4.3.} \\
&= \sum_{i=0}^{h} 2|\mathscr{C}(T_{h+i}, S)| \ (2^{N-(h+i)+1} - 1) \\
&> \sum_{i=0}^{h} |\mathscr{C}(T_{h+i}, S)| \ (2^{N-(h+i)+1} - 1) = \text{freq}_{T_N}(S). \qquad (7.9)
\end{aligned}$$

Hence S is not a MFCIS of T_N, which is a contradiction to our assumption.

– Case $(2h \geq N)$: The frequency of S in T_N is

$$\text{freq}_{T_N}(S) = \sum_{i=0}^{N-h} (|\mathscr{C}(T_{h+i}, S)| \ (2^{N-(h+i)+1} - 1). \qquad (7.10)$$

Let the term $|\mathscr{C}(T_{h+t-1}, S)| \ (2^{N-(h+t-1)+1} - 1)$ be the maximum among all the terms in (7.10). We rearrange the terms of (7.10) in the following way.

$$\begin{aligned}
\text{freq}_{T_N}(S) = \ & (\sum_{i=0}^{t-2} |\mathscr{C}(T_{h+i}, S)| \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_{h+t-1}, S)| \ (2^{N-(h+t-1)+1} - 1) \\
& + \ (\sum_{i=t}^{N-h-1} |\mathscr{C}(T_{h+i}, S)| \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_N, S)| \ (2^{N-N+1} - 1) \\
= \ & (\sum_{i=0}^{t-2} |\mathscr{C}(T_{h+i}, S)| \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_{h+t-1}, S)| \ (2^{N-(h+t)} - 1) \\
& + \ (\sum_{i=t}^{N-h-1} |\mathscr{C}(T_{h+i}, S)| \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_N, S)|. \quad (7.11)
\end{aligned}$$

The frequency of S' in T_N is

$$\begin{aligned}
\text{freq}_{T_N}(S') = & \sum_{i=1}^{N-h} (|\mathscr{C}(T_{h+i}, S')| \ (2^{N-(h+i)+1} - 1) \\
& \text{We are rearranging the terms using the same } t \text{ from (7.10)} \\
= & (\sum_{i=1}^{t-1} |\mathscr{C}(T_{h+i}, S')| \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_h + t, S')| \ (2^{N-(h+t)+1} - 1) \\
& + \ (\sum_{i=t+1}^{N-h} |\mathscr{C}(T_{h+i}, S')| \ (2^{N-(h+i)+1} - 1)) \\
\geq & (\sum_{i=1}^{t-1} |\mathscr{C}(T_{h+i-1}, S)|^2 \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_{h+t-1}, S)|^2 \ (2^{N-(h+t)+1} - 1) \\
& + \ (\sum_{i=t+1}^{N-h} |\mathscr{C}(T_{h+i-1}, S)|^2 \ (2^{N-(h+i)+1} - 1)) \quad \text{using Fact 7.4.2} \\
\geq & (\sum_{i=1}^{t-1} |\mathscr{C}(T_{h+i-1}, S)|^2 \ (2^{N-(h+i)+1} - 1)) + 2|\mathscr{C}(T_{h+t-1}, S)| \ (2^{N-(h+t)+1} - 1) \\
& + \ (\sum_{i=t+1}^{N-h} |\mathscr{C}(T_{h+i-1}, S)|^2 \ (2^{N-(h+i)+1} - 1)) \quad \text{using Observation 7.4.3} \\
= & (\sum_{i=1}^{t-1} |\mathscr{C}(T_{h+i-1}, S)|^2 \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_{h+t-1}, S)| \ (2^{N-(h+t)+1} - 1) \\
& + \ (\sum_{i=t+1}^{N-h} |\mathscr{C}(T_{h+i-1}, S)|^2 \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_{h+t-1}, S)| \ (2^{N-(h+t)+1} - 1) \\
> & (\sum_{i=0}^{t-2} |\mathscr{C}(T_{h+i}, S)| \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_{h+t-1}, S)| \ (2^{N-(h+t)} - 1) \\
& + \ (\sum_{i=t}^{N-h-1} |\mathscr{C}(T_{h+i}, S)| \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_{h+t-1}, S)| \ (2^{N-(h+t)} - 1)
\end{aligned}$$

$$\begin{aligned}
> & (\sum_{i=0}^{t-2} |\mathscr{C}(T_{h+i}, S)| \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_{h+t-1}, S)| \ (2^{N-(h+t)} - 1) \\
& + \ (\sum_{i=t}^{N-h-1} |\mathscr{C}(T_{h+i}, S)| \ (2^{N-(h+i)+1} - 1)) + |\mathscr{C}(T_N, S)| \\
= & \ \text{freq}_{T_N}(S). \quad \text{From 7.11} \qquad (7.12)
\end{aligned}$$

Hence $\text{freq}_{T_N}(S') < \text{freq}_{T_N}(S)$, which is a contradiction to our assumption.

Therefore, the MFCIS$(T_N) \not\leq T_{N-1}$ for $N \geq 4$. □

Corollary 7.4.5. *The MFCIS$(T_N) \not\leq T_{N-1} \cup \{r\}$, where r is the root of T_N.*

From Lemma 7.4.4 and Corollary 7.4.5 we obtain the following.

Fact 7.4.6. *If $S \subseteq T_N$ is a MFCIS of T_N then*

- *S contains root and some leaves of T_N, and*
- *S contains vertices from both T_N^L and T_N^R.*

Lemma 7.4.7. *Let $N \geq 4$. If S is an induced subgraph of T_N such that $S \cong MFCIS(T_N)$, then the graphs in $S \setminus r$ are the MFCISs of T_{N-1}.*

Proof. Let S_1 and S_2 be left and right subtrees of S respectively. From Corollary 7.4.5, $S \not\leq T_{N-1} \cup \{r\}$. Therefore without loss of generality, let $S_1 \leq T_N^L$ and $S_2 \leq T_N^R$.

From Fact 7.4.1, it is sufficient to show S_1 is a MFCIS of T_N^L and S_2 is a MFCIS of T_N^R to prove the lemma.

Using Fact 7.4.2, the frequency of S in T_N is

$$\text{freq}_{T_N}(S) = \begin{cases} \text{freq}_{T_N^L}(S_1) \cdot \text{freq}_{T_N^R}(S_2) & \text{if } S_1 \cong S_2, \\ 2 \cdot \text{freq}_{T_N^L}(S_1) \cdot \text{freq}_{T_N^R}(S_2) & \text{otherwise.} \end{cases}$$

We will prove this Lemma by contradiction. Suppose S_1 is not a MFCIS of T_N^L and S_2 is not a MFCIS of T_N^R. Therefore, there exist graphs $H \subseteq T_N^L$ and $J \subseteq T_N^R$ such that the $\text{freq}_{T_N^L}(H) > \text{freq}_{T_N^L}(S_1)$ and the $\text{freq}_{T_N^R}(J) > \text{freq}_{T_N^R}(S_2)$. Let B be a subtree of T_N such that $V(B) = V(H) \cup V(J) \cup \{r\}$ and $E(B) = E(H) \cup E(J) \cup \{ru\} \cup \{rv\}, u \in V(H), v \in V(J)$. The

$$\text{freq}_{T_N}(B) = \text{freq}_{T_N^L}(H)\text{freq}_{T_N^R}(J) > \text{freq}_{T_N}(S).$$

Therefore S is not a MFCIS of T_N, which is a contradiction. □

From Lemma 7.4.7, we observe the following fact.

Fact 7.4.8. *If $H \subseteq T_N$ with left and right subtrees as H^L and H^R respectively, then H^L is a MFCIS of T_N^L and H^R is a MFCIS of T_N^R.*

7.4.4 A recursive construction for MFCIS of T_N

Using Lemma 7.4.4 and Corollary 7.4.7 we give a construction for the MFCISs of T_N using the MFCISs of T_{N-1}. First we give the MFCISs for T_3, then we will construct the MFCISs for $T_N, N \geq 4$ using MFCISs of T_3.

Lemma 7.4.9. *The MFCISs of T_3 are the graphs in Figure 7.3 and the $freq_{T_3}(MFCIS(T_3)) = 2^5$.*

Proof. By direct computation. □

Observation 7.4.10. *The graphs in Figure 7.3 are non-isomorphic to each other.*

Lemma 7.4.11 gives a recursive construction for the MFCISs for T_4 using the MFCISs of T_3.

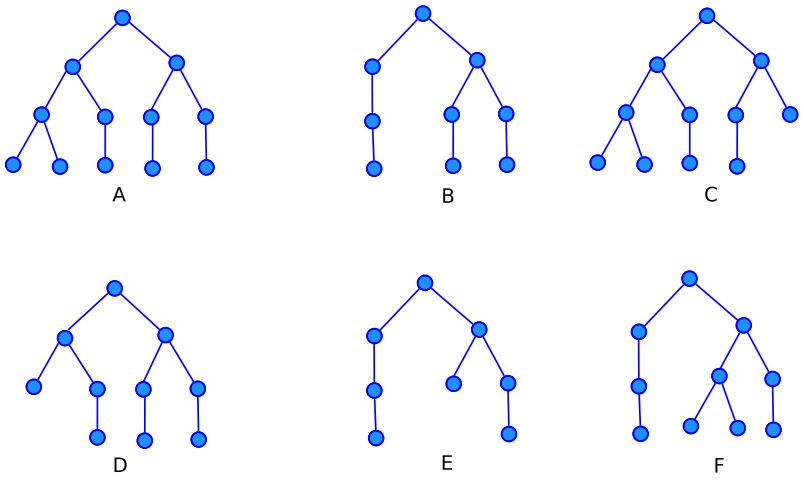


Figure 7.3: MFCISs of T_3.

Lemma 7.4.11. *Let A and B be any two distinct graphs in Figure 7.3. The tree with A and B as its left and right subtrees respectively is a MFCIS of T_4.*

Proof. Let J be a tree with A and B as its left and right child respectively. For the sake of contradiction, suppose J is not a MFCIS of T_4. This implies that there exists an induced subtree $H \ncong J$ of T_4, which is isomorphic to a MFCIS of T_4. Let H^L, H^R be left and right subtree of H. By Lemma 7.4.7, H^L and H^R are the MFCISs of T_4^L and T_4^R respectively.

By Lemma 7.4.4, $D \not\leq T_3$. Therefore

$$\begin{aligned}
&\text{freq}_{T_4}(H) \\
&\leq 2\ \text{freq}_{T_3}(H^L) \cdot \text{freq}_{T_3}(H^R) \qquad \text{using Fact 7.4.2} \\
&\leq 2\ \text{freq}_{T_3}(A) \cdot \text{freq}_{T_3}(B) \\
&= \text{freq}_{T_4}(J).
\end{aligned}$$

Hence the contradiction. □

Consider Figure 7.4. The purple vertices induce a subgraph isomorphic to graphs from Figure 7.3. The induced subgraphs in purple and green are two MFCISs of T_4 and the whole graph is a MFCIS of T_5. This graph is constructed recursively using the construction used in Lemma 7.4.11.

Observation 7.4.12. *There are* 15 *distinct non-isomorphic MFCISs for T_4.*

Proof. Using the construction of Lemma 7.4.11 and Observation 7.4.10, the number of distinct MFCISs for T_4 is $\binom{6}{2} = 15$. □

Counting the frequency of MFCIS(T_N) in T_N

In this section we will count the frequency of MFCIS(T_N) in T_N.

Theorem 7.4.13. *If $N > 3$, then*

$$freq_{T_N}(MFCIS(T_N)) = 2[freq_{T_{N-1}}(MFCIS(T_{N-1}))]^2.$$

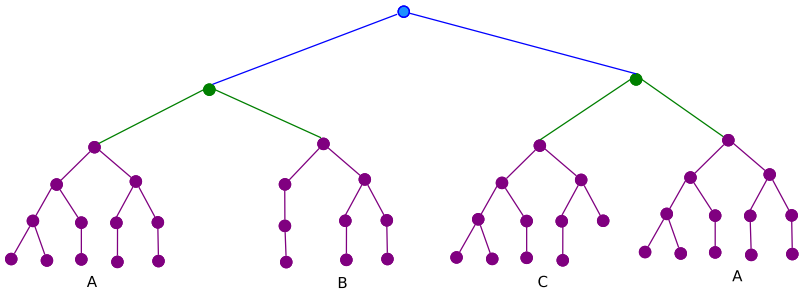


Figure 7.4: A MFCIS of T_5 constructed from MFCIS of T_3 using the construction used in Lemma 7.3.

Proof. By Observation 7.4.12 there are 15 non-isomorphic MFCISs for T_4. Using the construction of Lemma 7.4.11, we can construct $\binom{15}{2}$ non-isomorphic MFCISs for T_5. Continuing the same process we can construct $k > 2$ number of non-isomorphic MFCISs for T_{N-1}. Therefore by Fact 7.4.2,

$$\text{freq}_{T_N}(\text{MFCIS}(T_N)) = 2\ [\text{freq}_{T_{N-1}}(\text{MFCIS}(T_{N-1}))]^2.$$

□

Ratio of order of MFCIS(T_N) with order of T_N

Theorem 7.4.14. *For $N > 4$, $\frac{|V(MFCIS(T_N))|}{|V(T_N)|} \geq 0.60$.*

Proof. By the construction of the MFCIS(T_N), all vertices from level 0 to $N-4$ of T_N are also there in a MFCIS of T_N. The graph T_4 has 31 vertices. The average number of vertices in a MFCIS of T_4 using construction in Lemma 7.4.11, is 18. The number of T_4 contained in T_N is 2^{N-4}. Therefore

$$\begin{aligned}
&\frac{|V(\text{MFCIS}(T_N))|}{|V(T_N)|} \\
&\geq \frac{2^{N-3} - 1 + 18(2^{N-4})}{2^{N-3} - 1 + 31(2^{N-4})} \\
&= \frac{(2^{N-4}(18+2)) - 1}{(2^{N-4}(31+2)) - 1} \\
&\approx 0.61.
\end{aligned}$$

□

7.5 Conclusion

In this chapter we introduced the term MFCIS more rigorously. We proved that finding a MFCIS is #P-hard. We also developed a framework, which computes the MFCISs of all graphs with order up to 10. We proved for a perfect binary tree, the order of any MFCIS is a large proportion of binary tree. These results can be extended to any perfect m-ary tree. An open problem that arises from this research is to find a good characterisation of

a graph G which ensures the existence of a graph H such that G is a MFCIS. Another interesting aspect of the study is to find whether there exists of an efficient exact algorithm to find a MFCIS of a given graph.

Chapter 8

OLGA: a research tool

In this chapter we demonstrate how OLGA can be used as a research tool to generate new mathematical discoveries and verify some known results. We also depict graphs satisfying some interesting properties related to chromatic number and MFCIS.

8.1 Chromatic critical graphs

A graph is *chromatic critical* if removing a vertex or an edge from the graph decreases the chromatic number of the resultant graph by 1. Chromatic critical graphs are well studied in [25, 44, 141].

8.1.1 Edge critically chromatic graphs

An *edge-critical k-chromatic graph* is a graph with chromatic number k such that for each edge e of G, the graph $G \setminus e$ has chromatic number $k-1$.

EDGE-CRITICALLY k-CHROMATIC:
Input: A graph G and a number k.
Question: Is G edge-critically k-chromatic?

Lemma 8.1.1. *EDGE-CRITICALLY k-CHROMATIC is in D^P.*

Proof. Define the languages L_1, L_2 and L_3 as follows.

$$L_1 = \{G : G \text{ is } k\text{-colourable}\}.$$
$$L_2 = \{G : G \text{ is not } (k-1)\text{-colourable}\}.$$
$$L_3 = \{G : G \setminus e \text{ is } (k-1)\text{-colourable}, \forall e \in E(G)\}.$$

Observe that EDGE-CRITICALLY k-CHROMATIC $= L_1 \cap L_2 \cap L_3$. Here $L_1 \in$ NP, $L_2 \in$ co-NP and $L_3 \in$ NP. The class NP is closed under intersection, therefore $L_1 \cap L_3 \in$ NP. Therefore EDGE-CRITICALLY k-CHROMATIC $\in D^P$. □

We depict some edge-critically 4-chromatic graphs with order up to 8 using OLGA in Figure 8.1. We have generated all such graphs with order up to 10 and OLGA is

n	ECCG
2	1
3	1
4	1
5	2
6	2
7	5
8	9
9	47
10	338

Table 8.1: Number of edge-critical chromatic graphs with up to 10.

capable of generating all critically k-chromatic graphs with order up to 12. Royle listed all critically k-chromatic graphs, for $k \in [4, 10]$ with order up to 11 [127].

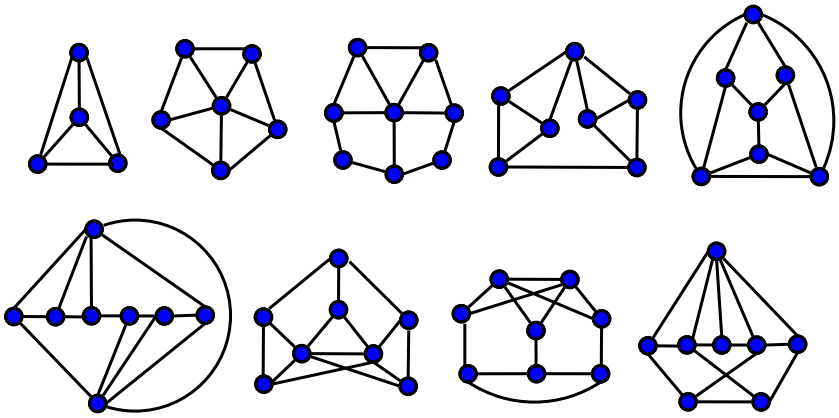


Figure 8.1: Some edge-critically 4-chromatic graphs with order up to 8 (graphs drawn with reference to [142]).

In Table 8.1 we present the number of edge-critically chromatic graphs with order up to 10. We use n in Table 8.1 to represent the number of vertices and ECCG represents the number of edge-critically k-chromatic graphs on n vertices.

We compared our data with Royle's data. Our results match with Royle's results on all graphs graphs of order $n \leq 10$. This independent re-computation of edge-critically chromatic graphs shows the correctness of the data generated by OLGA and also validates Royle's data. Another aspect of this data generation is to identify the relationship between different parameters, one such fact is presented in Figure 8.2.

In Table 8.2 we present the number of edge-critical k-chromatic graphs, for $k \in [2, 10]$ with order up to 10. We use k in Table 8.2 to represent the chromatic number and $\#k$-ECCG represents the number of edge-critically k-chromatic graphs with order up to 10.

k	#k-CCG
2	1
3	4
4	180
5	187
6	26
7	4
8	2
9	1
10	1

Table 8.2: Number of edge-critical k-chromatic graphs, for $k \leq 10$ with order up to 10.

We studied the edge-critically k-chromatic graphs generated by OLGA and observed there are symmetries in these graphs. Therefore we investigated the size of the automorphism group for these graphs. However we found some edge-critically k-chromatic graphs with trivial automorphism as shown in Figure 8.2.

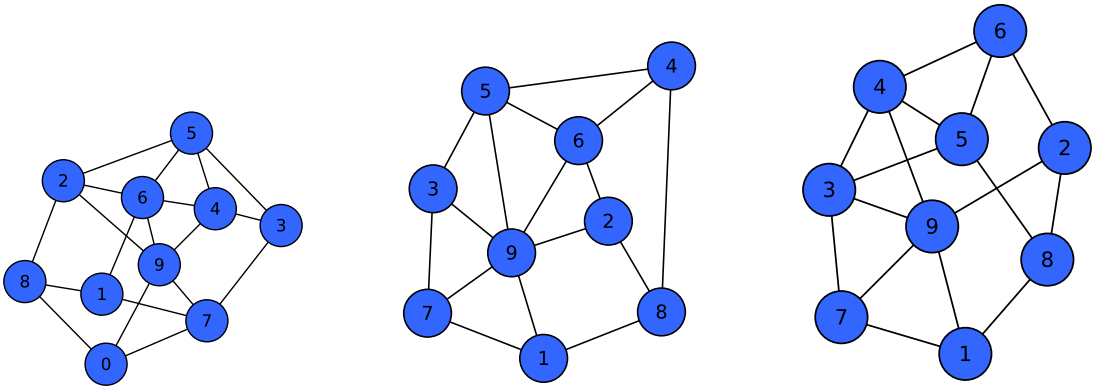


Figure 8.2: Edge-critically 4-chromatic graphs with trivial automorphism group.

8.1.2 Vertex-critically chromatic graphs

A *vertex-critical k-chromatic graph* is a graph with chromatic number k such that for each vertex v of G, the graph $G \setminus v$ has chromatic number $k-1$.

VERTEX CRITICALLY k-CHROMATIC:
Input: A graph G and a number k.
Question: Is G vertex-critically k-chromatic?

Lemma 8.1.2. *VERTEX CRITICALLY k-CHROMATIC is in D^P.*

Proof. Define the languages L_1, L_2 and L_3 as follows.

$$L_1 = \{G : G \text{ is } k\text{-colourable}\}.$$
$$L_2 = \{G : G \text{ is not } (k-1)\text{-colourable}\}.$$
$$L_3 = \{G : G - v \text{ is } (k-1)\text{-colourable}, \forall v \in V(G)\}.$$

Note that VERTEX CRITICALLY k-CHROMATIC $= L_1 \cap L_2 \cap L_3$. Here $L_1 \in$ NP, $L_2 \in$ co-NP and $L_3 \in$ NP. The class NP is closed under intersection, therefore $L_1 \cap L_3 \in$ NP. Therefore VERTEX CRITICALLY k-CHROMATIC $\in D^P$. □

Note that every edge-critically k-chromatic graph is also vertex-critically k-chromatic. Figure 8.3 contains all vertex-critically 4-chromatic graphs with order up to 8 that are not edge-critically 4-chromatic. We generated these graphs using OLGA. We have generated all such graphs with order up to 10.

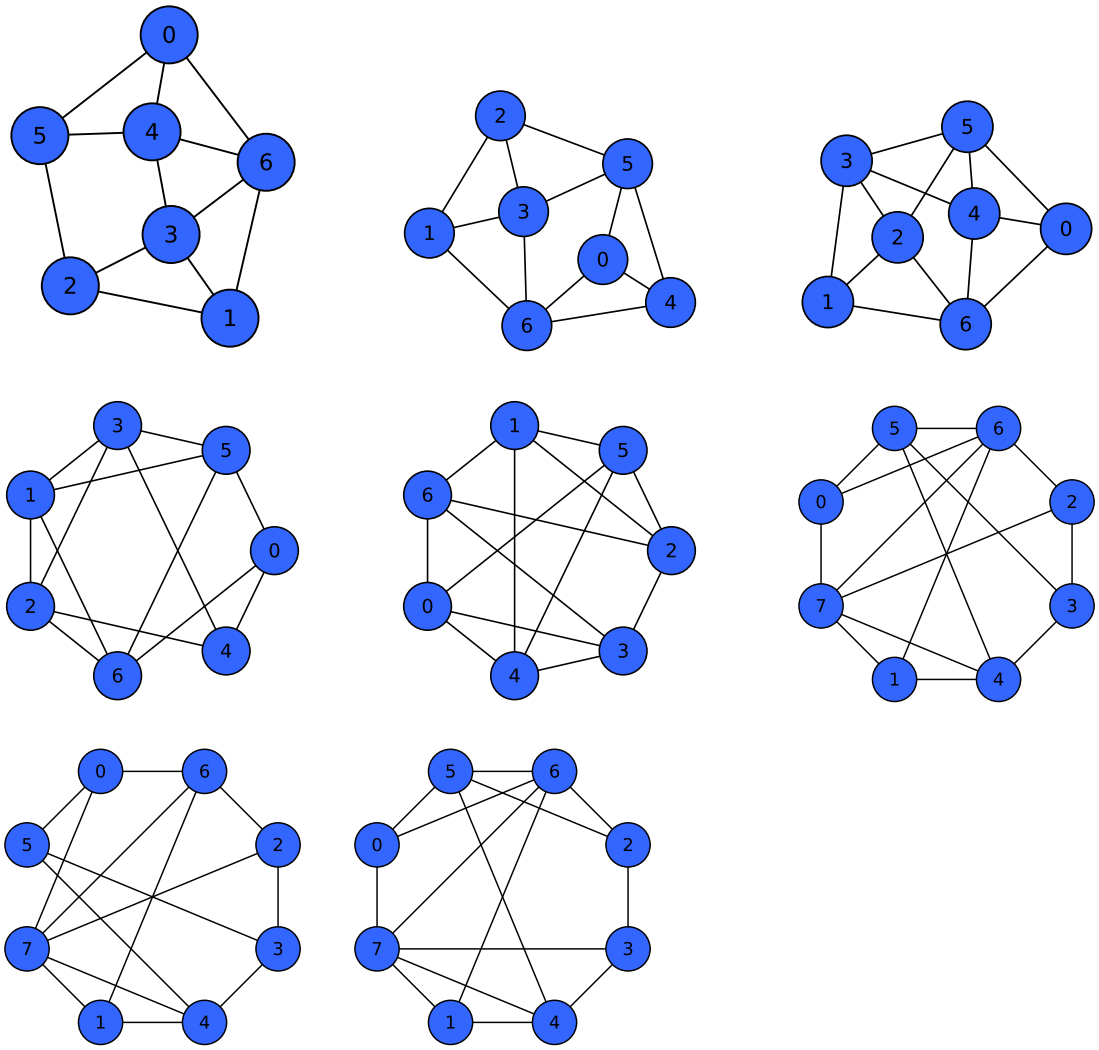


Figure 8.3: All vertex-critical 4-chromatic graphs with order up to 8 that are not edge-critical 4-chromatic

.

n	VCCG
2	1
3	1
4	1
5	2
6	2
7	10
8	20
9	368
10	5429

Table 8.3: Number of vertex-critical chromatic graphs with order up to 10.

k	#k-VCCG
2	1
3	4
4	2594
5	3076
6	246
7	9
8	2
9	1
10	1

Table 8.4: Number of vertex-critical k-chromatic graphs, for $k \leq 10$ with order up to 10.

We list the number of vertex-critically k-chromatic graphs with order up to 10 in Table 8.3. Note that these numbers also include the number of edge-critically chromatic graphs. We use n in Table 8.1 to represent the number of vertices and VCCG represents the number of vertex-critically k-chromatic graphs on n vertices.

We compared our data with Royle's data. Interestingly, our results differ from Royle's results. There were some graphs reported in Royle's list which we could not find. This requires further investigation and this signifies the benefits of tools like OLGA. We will investigate this issue further.

We also grouped the vertex-critically k-chromatic graphs with order up to 10 based on their chromatic number. We list the number of vertex-critically k-chromatic graphs with order up to 10 for a given k in Table 8.4. We use k to denote the chromatic number and #k-VCCG to denote all vertex-critically k-chromatic graphs with order up to 10.

We did this study to demonstrate how an extended use of OLGA can be useful in extracting more information on graphs and parameters that are present (can be generated) using OLGA.

8.2 MFCIS data from OLGA

We generated all the MFCISs for all unlabelled graphs with order up to 10. We classify the graphs based on the multiplicity of their MFCISs. We list our analysis in Table 8.5.

We use n in Table 8.5 to represent the number of vertices and m_i represents number of graphs on n vertices with exactly i distinct MFCISs.

n	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}	m_{11}
3	1	1	0	0	0	0	0	0	0	0	0
4	3	3	0	0	0	0	0	0	0	0	0
5	15	5	1	0	0	0	0	0	0	0	0
6	88	21	2	1	0	0	0	0	0	0	0
7	769	77	5	1	0	0	0	0	0	0	0
8	10374	685	49	5	2	2	0	0	0	0	0
9	243679	17011	354	19	9	3	3	2	0	0	0
10	10965151	733498	17666	188	39	7	7	7	2	1	1

Table 8.5: Multiplicity of MFCIS

Analysing Table 8.5 we obtain the following new integer sequences:

- The column m_1 of Table 8.5 gives the number of graphs with n vertices with unique MFCIS. The sequence is

$$1, 3, 15, 88, 769, 10374, 243679, 10965151, \cdots .$$

- The column m_2 of Table 8.5 gives the number of graphs with n vertices with the MFCIS multiplicity exactly 2. The sequence is

$$1, 3, 5, 21, 77, 685, 17011, 733498, \cdots .$$

- The column m_3 of Table 8.5 gives the number of graphs with n vertices with the MFCIS multiplicity exactly 3. The sequence is

$$0, 0, 1, 2, 5, 49, 354, 17666, \cdots .$$

We depict a few graphs with their MFCISs in Figures 8.4, 8.5 and 8.6. These graphs have the maximum multiplicity of MFCIS over 9 vertices and 10 vertices respectively. In Figure 8.4, there are 8 different MFCISs of the graph and the frequency of the MFCIS in the graph is 36. Similarly in Figure 8.5, there are 8 different MFCISs of the graph and the frequency of the MFCIS in the graph is 12. In Figure 8.6 there are 11 different MFCISs for the graph and the frequency of the MFCIS is 60. We also give the identification number (graph id) used in OLGA to uniquely identify these graphs.

The graphs presented in Figures 8.4, 8.5 and 8.6 have automorphism groups of size 72, 48 and 120 respectively. This gives an intuition that the multiplicity of the MFCISs increases if the size of automorphism group is big. To investigate this claim further, we selected all the graphs of order 9 and 10 whose multiplicity of the MFCISs is in the top 3 multiplicities among all graphs of order 9 and 10 respectively and computed the sizes of

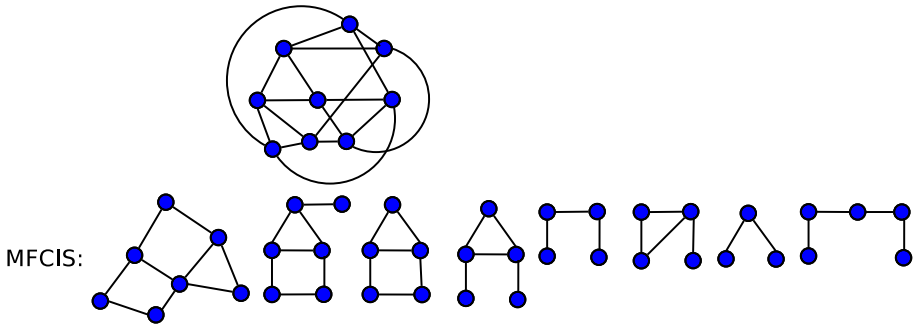


Figure 8.4: Graph of order 9 with graph id 3669386708, with its 8 MFCISs.

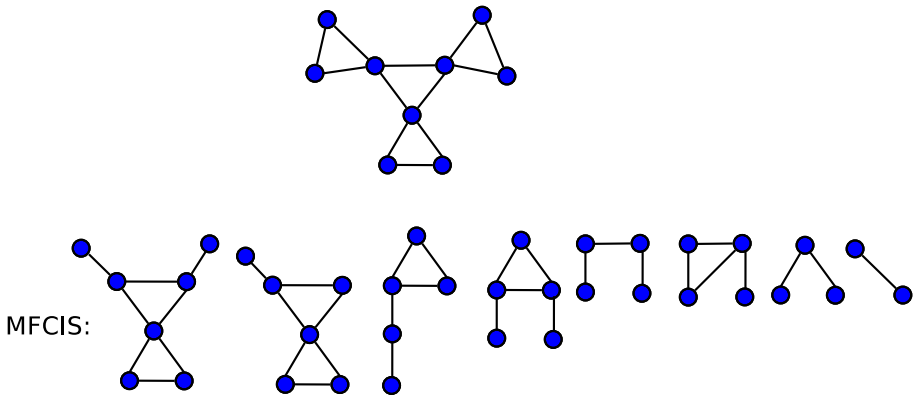


Figure 8.5: Graph of order 9 with graph id 1355399303, with its 8 MFCISs.

their automorphism groups using OLGA. We list these values in Table 8.6. The column *G*id refers to the graph identification number of the graph G[1] used in OLGA.

In Figure 8.7 we depict the corresponding graphs for the graph ids mentioned in Table 8.6. Note that there are 3 graphs from Table 8.6 are missing in Figure 8.7, as we have depicted them in Figures 8.4, 8.5 and 8.6.

Observe that the graph of order 10 with graph id 3771332167 has trivial automorphism. From Table 8.6 this graph has multiplicity 9 for its MFCISs. Therefore, this goes against our previous assumption on the relation between size of the automorphism group and the multiplicity of the MFCISs. This also raises the question: If there exists a parameter p such that if $p > k$ then the multiplicity of the MFCISs is at most k?

8.3 Conclusion

In this chapter we computed critically k-chromatic graphs and listed their occurrences with order up to 10. We can do similar things on other parameters that OLGA contains. For example, we can study about k-domination critical graphs. We also demonstrated how OLGA can generate research questions by analysing data generated from the MFCISs computation.

[1]We provide the code in Section A.4 that generates the graph using its graph id and order.

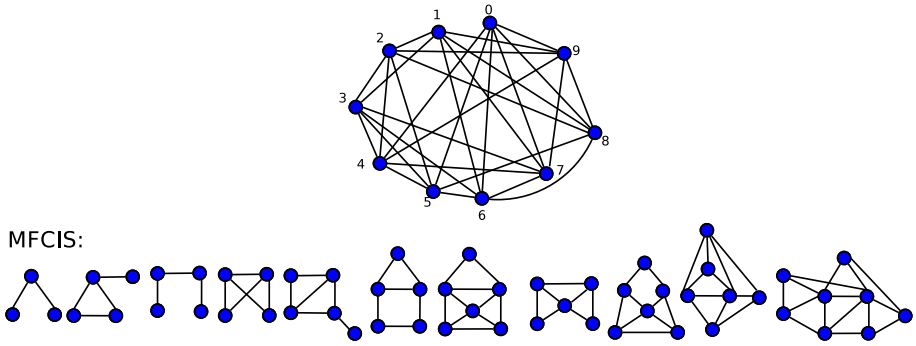
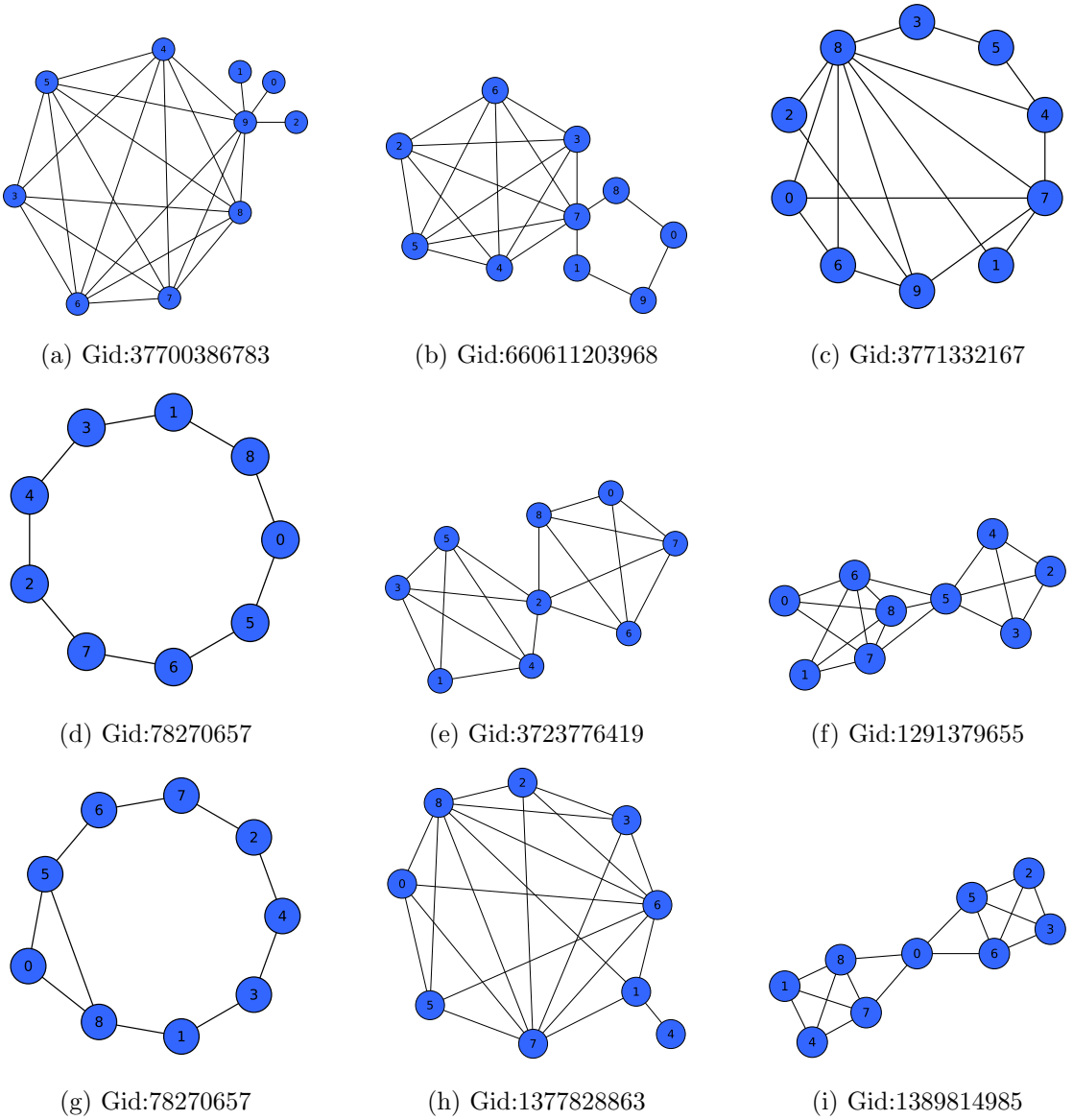


Figure 8.6: Graph of order 10 with graph id 6450870734291, with its 11 MFCISs.

Gid	n	multiplicity	$\lvert\mathrm{Aut}(G)\rvert$
78270657	9	6	2
1389814985	9	6	8
1377828863	9	6	288
78270656	9	7	18
1291379655	9	7	36
3723776419	9	7	72
1355399303	9	8	48
3669386708	9	8	72
3771332167	10	9	1
37700386783	10	9	720
660611203968	10	10	240
6450870734291	10	11	120

Table 8.6: Graphs with multiple MFCISs with the size of their automorphism group.



(a) Gid:37700386783 (b) Gid:660611203968 (c) Gid:3771332167

(d) Gid:78270657 (e) Gid:3723776419 (f) Gid:1291379655

(g) Gid:78270657 (h) Gid:1377828863 (i) Gid:1389814985

Figure 8.7: Graphs with graph ids from Table 8.6.

Chapter 9

Conclusion and future work

This chapter summarises the tasks carried out in this thesis. We give insights and future directions.

9.1 Highlights of the thesis

We started this ambitious project with a hope to build a large, novel system, OLGA[1] which will not only serve the purpose of a queryable graph repository but also lay the foundations for new mathematical discoveries.

We studied various graph repositories thoroughly and made a few observations on the specialties of the repositories. Details of graph repositories are discussed in Chapter 3. This study helped us in understanding the difficulties in creating a repository of graphs.

We managed to create a queryable repository of graphs, OLGA, which stores all simple connected unlabelled graphs with up to 10 vertices and more than 25 parameters (refer to Chapter 6 for details) related to these graphs.

During this process of creating OLGA we solved some practical technical problems as discussed in Chapter 5. We also highlighted the architecture used in OLGA for parameter computation.

We demonstrated the facilities that OLGA provides as a research tool in Chapter 4. We related the recursive nature of computing parameters in OLGA with self-reducibility, from complexity theory, in Section 4.5.

We introduced MFCIS as a parameter to compare various computation methods in the OLGA setup. However MFCIS is theoretically rich as a parameter by itself. We developed the theory of MFCIS and reported our findings in Chapter 7.

After generating the graph-data using OLGA, we demonstrated some extended use of OLGA as a research tool. We analysed these data and reported our findings in Chapter 8.

In the next section we give some enhancements and future direction for this work.

[1] A prototype of OLGA was done by Barnes [10], in 2009. For details refer to Section 3.2.3

9.2 OLGA extension

The natural extension of OLGA is to extend it to graphs of higher order. However, the space constraint is a bottleneck in this case. Table 9.1 shows the space requirements to store only graphs in `graph6` format.

# vertices	# connected unlabelled graphs	Memory required
2	1	3B
3	2	6B
4	6	18B
5	21	84B
6	112	560B
7	853	5.1KB
8	11 117	77.8KB
9	261 080	2.1MB
10	11 716 571	117.2MB
11	1 006 700 565	12.1GB
12	164 059 830 476	≅ 2.13TB
13	50 335 907 869 219	≅ 760TB

Table 9.1: Space requirements for graphs.

Table 9.1 shows that storing all graphs up to 13 vertices requires huge amount of space. Another challenge is the storage of the metadata, i.e., the parameters associated with each graph. In the case of OLGA the space needed for storing the parameters is much more than the space needed to store the graphs. We highlighted this issue in case of the Tutte polynomial and eigenvalues in section 6.2. Therefore extending OLGA up to 12 vertices is feasible (although difficult). We are in the process of extending OLGA up to 11 vertices. Considerable improvement on the storage can be made by storing the parameter values in some compressed form, like McKay's `graph6` format for graphs.

In the current version of OLGA, we do not store properties of graphs, as computing some of the properties like hamiltonicity and eulerianness requires only one query to OLGA. For example, the query "list all graphs of order n whose circumference is exactly n" will list all graphs that contain a Hamiltonian cycle. Another such property is planarity which can be inferred from genus. For example, the query "list all graphs whose genus value is 0" will list all planar graphs. Storing these properties requires only one bit of space per graph. Moreover these properties are so widely used that storing them will be more useful.

A straightforward extension of OLGA is to add parameters which can be computed from the data of existing parameters. In the current version of OLGA we used this technique to compute clique number and vertex cover number from independence number. The advantage of this approach is computationally we get the data of several related parameters almost freely while computing one parameter.

Another direction of extending OLGA is by adding more parameters which cannot be computed from the existing data. We already have a scalable design of OLGA, so adding new parameters should be simple. However, we are building a test suite for OLGA

to ensure the correct functionality of OLGA on addition of new parameters. The test suite will perform the basic checks like the chromatic index is not less than the maximum degree or edge connectivity is not less than vertex connectivity. Suppose we introduce a new graph parameter to OLGA. The role of the test suite is to ensure that all previous parameters do not malfunction after the new parameter gets added to OLGA.

On the theory front, new discoveries can arise from analysis of data generated by OLGA. For example, in Section 8.2 we indicated a potential relationship between multiplicity of MFCIS with size of the automorphism group. Similar relations can be explored between other graph parameters analysing data generated by OLGA.

Another application of OLGA is to verify existing conjectures involving the parameters that OLGA contains. We demonstrated a use case of how OLGA can be used to verify conjectures in Section 4.1.

9.3 MFCIS characterisation

In this thesis we introduced the term MFCIS and proved that finding a MFCIS is #P-hard. We also proved that the order of the MFCISs of perfect binary trees is a large proportion of the order of the tree. The scope of research in MFCIS is very broad. We list some of the potential research questions involving the MFCIS.

- Finding a good characterisation of G such that there exists a graph H such that G is a MFCIS of H.
- The algorithm we used to compute the MFCISs of a graph is a naive algorithm that uses exponential space. However, it will be interesting to explore the existence of a polynomial space algorithm for finding a MFCIS.
- We depicted a few graphs with their MFCISs in Chapter 9. There seems to be a relation between the size of the automorphism group of the graph and multiplicity of MFCIS. Can we find the relationship between these two or is there a property that guarantees multiple MFCISs exist for a graph?
- What is the effect of vertex (edge) deletion on MFCIS and on the frequency of MFCIS of a graph?
- Finding a characterisation of a graph G of bounded treewidth which ensures the problem of finding a MFCIS of G is fixed-parameter tractable?
- Finding a good characterisation of a graph G of bounded treewidth which ensures the existence of a polynomial time algorithm for computing a MFCIS of G?
- Characterising MFCIS for H-free graphs, e.g. triangle-free graphs.

9.4 Bound matching strategy

In Chapter 6 we discussed recursive algorithms for computing graph parameters. We computed recursive upper bounds and lower bounds for parameters that do not have a

simple recursion. We reported the value of the parameter where the greatest lower bound coincides with the least upper bound. We listed such cases for chromatic index, treewidth and genus. This gives us an indication of the accuracy of the recursions used to compute these parameters. A natural question to ask is:

- What is the effect of additional constraints in the recursive bound matching strategy? For example, in the case of treewidth, adding an extra condition that the treewidth of a tree is 1 reduced the percentage of graphs with unmatched bounds.

More difficult questions to ask are:

- Given recursive bounds, can we predict an upper bound and a lower bound on the number of unmatched cases?
- Can we prove the proportion of unmatched cases converges to a limit as $n \to \infty$?

Appendix A

A guide to OLGA

This chapter describes the internals of OLGA code. We start the chapter by giving commands for quick installation. We then explain the directory structure of OLGA. We give parameter-specific information (code) in the next section. We conclude the chapter by giving a utility function that converts the graph id to the corresponding graph.

A.1 OLGA installation

OLGA was built on the following system.

System specification:
Hardware: OptiPlex 9020 (OptiPlex 9020) (Dell Inc.)
Width: 64 bits
Product: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
Clock: 100MHz
RAM: 8GB
O/S: Ubuntu 16.04.5 LTS

Prerequisite:
Operating system: `Ubuntu 14.04 LTS` or higher.
Language: `C++11`, Python 3.5 onwards.
Compiler: `clang 3.3`
Database: `SQLite 3.11.0`

We list the sequence of commands to be followed for installing OLGA in the command line. This information is also available in a file named README.md inside OLGA directory. We recommend `bash` shell to run the commands.

```
# Installation
sudo apt-get install git
git clone <git repository for OLGA>
sudo apt-get install libeigen3-dev clang libomp-dev
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev
```

```
libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev
## 1.Googletest: This is a unit test library to help debugging.
This can be ignored if you are an end user.

make init


## 2. Backward-cpp: This part is important as this provides
the stack trace in linux.

git submodule init
git submodule update

#If the below command fails, then use this alternate command:

git clone https://github.com/bombela/backward-cpp.git 3rd-party/backward-cpp

## 3. backward-cpp dependencies

sudo apt-get -y install binutils-dev

## 4.Install pstream: This allows us to execute
other c++ programs from our current interface.

sudo apt-get -y install libpstreams-dev


## 5. Nauty stuff : This was not part of OLGA till 2018, we enabled a branch
with nauty to speed-up the graph generation process.

wget http://pallini.di.uniroma1.it/nauty27rc1.tar.gz
#or
wget http://users.cecs.anu.edu.au/~bdm/nauty/nauty27rc2.tar.gz
tar xf nauty27rc1.tar.gz 3rd-party/
./configure
make


## 6. Create data folder: This folder stores all the data generated by OLGA.

mkdir data


## 7. Finishing up: If everything worked fine till this point, then we are
ready to start OLGA. Always clean the binaries before getting started.

make clean
make welcome
THREADS=1 OPTION=3 START=3 STOP=4 ./welcome
```

A.2 OLGA directory structure

We used self-explanatory directory names. The directories in OLGA are listed here.

- **include**: This directory contains all the header files for the programs used in OLGA to generate graphs and to compute parameters.
- **src**: This directory contains all the source code for various parameters of OLGA and for graph generation.
- **python**: This directory contains all the python code needed for OLGA to interact with the database used in user interface. It also contains code for computing parameters like the Tutte polynomial and eigenvalues.
- **scripts**: This directory contains scripts for automating the steps that are common for adding a new parameter. For example, if the user wants to add a new parameter p, we automate the corresponding header files, source files and class in appropriate directories. So the user only needs to write code in the specific directories. This ensures that a new user will not break the existing code.
- **tests:** This directory will act as the test suite for OLGA. In the current state, the skeleton of the test suite is ready, we are in the process of filling in the test cases. The idea is to test the correctness of a parameter by verifying the known properties of the parameter as soon as the parameter gets implemented. For example, if the longest path is implemented we check the following conditions:

  $\mathrm{lp}(G) \geq \mathrm{circ}(G) - 1$,
  $\mathrm{lp}(P_n) = n$,
  $\mathrm{lp}(C_n) = n - 1$.

  We first create a few test graphs as mentioned in the conditions above. If $\mathrm{lp}(G)$ passes the above tests we delete these test graphs and allow the user to integrate the code to OLGA. Otherwise we raise an exception with the failed test case.

A.3 OLGA code structure

In this section we will first discuss the naming convention of files and the basic structure of the code. We tried to make these conventions uniform across all parameters.

Since graphs are the backbone of OLGA, we will give some insight on the graph class in this section. Operations like vertex deletion and edge deletion are frequently used in parameter computation, so we added these frequently-used functions to the graph class. We list the members[1] of the graph class here, although we will not give a detailed description of them. The variable names are mostly self-explanatory.

[1]Variables and functions declared inside a class are *members* and *member functions* of the class. The instances of the class are called *objects*.

```
class Graph {
public:

        int num_vertices;
        long long graph_id;

        Graph();
        Graph(int _num_vertices);

        // read from file
        Graph(string filename);
        ~Graph();

        void print(string str_prefix="") const;
        void print2(string str_prefix="") const;
        std::string get_string(std::string str_prefix="") const;
        bool connected() const;
        Graph* clone() const;
        bool is_edge(int ix, int jx) const;
        void copy(Graph* dst) const;
        void copy(Graph* dst, VectorFixed* vertices) const;
        bool is_k_n(long long certificate=-1) const;
        bool is_tree(bool b_connected) const;


        std::vector<int> get_degrees() const;
        std::vector<int> get_neighbors(int vertex) const;
        int get_max_degree(std::vector<int> degrees = std::vector<int>()) const;
        std::vector<int> get_cut_vertices() const;
        int num_neighbors(int vertex) const;

        void copy_edges_from(Graph const* src, int ix_vertex);
        void copy_edges_from(Graph const* src, std::vector<int> vertices);
        void copy_sans_vertex_into(Graph* other_graph, int vertex) const;
        void copy_sans_blanket_into(Graph* other_graph, int vertex) const;
        int number_of_nbrs(int vertex) const;
        void copy_complement_to(Graph* dst) const;

        //bool is_tree();
        int number_of_edges();
        int get_number_of_edges() const;
        void add_edge(int ix, int jx);
        void set_edge(int row, int col);
        void remove_edge(int ix, int jx);
        void reset_vertex(int ix_vertex);
        void reset_edge(int row, int col);
        void set_all_edges();
        void reset_all_edges();
        void construct_graph_id();
        void imbibe_graph_id(long long graph_id);
        // not thread safe
```

```
        void complement();

        // generate a complete graph with n vertices
        static Graph* K_n(int n);
        // store and retrieve number of upper triangular entries
        static vector<long> NUM_UPPER_TRIANGULAR;
        static void init_upper_triangular(int n);
        static long num_upper_triangular(int n);
        static Graph* from_graph_id(long long graph_id);

private:
        // init the matrix with zeros
        void init_matrix(int num_vertices);
        void fill(bool value); // where all?
        void fill_diagonal(bool value); // where all?

        void set_num_vertices(string line);
        void read_adj_list(ifstream file);
        void update_reachability(int, bool*) const;
        void check_bounds(int row, int col) const;
        void validate_row_col(int row, int col, std::string requestor="") const;

        void get_cut_vertices_helper(int vertex, bool *visited, int *discovered,
                                     int *lows, int *parents, bool *art_points,
                                     int& time_cut_vertex) const;

        //MatrixXf mat;
        bool **data;
        int _num_edges;
};
```

Every parameter in OLGA uses the graph class as the base of its parameter class. However each parameter requires its own variables and functions based on the computation method. So each parameter inherits some members from the graph class and adds new members that are parameter specific and private to the parameter class.

Every parameter has two header files.

The first one is responsible for reading and writing graph ids and parameter values from and to the database. It also contains the base case used in the recursion. So this file stores the signature of these operations. The naming convention for this file is "parameter name.hh".

The second one is a helper for the first one. Actual parameter computation happens in the helper file. In the case of parameters using recursive bounds, the helper file computes the recursive bounds and sends the corresponding upper and lower bounds with the value where these bounds coincide to the callee function[2]. The corresponding header file stores the signature of the functions used for parameter computation. The naming convention for this file is "parameter name_helper.hh".

[2]If a function A calls another function B, then B is the *called* function and A is the *callee* function.

Parameter	Header file	Source file
Automorphism group size	aut.hh aut_helper.hh	aut.cc aut_helper.cc
Chromatic index	chromatic_index.hh chromatic_index_helper.hh	chromatic_index.cc chromatic_index_helper.cc
Chromatic number	chromatic_number_exact.hh chromatic_number_exact_helper.hh	chromatic_number_exact.cc chromatic_number_exact_helper.cc
Degeneracy	degeneracy.hh degeneracy_helper.hh	degeneracy.cc degeneracy_helper.cc
Domination number	domination_number.hh domination_number_helper.hh	domination_number.cc domination_number_helper.cc
Genus	genus.hh genus_helper.hh	genus.cc genus_helper.cc
Graph basic [3] properties	graph_basic_properties.hh graph_basic_properties_helper.hh	graph_basic_properties.cc graph_basic_properties_helper.cc
Graph [4] connectivity	graph_connectivity.hh graph_connectivity_helper.hh	graph_connectivity.cc graph__connectivity_helper.cc
Graph metric [5] properties	graph_metric_properties.hh graph_metric_properties_helper.hh	graph_basic_properties.cc graph_metric_properties_helper.cc
MFCIS	all_max_induced_subgraphs.hh all_max_induced_subgraphs_helper.hh	all_max_induced_subgraphs.cc all_max_induced_subgraphs_helper.cc
Independence number [6]	ind_number.hh ind_number_helper.hh	ind_number.hh ind_number_helper.cc
Treewidth	treewidth.hh treewidth_helper.hh	treewidth.cc treewidth_helper.cc

Table A.1: Parameter files.

Each header file has a corresponding source file where the tasks are performed. The source file corresponding to "parameter name.hh" is "parameter name.cc". Similarly the source file corresponding to "parameter name_helper.hh" is "parameter name_helper.cc".

Both "parameter name.hh" and "parameter name_helper.hh" are stored in the include directory.

Both "parameter name.cc" and "parameter name_helper.cc" are stored in the src directory.

If the parameter is computed using a direct recursion we use the same naming convention as discussed. However if the parameter is computed using exact algorithms we use a different convention. The header files are stored as "parameter name_exact.hh" and the corresponding source file is stored as "parameter name_exact.cc".

We list all parameters and their corresponding header files and source files in Table A.1. Parameters like the Tutte polynomial, eigenvalues, longest path and maximum matching are not listed in Table A.1. For simplicity we computed these parameters using python code. The source files for these parameters are present in the python directory.

[3] Order, number of components, degree sequence.
[4] Vertex connectivity, edge connectivity.
[5] Circumference, Diameter, Girth, Radius.
[6] also clique number and vertex cover number.

A.4 A useful function

We store the graphs through an identification number called graph id. In this section we present the python code to generate the graph in the form of an edge list from the graph id.

```
import graphviz as gv
import math
import numpy as np
def gid_to_adj_matrix(num_vertices, graph_id):
    adj_matrix =
    [[False for _ in range(num_vertices)] for _ in range(num_vertices)]
    graph_id_orig = graph_id
    for col in reversed(range(1,num_vertices)):
        for row in reversed(range(col)):
            remainder = graph_id % 2
            print('({0},{1}): {2},{3}'.format(row,col,graph_id,remainder))
            adj_matrix[row][col] = (remainder == 1)
            graph_id //= 2
    return adj_matrix
def adj_to_gv(adj_matrix):
    num_vertices = len(adj_matrix)
    g = gv.Graph(engine='neato')
    for ix in range(num_vertices):
        g.node(str(ix))
    for row in range(num_vertices):
        for col in range(row, num_vertices):
            if(adj_matrix[row][col]):
                g.edge(str(row),str(col))
    return g

def gid_to_gv(num_vertices, graph_id):
    return adj_to_gv(gid_to_adj_matrix(num_vertices, graph_id))

print(gid_to_gv(8,5684461)) # The first argument in this function is
the order of the graph and the second one is the graph id.
```

References

[1] N. Alon and J. H. Spencer. *The Probabilistic Method, 3rd edition*. Wiley, 2008.

[2] D. Archdeacon and C. P. Bonnington. Obstructions for embedding cubic graphs on the spindle surface. *Journal of Combinatorial Theory, Series B*, 91(2):229–252, 2004.

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

[4] L. Babai. Fixing the upcc case of split-or-johnson. `http://people.cs.uchicago.edu/~laci/upcc-fix.pdf`. accessed on 2019-02-22.

[5] L. Babai. Graph isomorphism in quasipolynomial time. *CoRR*, abs/1512.03547, 2015. `http://arxiv.org/abs/1512.03547`.

[6] R. Bailey, A. Jackson, and C. Weir. Distanceregular.org. `http://www.distanceregular.org/index.html`. Accessed: 2019-02-03.

[7] G. A. Baker, H. E. Gilbert, J. Eve, and G. S. Rushbrooke. *A Data Compendium of Linear Graphs with Application to the Heisenberg Model*. BNL (Series). Brookhaven National Laboratory, 1967.

[8] H. H. Baker, A. K. Dewdney, and A. L. Szilard. Generating the nine-point graphs. *Mathematics of Computation*, 28(127):833–838, 1974.

[9] J. Balcázar. Self-reducibility. *Journal of Computer and System Sciences*, 41(3):367–388, 1990.

[10] N. Barnes. *Towards an Online Graph Atlas*. BCompSc Hons dissertation. Clayton School of Information Technology, Monash University, 2009.

[11] V. Batagelj and M. Zaversnik. An $O(m)$ Algorithm for Cores Decomposition of Networks. *CoRR*, cs.DS/0310049, 2003. `https://arxiv.org/abs/cs.DS/0310049`.

[12] K. Berčič and J. Vidali. Discrete ZOO: Towards a fingerprint database of discrete objects. In *6th International Congress on Mathematical Software*, volume 10931 of *Lecture Notes in Computer Science*, pages 36–44, Editors: J. H. Davenport and M. Kauers and G. Labahn and J. Urban. South Bend, IN, USA, 2018.

[13] A. Berson. *Client/Server Architecture*. McGraw-Hill, Inc., 1992.

[14] G. H. Birkhoff. A determinant formula for the number of ways of coloring a map. *Annals of Mathematics*, 14:42–46, 1912–1913.

[15] C. M. Bishop. *Neural networks for pattern recognition*. Oxford University Press, 1995.

[16] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos. On exact algorithms for treewidth. *ACM Trans. Algorithms*, 9(1):12:1–12:23, 2012.

[17] A. Bondy. Beautiful conjectures in graph theory. *European Journal of Combinatorics*, 37:4–23, 2014.

[18] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. North Holland, New York, 1976.

[19] C. P. Bonnington and C. H. C. Little. *The Foundations of Topological Graph Theory*. Springer, New York, 2012.

[20] R. Boppana, J. Håstad, and S. Zachos. Does co-NP have short interactive proofs? *Information Processing Letters*, 25:27–32, 1987.

[21] R. Bose. Strongly regular graphs, partial geometries and partially balanced designs. *Pacific Journal of Mathematics*, 13(2):389–419, 1963.

[22] G. Brinkmann, J. Goedgebeur, H. Mélot, and K. Coolsaet. House of Graphs: a database of interesting graphs. *Discrete Applied Mathematics*, 161:311–314, 2013.

[23] G. Brinkmann, J. Goedgebeur, and J. Schlage-Puchta. Ramsey numbers $r(k_3, g)$ for graphs of order 10. arxiv preprint arxiv:1208.0501. 2012.

[24] G. Brinkmann and B. D. McKay. The program plantri. `https://users.cecs.anu.edu.au/~bdm/plantri/`, 2007. Accessed: 2019-01-19.

[25] R. L. Brooks. On colouring the nodes of a network. *Mathematical Proceedings of the Cambridge Philosophical Society*, 37(2):194–197, 1941.

[26] A. E. Brouwer and E. Spence. Cospectral graphs on 12 vertices. *The Electronic Journal of Combinatorics*, 16(1):N20, 2009.

[27] S. A. Burr. Diagonal Ramsey numbers for small graphs. *Journal of Graph Theory*, 7(1):57–69, 1983.

[28] F. C. Bussemaker, S. Cobeljic, D. M. Cvetkovic, and J. J. Seidel. Computer investigation of cubic graphs. *EUT report. WSK, Dept. of Mathematics and Computing Science*, 76, 1976.

[29] P. J. Cameron. Orbit counting and the Tutte polynomial. In *[58]*, pages 1–10.

[30] J. D. Carpinelli and A. Y. Oruc. Applications of Matching and Edge-coloring Algorithms to Routing in Clos Networks. *Networks*, 24(6):319–326, 1994.

[31] A. Cayley. On the analytical forms called trees, with application to the theory of chemical combinations. *Report of the British Association for the Advancement of Science*, 45:257–305, 1875.

[32] F. R. K. Chung and F. C. Graham. *Spectral Graph Theory*. American Mathematical Society, 1997.

[33] E. Cockayne, S. Goodman, and S. Hedetniemi. A linear algorithm for the domination number of a tree. *Information Processing Letters*, 4(2):41–44, 1975.

[34] M. Conder. Combinatorial data. `https://www.math.auckland.ac.nz/~conder/`, 2002. Accessed: 2016-01-03.

[35] M. Conder. Trivalent (cubic) symmetric graphs on up to 10000 vertices. `https://www.math.auckland.ac.nz/~conder/symmcubic10000list.txt`, 2011. Accessed: 2018-10-23.

[36] M. Conder and P. Dobcsányi. Trivalent symmetric graphs on up to 768 vertices. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 40:41–63, 2002.

[37] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1):99–143, 2007.

[38] K. Coolsaet, J. Degraer, and E. Spence. The strongly regular $(45, 12, 3, 3)$ graphs. *The Electronic Journal of Combinatorics*, 13(1):32, 2006.

[39] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Evaluating Performance of the VF Graph Matching Algorithm. In *ICIAP '99: Proceedings of the 10th International Conference on Image Analysis and Processing*, pages 1172–1177. IEEE Computer Society, Washington, DC, USA, 1999.

[40] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for CNF. In *DAC'04: Proceedings of the 41st Annual Design Automation Conference*, pages 530–534. ACM, NY, USA, 2004.

[41] E. R. V. Darn and E. Spence. Small regular graphs with four eigenvalues. *Discrete Mathematics*, 189:233–257, 1998.

[42] R. Diestel. *Graph Theory, 3rd edition*. Springer, New York, 2010.

[43] Y. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math Doklady*, 11:1277–1280, 1970.

[44] G. A. Dirac. The structure of k-chromatic graphs and some remarks on critical graphs. *Journal of the London Mathematical Society*, 27:269–271, 1952.

[45] C. Domb. On the theory of cooperative phenomena in crystals. *Advances in Physics*, 9(34–35), 1960.

[46] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.

[47] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

[48] H. C. Ehrlich and M. Rarey. Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(1):68–79, 2011.

[49] F. V. Fomin, F. Grandoni, and D. Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *EATCS Bulletin*, 87:47–77, 2005.

[50] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956.

[51] R. M. Foster and I. Z. Bouwer. *The Foster Census : R.M. Foster's census of connected symmetric trivalent graphs*. Winnipeg, Canada : Charles Babbage Research Centre, 1988.

[52] Django Software Foundation. Django 2.1. `https://www.djangoproject.com/`, 2018. Accessed: 2018-10-23.

[53] R. J. Frazer. *Graduate course project*. unpublished, Department of Combinatorics and Optimization, University of Waterloo, 1973.

[54] T. Gal. Degeneracy graphs: theory and application an updated survey. *Annals of Operations Research*, 46(1):81–105, 1993.

[55] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., San Francisco, 1979.

[56] C. Giatsidis. *Graph Mining And Community Evaluation With Degeneracy*. PhD thesis. Ecole Polytechnique, 2013.

[57] J. Goedgebeur and S. P. Radziszowski. New computational upper bounds for Ramsey numbers $r(3, k)$. *The Electronic Journal of Combinatorics*, 20(1):P30, 2013.

[58] G. Grimmett and C. McDiarmid, editors. *Combinatorics, Complexity, and Chance: A Tribute to Dominic Welsh*. Oxford Lecture Series in Mathematics and Its Applications 34, Oxford University Press, 2007.

[59] J. L. Gross and T. W. Tucker. *Topological Graph Theory*. Dover Publications, Inc., New York, 1987.

[60] B. Grünbaum and T. S. Motzkin. The number of hexagons and the simplicity of geodesics on certain polyhedra. *Canadian Journal of Mathematics*, 15:744–751, 1963.

[61] A. A. Hagberg, D. A. Schult, and P. J. Swart. In *Exploring network structure, dynamics, and function using networkx*, pages 11–15, Editors: G. Varoquaux, J. Millman, T. Vaught. Pasadena, CA, 2008.

[62] G. Haggard. Chromatic polynomials of 9 cages. `http://www.eg.bucknell.edu/~graphs/9cages.htm`. Accessed: 2019-02-03.

[63] G. Haggard. Chromatic polynomials of complete graphs. `http://www.eg.bucknell.edu/~graphs/complete.htm`. Accessed: 2019-02-03.

[64] G. Haggard. Tutte polynomials of complete graphs. `http://www.eg.bucknell.edu/~graphs/tutte.htm`. Accessed: 2019-02-03.

[65] F. Harary. *Graph Theory*. Addison-Wesley, 1969.

[66] F. Harary and E. M. Palmer. *Graphical Enumeration*. Academic Press, 1973.

[67] F. Harary and A. J. Schwenk. Which graphs have integral spectra? In *Graphs and Combinatorics*, pages 45–51. Editors: R. A. Bari and F. Harary. Springer, Berlin, Heidelberg, 1974.

[68] T. W. Haynes, S. T. Hedetniemi, and P. J. Slater. Fundamentals of domination in graphs, volume 208 of monographs and textbooks in pure and applied mathematics, 1998.

[69] B. R. Heap. The production of graphs by computer. In *Graph Theory and Computing*, pages 47–62. Editors: R. C. Read. Academic Press, Cambridge, Massachusetts, 1972.

[70] A. Hill and S. Wilson. Four constructions of highly symmetric tetravalent graphs. *Journal of Graph Theory*, 71(3):229–244, 2012.

[71] T. Hoppe and A. Petrone. Integer sequence discovery from small graphs. *Discrete Applied Mathematics*, 201(C):172–181, 2016.

[72] A. Hulpke. Constructing transitive permutation groups. *Journal of Symbolic Computation*, 39(1):1–30, 2005.

[73] P. Hutchinson. Diagram expressions useful in the theory of fluids. Technical report, United Kingdom Atomic Energy Authority (Research Group), 1964.

[74] Wolfram Research, Inc. *Mathematica, Version 11.3*. Wolfram Research, Inc., 2018. Champaign, IL.

[75] B. Jackson. A zero-free interval for chromatic polynomials of graphs. *Combinatorics, Probability and Computing*, 2(3):325–336, 1993.

[76] T. Januario, S. Urrutia, C. C. Ribeiro, and D. Werra. Edge coloring: A natural model for sports scheduling. *European Journal of Operational Research*, 254(1):1–8, 2016.

[77] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *ALENEX07: Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments*, pages 135–149. SIAM, 2007.

[78] I. N. Kagno. Linear graphs of degree ≤ 6 and their groups. *American Journal of Mathematics*, 68(3):505–520, 1946.

[79] J. G. Kalbfleisch. *Chromatic graphs and Ramsey's theorem.* PhD Thesis. University of Waterloo, 1966.

[80] R.M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Editors: R.E. Miller and J. Thatcher, Plenum, 1972.

[81] A. W. J. Kolen, J. K. Lenstra, C. H. Papadimitriou, and F. C. R. Spieksma. Interval scheduling: A survey. *Wiley Periodicals Inc.*, pages 530–542, March 2007.

[82] A. V. Kostochka and B. Y. Stodolsky. On domination in connected cubic graphs. *Discrete Mathematics*, 304(1–3):45–50, 2005.

[83] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search.* CRC Press, 1998.

[84] C. Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta mathematicae*, 15(1):271–283, 1930.

[85] D. R. Lick and A. T. White. k-degenerate graphs. *Canadian Journal of Mathematics*, 22:1082–1096, 1970.

[86] J. L. López-Presa, A. F. Anta, and L. N. Chiroque. Conauto-2.0: Fast isomorphism testing and automorphism group computation. arxiv preprint arxiv:1108.1060. 2011.

[87] N. Lord. Graph theory as I have known it, by W. T. Tutte (Book review). *The Mathematical Gazette*, 84(499):181–182, 2000.

[88] Z. Maksimovic. Extended version of On-line Encyclopedia of Integer Sequences A095854. `https://oeis.org/A075095/a075095.pdf`, 2004. Accessed: 2019-02-18.

[89] B. D. McKay. Transitive graphs with fewer than twenty vertices. *Mathematics of Computation*, 33(147):1101–1121, 1979.

[90] B. D. McKay. Combinatorial data. `http://users.cecs.anu.edu.au/~bdm/nauty/`, 1984. Accessed: 2018-10-23.

[91] B. D. McKay. Combinatorial data. `http://users.cecs.anu.edu.au/~bdm/data/graphs.html`, 1984. Accessed: 2018-10-23.

[92] B. D. McKay. Description of graph6, sparse6 and digraph6 encodings. `http://users.cecs.anu.edu.au/~bdm/data/formats.txt`, 2015. Accessed: 2018-10-23.

[93] B. D. McKay and G. F. Royle. The transitive graphs with at most 26 vertices. *Ars Combinatoria*, 30:161–176, 1990.

[94] B. D. McKay and E. Spence. Classification of regular two-graphs on 36 and 38 vertices. *Australasian Journal of Combinatorics*, 24:293–300, 2001.

[95] P. McWha. *Graduate course project.* unpublished, Department of Combinatorics and Optimization, University of Waterloo, 1973.

[96] H. Mélot. Facet defining inequalities among graph invariants: The system GraPHedron. *Discrete Applied Mathematics*, 156(10):1875–1891, 2008.

[97] M. Meringer. Fast generation of regular graphs and construction of cages. *Journal of Graph Theory*, 30(2):137–146, 1999.

[98] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|}|E|)$ algoithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, pages 17–27, New York, 1980. IEEE.

[99] S. Mitchell, M. O. Sullivan, and I. Dunning. *PuLP: a linear programming toolkit for python.* The University of Auckland, `http://www.optimization-online.org/DB_FILE/2011/09/3178.pdf`, 2011. Accessed: 2019-02-25.

[100] P. A. Morris. A catalogue of trees on n nodes, $n < 14$, mathematical observations, research and other notes, paper no. 1 sta (mimeographed). *Publications of the Department of Mathematics, University of the West Indies*, 1971.

[101] P. A. Morris. Self-complementary graphs and digraphs. *Mathematics of Computation*, 27:216–217, 1973.

[102] E. Neufeld. *Practical toroidality testing.* MSc thesis. University of Victoria, 1993.

[103] E. Neufeld and W. Myrvold. Practical toroidality testing. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, pages 574–580, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[104] S. D. Noble and D. J. A. Welsh. A weighted graph polynomial from chromatic invariants of knots. *Annales de l'Institut Fourier (Grenoble)*, 49(3):1057–1087, 1999.

[105] Oracle. MySQL 8.0. `https://www.mysql.com/`, 2018. Accessed: 2018-10-23.

[106] Oracle. MySQL 8.0. `https://dev.mysql.com/doc/refman/8.0/en/mysql-indexes.html`, 2018. Accessed: 2018-10-23.

[107] T. D. P. Peixoto. Graph-tool efficient network analysis. `http://graph-tool.skewed.de`, 2014. Accessed: 2019-02-01.

[108] A. Piperno. Search space contraction in canonical labeling of graphs. arxiv preprint arxiv:0804.4881. 2008. `https://arxiv.org/pdf/0804.4881.pdf`.

[109] G. Pólya and R. C. Read. *Combinatorial enumeration of groups, graphs, and chemical compounds.* Springer-Verlag, Berlin, Heidelberg, 1987.

[110] P. Potočnik. A list of 4-valent 2-arc-transitive graphs and finite faithful amalgams of index (4,2). *European Journal of Combinatorics*, 30(5):1323–1336, 2009.

[111] P. Potočnik, P. Spiga, and G. Verret. Bounding the order of the vertex-stabiliser in 3-valent vertex-transitive and 4-valent arc-transitive graphs. arxiv preprint arxiv:1010.2546. 2010. `https://arxiv.org/abs/1010.2546`.

[112] P. Potočnik, P. Spiga, and G. Verret. A census of 4-valent half-arc-transitive graphs and arc-transitive digraphs of valence two. arxiv preprint arxiv:1310.6543. 2013. `https://arxiv.org/pdf/1310.6543.pdf`.

[113] P. Potočnik, P. Spiga, and G. Verret. Cubic vertex-transitive graphs on up to 1280 vertices. *Journal of Symbolic Computation*, 50:465–477, 2013.

[114] L. Rabern. A note on Reed's conjecture. *SIAM Journal on Discrete Mathematics*, 22(2):820–827, 2008.

[115] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, 2002.

[116] R. C. Read. *The production of a catalogue of digraphs on 5 nodes*. Report UWI/CCl, Computing Centre, University of the West Indies, 1973.

[117] R. C. Read. A survey of graph generation techniques. In *Combinatorial Mathematics VIII*, pages 77–89, Editor: K. L. McAvaney. Berlin, Heidelberg, 1981.

[118] R. C. Read and D. G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977.

[119] R. C. Read and G. F. Royle. Chromatic roots of families of graphs. *Graph Theory, Combinatorics, and Applications*, 2:1009–1029, 1991.

[120] R. C. Read and R. J. Wilson. *An Atlas of Graphs*. Oxford University Press, 1998.

[121] B. Reed. Paths, stars and the number three. *Combinatorics, Probability and Computing*, 5(3):277–295, 1996.

[122] B. Reed. ω, δ, and χ. *Journal of Graph Theory*, 27(4):177–212, 1998.

[123] K. B. Reid and E. Brown. Doubly regular tournaments are equivalent to skew Hadamard matrices. *Journal of Combinatorial Theory, Series A*, 12(3):332–338, 1972.

[124] N. Robertson and P. D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[125] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7:425–440, 1986.

[126] G. F. Royle. *Constructive Enumeration of Graphs*. PhD thesis, University of Western Australia, 1987.

[127] G. F. Royle. Combinatorial Catalogues. `http://staffhome.ecm.uwa.edu.au/~00013890/`, 2007. Accessed: 2018-10-23.

[128] G. F. Royle, M. Conder, B. D. McKay, and P. Dobscányi. Cubic symmetric graphs (The Foster Census). `http://staffhome.ecm.uwa.edu.au/~00013890/remote/foster/`, 2013. Accessed: 2019-02-25.

[129] L. Schietgat, J. Ramon, and M. Bruynooghe. A polynomial-time maximum common subgraph algorithm for outerplanar graphs and its application to chemoinformatics. *Annals of Mathematics and Artificial Intelligence*, 69(4):343–376, 2013.

[130] C. P. Schnorr. Optimal algorithms for self-reducible problems. In *Proceedings of the 3rd International Colloquium on Automata, Languages and Programming*, ICALP'76, pages 322–337. Edinburgh University Press, 1976.

[131] Y. Shi, M. Dehmer, X. Li, and I. Gutman. *Graph Polynomials*. Chapman and Hall/CRC, 2016.

[132] C. C. Sims. Computational Methods in the Study of Permutation Groups. *Computational problems in abstract algebra, proceedings of a conference held at Oxford University*, pages 169–183, 1970.

[133] C. C. Sims. Computation with Permutation Groups. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, SYMSAC'71, pages 23–28, New York,USA, 1971.

[134] M. S. Sio. *Towards an Online Graph Atlas for Graph Theory*. BCompSc Hons dissertation. Clayton School of Information Technology, Monash University, 2010.

[135] D. A. Spielman. Faster isomorphism testing of strongly regular graphs. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 576–584, Philadelphia, 1996. ACM.

[136] P. G. Tait. Remarks on the colouring of maps. *Proceedings of the Royal Society of Edinburgh*, 10(729):501–503, 1880.

[137] O. S. Tezer. SQLite vs MySQL vs PostgreSQL: A Comparison of Relational Database Management Systems. `https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems`, 2014. Accessed: 2018-10-23.

[138] A. D. Thomas and G. V. Wood. *Group Tables*. Shiva Publishing Ltd., Orpington, 1980.

[139] C. Thomassen. The zero-free intervals for chromatic polynomials of graphs. *Combinatorics, Probability and Computing*, 6(4):497–506, 1997.

[140] C. Thomassen. Chords in longest cycles. *Journal of Combinatorial Theory, Series B*, 129:148–157, 2018.

[141] B. Toft. On critical subgraphs of colour-critical graphs. *Discrete Mathematics*, 7(3-4):377–392, 1974.

[142] B. Toft and M. Stiebitz. The Theorem of R. L. Brooks. `https://www.rs.tus.ac.jp/egawa_60th_birthday/slide/invited_talk/Bjarne_Toft.pdf`, 2013. Accessed: 2019-02-21.

[143] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265, 1937.

[144] W. T. Tutte. A family of cubical graphs. *Mathematical Proceedings of the Cambridge Philosophical Society*, 43(4):459–474, 1947.

[145] W. T. Tutte. A ring in graph theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 43:26–40, 1947.

[146] W. T. Tutte. A contribution to the theory of chromatic polynomials. *Canadian Journal of Mathematics*, 6(80-91):3–4, 1954.

[147] W. T. Tutte. On the symmetry of cubic graphs. *Canadian Journal of Mathematics*, 11:621–624, 1959.

[148] S. H. Unger. Git–a heuristic program for testing pairs of directed line graphs for isomorphism. *Communications of the ACM*, 7(1):26–34, 1964.

[149] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[150] P. Vismara and B. Valery. Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. *Communications in Computer and Information Science*, 14:358–368, 2008.

[151] E. W. Weisstein. Coxeter graph. `http://mathworld.wolfram.com/CoxeterGraph.html`. Accessed: 2019-01-19.

[152] D. B. West. *Introduction to Graph Theory*. Prentice Hall Inc., 1996.