Generation of Graph Classes with Efficient Isomorph Rejection

S. Narjess Afzaly

A thesis submitted for the degree of
Doctor of Philosophy of
The Australian National University

August 2016

Declaration

Except where otherwise indicated, this thesis is my own original work.

S. Narjess Afzaly
25 August 2016

To my dearest grandmother,

Ashraf

(Seyedeh Masoumeh GhadimiNejhad)

Acknowledgments

Foremost, I would like to express my sincere gratitude to my supervisor Professor Brendan McKay for the continuous support of my PhD study and research, for his patience, care, enthusiasm and immense knowledge. I am extremely grateful for everything I learned from him and for the insight I gained working with him. In addition, I would like to thank the rest of my thesis committee members, Professor Weifa Liang, Dr Pascal Schweitzer, and especially Dr Scott Morrison for the guidance, for what I have learned working with him, and for his enthusiasm.

My acknowledgment also goes to Professor Ian Wanless, Dr Catherine Greenhill, Professor Tomasz Luczak, Professor Piotr Sankowski and Professor Andrzej Ruciński for providing me the opportunity to visit their respective departments. I am grateful to Dr Beáta Faller, Mr Michał Dębski, Dr David Penneys, Dr Vikram Kamat, Professor Stanisław Radziszowski, Dr Erika Skrabulakova, Dr Joanna Polcyn-Lewandowska and Professor Vladimir Modrák, for their insightful discussion.

I thank my parents, Ali and Nahid, my grandmother, Ashraf, and my two brothers, HamidReza and MahmoudReza, for supporting me throughout my life and encouraging me to take the path of knowledge. I also thank my friends Sanam, Fiona, Ian, Bishan, Anila, Rani, Sahar, David, Pierre, Sara, Richa, Quifen, Edward, Samuel, Khoi-Nguyen, Ehsan, Azadeh, Jessie, Soheila, Soraya and other friends at CSIT and University House for their priceless support.

I appreciate the services ANU offers to students especially through University House, PARSA, CHELT, Counselling Centre and ANU Security. I am thankful for the support and services provided by the great staff members at CSIT, especially Mrs Janette Rawlinson. I also acknowledge the places where I wrote a big portion of my thesis: Telstra tower, Max Brenner, Grill'd, YHA, and ANU Canberra and Kioloa Coastal Campus in Australia, Cafe Artist, Cafe Lemon and Hotel Shabestan in Rasht, and the trains around Poland.

Abstract

In this thesis, efficient isomorph-free generation of graph classes with the method of generation by canonical construction path(GCCP) is discussed. The method GCCP has been invented by McKay in the 1980s. It is a general method to recursively generate combinatorial objects avoiding isomorphic copies. In the introduction chapter, the method of GCCP is discussed and is compared to other well-known methods of generation. The generation of the class of quartic graphs is used as an example to explain this method. Quartic graphs are simple regular graphs of degree four. The programs, we developed based on GCCP, generate quartic graphs with 18 vertices more than two times as efficiently as the well-known software *GENREG* does.

This thesis also demonstrates how the class of principal graph pairs can be generated exhaustively in an efficient way using the method of GCCP. The definition and importance of principal graph pairs come from the theory of subfactors where each subfactor can be modelled as a principal graph pair. The theory of subfactors has applications in the theory of von Neumann algebras, operator algebras, quantum algebras and Knot theory as well as in design of quantum computers. While it was initially expected that the classification at index $3 + \sqrt{5}$ would be very complicated, using GCCP to exhaustively generate principal graph pairs was critical in completing the classification of small index subfactors to index $5\frac{1}{4}$.

The other set of classes of graphs considered in this thesis contains graphs without a given set of cycles. For a given set of graphs, $\mathcal{H}$, the Turán Number of $\mathcal{H}$, $ex(n, \mathcal{H})$, is defined to be the maximum number of edges in a graph on n vertices without a subgraph isomorphic to any graph in $\mathcal{H}$. Denote by $EX(n, \mathcal{H})$, the set of all *extremal graphs* with respect to n and $\mathcal{H}$, i.e., graphs with n vertices, $ex(n, \mathcal{H})$ edges and no subgraph isomorphic to any graph in $\mathcal{H}$. We consider this problem when $\mathcal{H}$ is a set of cycles. New results for $ex(n, \mathcal{C})$ and $EX(n, \mathcal{C})$ are introduced using a set of algorithms based on the method of GCCP. Let $\mathcal{K}$ be an arbitrary subset of $\{C_3, C_4, C_5, \ldots, C_{32}\}$. For given n and a set of cycles, $\mathcal{C}$, these algorithms can be used to calculate $ex(n, \mathcal{C})$ and extremal graphs in $Ex(n, \mathcal{C})$ by recursively extending smaller graphs without any cycle in $\mathcal{C}$ where $\mathcal{C} = K$ or $\mathcal{C} = \{C_3, C_5, C_7, \ldots\} \cup \mathcal{K}$ and $n \leq 64$. These results are considerably in excess of the previous results of the many researchers who worked on similar problems.

In the last chapter, a new class of canonical relabellings for graphs, *hierarchical canonical labelling*, is introduced in which if the vertices of a graph, G, is canonically labelled by $\{1, \ldots, n\}$, then $G \setminus \{n\}$ is also canonically labelled. An efficient hierarchical canonical labelling is presented and the application of this labelling in generation of combinatorial objects is discussed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph theory is a branch of combinatorics that has drawn wide attention from mathematicians and computer scientists over the last two centuries. Since a graph is just an abstraction of a network and it can be interpreted in different ways, applications of graph theory appear in different areas such as transportation, chemistry, nanotechnology, bioinformatics, computing theory, parallel computing and social networks. Many new applications arise constantly that show the importance of research in this field.

Given some structural properties of some combinatorial objects (objects with discrete structures), exhaustive generation is listing all combinatorial objects with those properties. This process is also called structure enumeration. Exhaustive generation is interesting for both theoretical and practical purposes. The history of humans making lists of structures with given properties starts hundreds of years before Christ. Theaetetus determined the complete list of regular polyhedra in 400 B.C. Exhaustive generation has been far more advanced with the invention of computers. Many algorithms are invented to enumerate or generate different combinatorial structures. Some of these structures such as subsets, permutations, partitions and trees have nice recursive decompositions that translate into efficient generation algorithms without producing equivalent copies. A survey on generating these combinatorial objects can be found in [124, 185]. For combinatorial objects with equivalence classes, we may wish to list members of equivalence classes of combinatorial objects. In that sense, generating a class of combinatorial objects is constructing an exhaustive non-isomorphic (non-equivalent) list of the objects in the class. An important example is generating unlabelled graphs under the equivalence relation of isomorphism. For many classes of unlabelled graphs, the isomorphism problem and avoiding equivalent copies during the generation process tends to be difficult. Although Polya theory [185] can be used to count unlabelled graphs, it does not help much in generating the graphs.

The generation of graphs and providing exhaustive catalogs of different graph classes is also an indispensable tool for computerized investigations in graph-theoretical

research. It can be used in a search for counterexamples, verifying conjectures, refining proofs or discovering new conjectures. On the other hand, the flexibility with which most combinatorial objects can be modeled by graphs has meant that efficient programs for generating graphs can also be used to study and enumerate a variety of other combinatorial structures, including the Latin rectangles and block designs. Therefore, the problem of generating graphs has been considered by many authors.

Apart from mathematics, the generation of graphs can be also useful in industry and in different scientific disciplines such as chemistry [181, 62], physics [199], quantum physics [148] and crystallography [61] where the problem can be modelled by graphs. Modeling problems in terms of graphs and generating the relevant class of graph in search for the best solutions is a fast and inexpensive method to further the results and push the boundaries of knowledge to the point where the real experiments can't reach. For example, in computer science, by generating the relevant class of graphs, all the digital circuits of a given structure and size can be studied. As another example, in chemistry, all the compounds with a given molecular formula, or all the stereoisomers of a given molecule can be found using graph generation and these lists can be used to determine or predict the structure of molecules. Many graph generators were designed specifically for their application in chemistry [98, 99, 127, 137, 100]. In fact, generation of some classes of graphs has been initiated by chemists. For example, the class of cubic (3-regular) graphs is a class in which chemists have a great interest in. This class has been enumerated for the first time in 1889, up to 10 vertices, by Jan de Vries [59, 60]. This enumeration has been expanded over the time by other chemists and mathematicians using hand and later on computers [19, 49, 109, 177, 47, 48, 160, 35, 186, 42].

In addition to specific classes of graphs, researchers have worked on developing fast generators that can produce all graphs [97, 145]. These software packages have the options to restrict some specifications of the generated graphs, such as number of vertices, number of edges or the girth of the graphs. Different classes of graphs can be produced by filtering the graphs generated that are not in the considered class. However, for many classes, this is not an efficient way of generation especially when most of the generated graphs must be ruled out. Therefore, it is commonplace to develop specific generators for interesting classes of graphs.

There are several general techniques for developing algorithms that list equivalence classes, particularly equivalence classes that arise from symmetry conditions. Two most commonly used techniques are called *orderly generation* and *generation by canonical construction path*(GCCP). These approaches are not based on comparison for discarding the isomorphic copies. Therefore, they are time and storage efficient. In the first method, objects are only accepted in their canonical form while in the latter,

objects are only accepted if they are generated in a canonical way. We explain and compare these two methods of generation in the next section. GCCP is the method we used in this thesis to generate different classes of graphs. We also introduce a new version of the orderly generation at the end of the thesis.

In general, because of the isomorphism problem, the time for all generation algorithms can grow dramatically fast in terms of the number of vertices. The method by which isomorphic copies are avoided is one of the main criteria that determines the efficiency of a generation algorithm. Goldberg [95] introduces an iterative orderly generation algorithm by adding a vertex of maximum degree at a time and provided a polynomial time upper bound per generation of one graph. Some other methods avoid isomorphic copies by defining a canonical labelling. A canonical labelling of graphs is a way to define a unique labelled representative graph for each isomorphism class. Deciding whether two given graphs are isomorphic is a fundamental problem in graph theory. There seems to be a strong bond between graph generation and isomorphism algorithms. At the end of this thesis, we introduce a new class of canonical labellings and demonstrate how it can be used in generation of graphs. Solutions to the graph isomorphism problem are also used to study and enumerate a variety of other combinatorial structures, including distance-regular graphs [43], strongly regular graphs [81], block designs [57], Latin squares [159], partial geometries [190], and integer programs [174]. McKay has introduced isomorphism algorithms that are practically very fast. These algorithms are implemented in a software package, *nauty* [147]. Besides different applications in mathematical research, these algorithms have been used by several authors in a wide range of scientific disciplines such as particle physics [172], digital circuits [106], condensed matter physics [46], operations research [142, 180], semiconductors [200], error-correcting codes [173], neural networks [108], computer architecture design [14, 56], materials engineering [45, 201], artificial intelligence [11], event simulation [136], bioinformatics [107], computer security [125], and cryptology [52].

1.1 Definitions and Preliminaries

A *graph* is a pair $G = (V, E)$ where V is the set of *vertices* of G and E is the set of *edges* of G. In an undirected graph, E is a subset of the set of all 2-element subsets of V. The *order* and the *size* of a graph are defined respectively to be the number of its vertices and edges. The *degree* of a vertex of a graph is the number of edges incident to that vertex. A *regular* graph is a graph where the degree of all vertices is the same. If every vertex of a graph has degree r, then we say the graph is regular of degree r, or simply *r-regular*. 3-regular and 4-regular graphs are called *cubic* and *quartic* graphs respectively.

Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. G' is a *subgraph* of G if we have $V' \subseteq V$ and $E' \subseteq E$. If G' is a subgraph of G, then G a *supergraph* of G'. This is denoted by $G' \subseteq G$. Also, we say G' is a subgraph of G induced by V', if G' is a subgraph of G and for all $u, v \in V'$, we have $\{u, v\} \in E'$ iff $\{u, v\} \in E$.

A non-empty graph $P = (V, E)$ is called a *path* and is denoted by $P = x_0x_1...x_k$ if $V = \{x_0, x_1, ..., x_k\}$ and $E = \{x_0x_1, x_1x_2, ..., x_{k-1}x_k\}$ where x_is are all distinct vertices. If $P = x_0...x_{k-1}$ is a path and $k \geq 3$, then the graph $C = P + x_{k-1}x_0 = x_0...x_{k-1}x_0$ is called a *cycle*, The length of a cycle is its number of edges (or vertices). A cycle of length k is called a *k-cycle* and is denoted by C_k. The cycles C_3, C_4 and C_5 are called *triangle*, *quadrilateral* and *pentagon*, respectively. A cycle is odd (even) if its length is odd (even). A graph is called *cyclic* if it has at least one cycle as a subgraph. The minimum length of a cycle contained in a graph, G, is called the *girth* of G and is indicated by $g(G)$.

A graph $G = (V, E)$ is called *r-partite* if V admits a partition into r classes such that every edge has its ends in different classes, i.e, vertices in the same partition class must not be adjacent. 2-partite graphs are called *bipartite*. It is well-known that a graph is bipartite iff it does not contain any odd cycle.

In a labelled graph with n vertices, the vertices can be labelled by $\{1, 2, \ldots, n\}$. An adjacency matrix of a labelled graph, G, is an $n \times n$-binary-matrix, A_G, where $A_G[i, j] = 1$ if and only if vertices i and j are adjacent in G. We say two labelled graphs are *identical* if their adjacency matrices are the same.

Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. We say G and G' are *isomorphic* and write $G \simeq G'$, if there exists a bijection $\varphi : V \to V'$ where for all $u, v \in V$, we have $\{u, v\} \in E$ if and only if $\{\varphi(u), \varphi(y)\} \in E'$. In this case, φ is called an *isomorphism* from G to G'. A bijection, $\sigma : V \to V$, from the vertex set of G to itself is called an *automorphism* of G if it takes G to itself, that is if for all $u, v \in V$, we have $\{u, v\} \in E$ if and only if $\{\sigma(u), \sigma(y)\} \in E$. The set of all automorphisms of G is called the automorphism group*automorphism group* of G. The automorphism group of G deduces a partition, π, of the vertex set of G, called the *orbit partition* of G where each part (or cell) of π is called an *orbit* of the automorphism group of G or simply an orbit of G. Two vertices of G, say $u, v \in V$ are in the same orbit iff there is an automorphism of G that takes u to v.

A canonical labelling is a function that fixes one representative for each isomorphism class of labelled graphs. More precisely, a *canonical labelling* is a function, C, that takes a labelled graph, G, to a labelled graph, $C(G)$, called *canonical isomorph* of G, where:

- $C(G)$ is isomorphic to G, and
- For any graph, H, isomorphic to G we have $C(H) = C(G)$

We say a graph is a canonical isomorph, canonical, or *canonically labelled*, by a given canonical labelling if the graph is identical to its image under that canonical labelling.

1.2 Methods of Generation

There are several methods to generate combinatorial objects [124, 185]. When it comes to unlabelled graphs, generation is exhaustively listing pair-wise non-isomorphic labelled graphs in a given class, that is generating one labelled representative for each unlabelled graph in the class. Since the graph isomorphism problem tends to be difficult, the time for all isomorph-free graph generation algorithms can grow dramatically fast in terms of the number of vertices. Therefore, the efficiency of a generation method can strongly depend on the efficiency of the method applied to avoid isomorphic copies.

Methods of generation differ based on the algorithms they use to generate each object and the method they apply to avoid isomorphic copies. Recursive generation algorithms are more advanced methods to generate graphs. In these methods larger objects (*children*) in a graph class are generated from smaller objects (*parents*) in that class via some well-defined operations(*extensions*). These parent-child relations define a collection of search trees where each node is a graph. The root of these generation trees are the graphs with no parent; the graphs that can not be extended from any other graph in the class. These graphs are called *irreducible* graphs of the class. It is important to rule out the isomorphic copies from these trees. The methods we discuss in this section are all based on generation trees. The main difference between these methods comes from their different approaches to avoid isomorphic copies of the graphs in the class. In the following sections, we explain the most commonly used methods. There are, of course, other generation methods that do not fall exactly in a method mentioned here, for example, the methods used in [15, 36].

1.2.1 The Method of Naive Generation

In the method of naive generation, we use a comparison-based approach to discard isomorphic copies. The basic idea is to store a list of non-isomorphic generated graphs. Hence, this method is also called the *recorded objects method*. When a graph is generated it is compared to all graphs in this list. If it is isomorphic to one of these graphs, then it is discarded. Otherwise, we add the newly generated graph to the list. The advantages of this method is its simplicity. But this method is not efficient for producing a large class of graphs because a considerable amount of memory is required to store the list of non-isomorphic generated graphs. Also, checking if there is an isomorphism between two given graphs can be very complicated. Therefore, this method becomes very inefficient for larger classes of graphs, performing a pair-wise isomorphism checking between any two generated graphs. Moreover, since this method does not use an

effective pruning technique, lots of isomorphic intermediate nodes (possibly all nodes on the search tree) can be produced.

In variations of this method, the running time and the storage required can be reduced using heuristic algorithms and efficient data structures. Nonetheless, to have much more efficient generation methods, we need to avoid pairwise isomorphism checking. In the following sections, we discuss methods with non-comparison-based approaches to discard isomorphic copies. These methods have more complicated algorithms but they perform more efficiently in terms of the running time. Also with these methods, it is not required to store the generated graphs. Hence, they are extremely storage-efficient.

1.2.2 Generation by Canonical Representatives (Orderly Generation)

The method of generation by canonical representatives is the first general method of isomorph-free generation of graphs and combinatorial objects. It was introduced in the late 1970s by Read [182] and Faradzev [79, 78], independently. The main idea is to generate only the canonically labelled graph in each isomorphism class. Hence, for each class of isomorphism, any generated isomorph is discarded unless it is the canonically labelled one.

Orderly generation is a variation of this method where canonical graphs are defined recursively. The main idea in the orderly algorithms is to carefully pick the representative of each equivalence class and the order in which larger objects are constructed out of smaller objects to guarantee the canonical objects can be found easily via backtracks on the search trees. This method can be modified to increase the efficiency. Different classes of graphs are generated by variations of this method [55, 54, 64, 163, 35]. For example, Goldberg [95] considers the use of structure information in the orderly generation approach and introduces an iterative orderly generation algorithm by adding a vertex of maximum degree at a time and provided a polynomial time upper bound per generation of one graph. Examples of software packages using variations of this method of generation are *Minibaum* developed by Brinkmann to produce cubic graphs, and *GENREG* developed by Meringer to produce regular graphs.

1.2.3 Generation by Canonical Construction Path

The method of generation by canonical construction path (GCCP) was invented by McKay in the late 1980s. This method is also known as the method of *canonical augmentation* [125] because, in this method, many objects can be made by augmenting smaller objects that can be made from even smaller objects recursively. Thus, one can start from some subset of primitive objects such that any other object can be constructed from at least one of the primitive objects by repeatedly augmenting the objects. This

way a parent-child relation between the objects is defined as well as a search tree based on these parent-child relations where each node is an object in the class. Note each graph can be made in different ways and hence, isomorphic copies can appear along the way on the search tree. That is two different nodes on the search tree may present isomorphic copies and hence, there can be multiple paths on the search tree from each object (on different nodes) to primitive objects. The basic idea to avoid producing isomorphic copies is to define a single canonical construction path for each object and then explore only the canonical paths. This way, from each isomorphism class only one member is accepted; the one that is generated in a canonical way (by canonical construction path). As opposed to the orderly generation, here an accepted object is possibly not canonically labelled. We define a reduction to be the inverse operation of an augmentation (extension), that is the operation that takes a graph to its parent. Primitive graphs with no parent are called *irreducible*. Any graph that is not irreducible, can be reduced in one or more ways via different reductions. In the GCCP method, we need to consider a function that takes a graph and returns a specific set of reductions for that graph, called the *canonical* or the *genuine reductions*, that are equivalent to each other under the action of the automorphism group. With these definitions, a generated graph is accepted only if it is generated with an augmentation (extension) whose inverse operation is a canonical (genuine) reduction.

Cubic graphs with up to 20 vertices [160] is the first class of graphs generated by GCCP. The generation method of GCCP is extremely versatile and has been successfully applied to many classes of graphs and other combinatorial objects including graphs with some hereditary property [146], cubic graphs [39, 160, 186, 42], Ramsey graphs [80, 154, 157, 149, 155, 158], hypergraphs [153], digraphs [105], chromatic-index-critical graphs [41] Latin rectangles and balanced incomplete block designs [156, 157, 152], structures of finite geometry [184], relational models [115] and unlabelled orders [104]. Examples of graph generation software based on this method are *Geng* [145] developed by McKay to produce graphs with given specifications, and *Snarkhunter* [42] developed by Brinkmann, Goedgebeur and McKay to generate cubic graphs and specific subclasses of cubic graphs.

The efficiency and applicability of this method can be improved via group theoretic approaches and permutation group computations. While some steps have been taken toward this end, this problem is yet to be fully studied.

In the following sections, we explain this method more precisely, compare it with the orderly method, and demonstrate how we applied this method to efficiently generate quartic graphs. More on this method and its mathematical basis can be found in [146].

1.3 GCCP and Generation of Quartic Graphs

In this section, we explain the method of generation by canonical construction path (GCCP) in more detail. This is the method we used to generate different classes of graphs including *quartic graphs*. We use the class of quartic graphs here as a sample to explain GCCP. A quartic graph, or a 4-regular graph, is a graph where the degree of each vertex is exactly four.

GCCP is a general method to recursively generate combinatorial objects in which larger objects, (*children*), are constructed out of smaller ones, (*parents*), by some well-defined operation, (*extension*). The inverse operation of extensions is called *reductions*. For each generated graph, a set of reductions, that are equivalent to each other under the action of the automorphism group, is defined to be genuine. To avoid isomorphic copies, each graph is extended only by non-equivalent extensions. Also after a graph, G, is generated, it is discarded unless it is generated by an extension whose inverse operation is a genuine reduction in G.

1.3.1 How Isomorphic Copies are Avoided

The formal proof of GCCP being exhaustive and isomorph-free can be found in [146]. Here, we give a little insight to an inductive proof. The induction is an invariant, $\mathcal{O}(G)$ of a graph, G, that increases during an extension. We call this invariant the order of the graph. In the literature, the term order is usually used for the number of vertices of the graphs that is different from the definition we used in this section. However, for some extensions, this invariant can also be the number of vertices. As such is the extension we present later on for generation of quartic graphs. Suppose for some fixed integer, k, the GCCP process produces no more than one isomorph in each isomorphism class of order at most k. We claim that the generation process produces no more than one isomorph in each isomorphism class of order $k+1$.

Suppose for the sake of contradiction that G and G' are two isomorphic generated graphs of order $k+1$ that are extended from their parents, P_1 and P_2, by extensions x_1 and x_2 respectively. Let r_1 and r_2 be the reductions that are inverse operations of x_1 and x_2, respectively. The reductions r_1 and r_2 can not be equivalent to each other under the action of automorphism group, otherwise x_1 and x_2 would be equivalent and P_1 and P_2 would be isomorphic. But this is impossible because by the induction hypothesis, P_1 and P_2 can not be two separately generated isomorphic graphs and if they are exactly one generated graph, then x_1 and x_2 can not be equivalent by the definition of GCCP. Therefore, r_1 and r_2 are non-equivalent reductions. And since G and G' are isomorphic, by definition of GCCP, only one of r_1 and r_2 can be genuine. Thus, G and G' can not be both generated and accepted.

1.3.2 Generation of Quartic Graphs

The generation of regular graphs and their subclasses up to isomorphism is one of the oldest problems in constructive combinatorics. Various algorithms for the generation of regular graphs with different degrees of regularity have been invented. Beezer and Riegsecker [25] described an algorithm to generate regular graphs based on which they produced all regular graphs up to 13 vertices. Meringer [164] introduced an efficient algorithm to generate regular graphs with given number of vertices and vertex degree. The method is based on a variation of the orderly generation. The program [162] developed based on this method, allowed the generation of all regular graphs of degree 5 on 16 vertices for the first time. This program can be restricted to generate graphs with a given girth.

The generation of cubic (3-regular) graphs and its subclasses has been studied by several researchers [59, 60, 19, 49, 109, 177, 47, 48, 160, 35, 186, 40, 38, 42]. A fast generator for cubic graphs designed in recent years can be found in [37]. This program can also be restricted to generate cubic graphs with girth at least 4 or 5.

Quartic, 4-regular or 4-valent graphs are simple graphs of degree four. This class of graphs contains the smallest regular graphs of an even degree apart from the trivial case of two-regular graphs that only contains cycles. Toida studied [194] the generation of the quartic graphs in the 1970s and later, de Vries [59] enumerated small quartic graphs. Generating quartic graphs has also been considered while studying the generation of larger classes of graphs [51, 63, 163]. In 1992, a complete list of regular graphs on 13 vertices including quartic ones was provided by Beezer and Riegsecker [25]. Meringer in [164] also generated quartic graphs on 18 vertices.

In 2001, Menon and her supervisor, McKay [161] developed some algorithms, based on GCCP method, to produce quartic and bipartite quartic graphs up to 14 and 20 vertices respectively. Nevertheless, these algorithms were never published. We have developed programs with a parallelisation option based on the same algorithms to produce a complete list of non-isomorphic connected, bi-connected, and not necessary connected quartic graphs with a given number of vertices. These programs are time and storage efficient. They can produce quartic graphs up to 18 vertices more than two times as efficiently as the well-known software *GENREG* [162] does. The definition of the extension and the reduction we used are as follows:

The Extension

In [146], an upper object is defined to be a sub-object containing the information needed to get from an object to one of its children, that is basically the information that determines an extension. To generate quartic graphs we define an extension to be removing two disjoint edges, adding a new vertex and joining it to the four endpoints

of the two removed edges. Therefore, each pair of disjoint edges in a graph is an upper object of the graph.

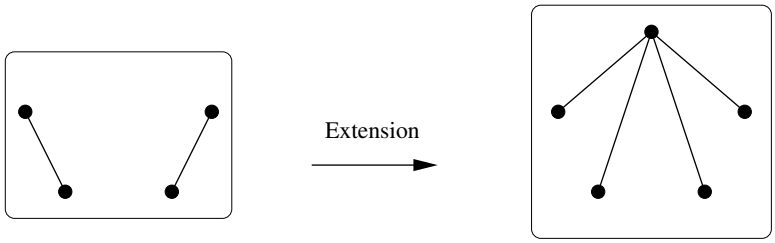

Figure 1.1: An extension: Removing two disjoint edges, adding a new vertex and joining it to the four endpoints of the two removed edges.

The Reduction

In [146], a lower object is defined to be a sub-object that contains the information needed to get from a graph to one of its parents, that is the information that determines a reduction. To generate quartic graphs we define a reduction to be deleting a vertex and adding two extra disjoint edges between its neighbours without producing multiple edges. Therefore, not every vertex can be removed by a reduction. To define our reduction more precisely, we first define the lower objects, *dovi*s. A dovi is a subgraph of a child that is reduced to two disjoint edges during the reduction. It consists of 5 vertices; one vertex as the base, b, and its four neighbors. It can be considered as $(b, \{\{v_1, v_2\}, \{u_1, u_2\}\})$ where:

- v_1, v_2, u_1 and u_2 are the 4 neighbours of the base, b,
- There is no edge between v_1 and v_2 and
- There is no edge between u_1 and u_2

A dovi, $(b, \{\{v_1, v_2\}, \{u_1, u_2\}\})$, can be reduced by deleting b and its four incident edges, and adding the two edges $\{v_1, v_2\}$ and $\{u_1, u_2\}$.

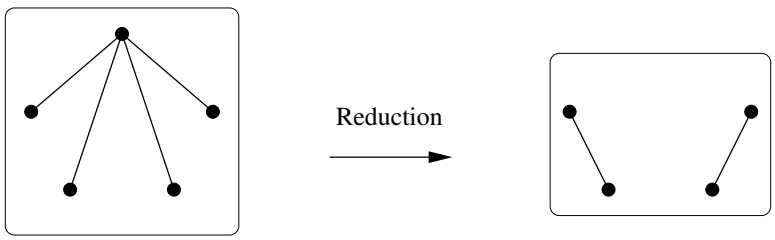

Figure 1.2: A reduction: Deleting a vertex and adding two extra disjoint edges between its neighbours without producing multiple edges.

Note that, each vertex can be the base of at most three different dovis and therefore, can serve as the base to at most three different reductions.

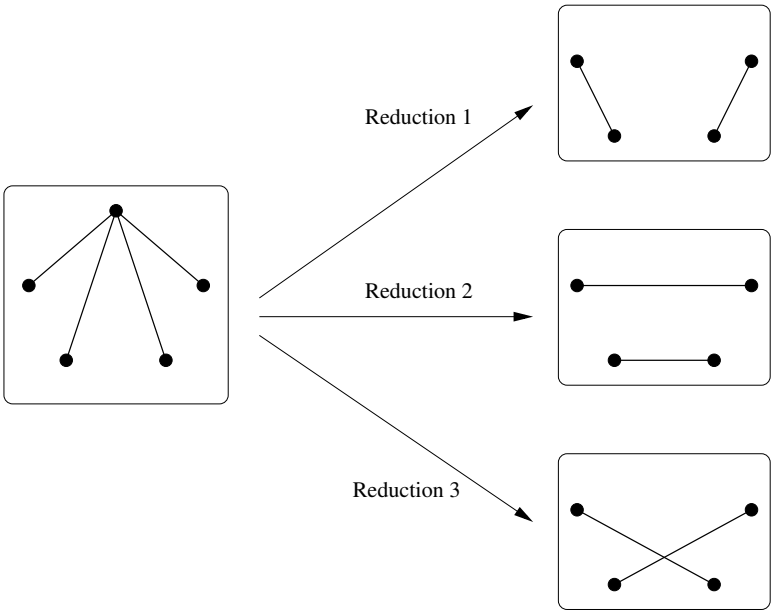

Figure 1.3: A dovi can be reduced in at most three different ways.

The Genuine Reduction

A genuine reduction is the one that reduces a winning dovi. A *winning* dovi, $d = (b, \{\{v_1, v_2\}, \{u_1, u_2\}\})$, is the one with the lexicographically largest 3-tuple $(x_0(d), x_1(d), x_2(d))$ where:

- $x_0(d) = f(b)$ and f is a function of some combinatorial invariants of b.
- $x_1(d)$ is calculated based on a certain combination of $f(v_1)$, $f(v_2)$, $f(u_1)$ and $f(u_2)$.
- $x_2(d)$ is the highest rank of a dovi in the same orbit as d, in the list of dovis sorted lexicographically based on the canonical labelling of their vertices calculated by *nauty*.

The Irreducible Graphs

Clearly, the set of quartic graphs is closed under the extension we defined. But there are quartic graphs that can not be generated by extending any other quartic graph. These are the ones that have no dovi as an induced subgraph and hence can not be reduced to a parent. Therefore, these quartic graphs are called *irreducible graphs*.

Theorem 1.3.1 determines the structure of the irreducible graphs. We used the idea in this theorem as the building block to construct all irreducible quartic graphs of a given order. Before stating this theorem, we first present some relevant definitions and lemmas.

Definition 1. *We say a vertex, b, in a quartic graph, G, is removable if there exist vertices v_1, v_2, u_1 and u_2 such that:*

- *v_1, v_2, u_1 and u_2 are adjacent to b in G, and*
- *$\{v_1, v_2\}, \{u_1, u_2\} \notin E(G)$.*

Otherwise b is irremovable.

In other words, a vertex is irremovable iff it is not the base to any dovi.

Definition 2. *A quartic graph, G, is irreducible if all its vertices are irremovable. Otherwise, it is reducible.*

Definition 3. *We define $K_5 - \{e\}$ to be the graph obtained by deleting an edge from K_5.*

Definition 4. *A vertex is trapped if it is a vertex of degree four in one of the three graphs shown in Figure 1.4.*

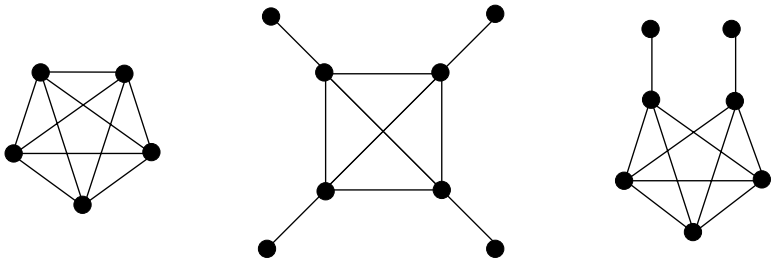

Figure 1.4: A vertex is *trapped* if it is a vertex of degree four in a graph in this picture.

We use the following lemmas to prove Theorem 1.3.1.

Lemma 1.3.1. *If $G_1 = K_4$ is an induced subgraph of a quartic graph, G, then any vertex of G_1 is irremovable.*

Proof. Let b be a vertex of a G_1. Since its three neighbours in G_1 are adjacent together, b is irremovable. □

Lemma 1.3.2. *Let $G_1 = K_4$ be an induced subgraph of a quartic graph, G. If a vertex, b, is adjacent to exactly two vertices of G_1, then b is removable.*

Proof. Let v_1, v_2, v_3 and v_4 be the four vertices of G_1 where b is adjacent to v_1, v_2 and two other vertices, u_1 and u_2, not in G_1. Since each of the vertices v_1 and v_2 is adjacent to b and has three other neighbours in G_1, they can not be adjacent to any other vertex. Therefore, there is no edge between v_1 and u_1 and no edge between v_2 and u_2. Hence $\{v_1, u_1\}$ and $\{v_2, u_2\}$ are not in $E(G)$ and b is removable. □

Theorem 1.3.1. *A quartic graph, $G = (V, E)$ is irreducible iff its vertex set can be partitioned into disjoint subsets such that each subset induces a subgraph isomorphic to a K_5, a K_4 or a $K_5 - \{e\}$.*

Graphs $G_1 = K_5$, $G_2 = K_4$ and $G_3 = K_5 - \{e\}$ are depicted in Figure 1.5.

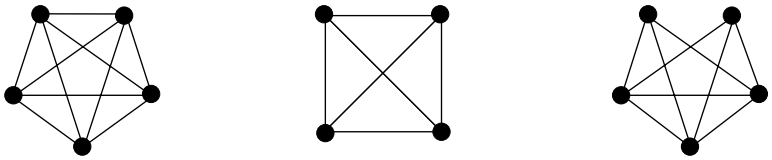

Figure 1.5: A quartic graph is irreducible iff its vertex set can be partitioned into disjoint subsets such that each subset induces a subgraph isomorphic to one of the graphs G_1, G_2 or G_3.

Proof. If V, the vertex set of graph G, can be partitioned into disjoint subsets where each subset induces a subgraph isomorphic to a K_5, a K_4 or a $K_5 - \{e\}$, then it is easy to verify every vertex of G is contained in a K_4. So by lemma 1.3.1, all the vertices are irremovable and hence G is irreducible.

To prove the other direction of the theorem, we only need to show if G is irreducible, then every vertex of G is trapped. Let u be an arbitrary vertex of G and v_1, v_3, v_3 and v_4 be the four neighbours of u. We define G_u to be the subgraph induced by u and its neighbours. Since u is irremovable, all of the three following statements holds:

- G contains at least one of the edges $\{v_1, v_2\}$ and $\{v_3, v_4\}$,
- G contains at least one of the edges $\{v_1, v_3\}$ and $\{v_4, v_2\}$, and
- G contains at least one of the edges $\{v_1, v_4\}$ and $\{v_3, v_2\}$.

Therefore, G_u must have a subgraph isomorphic to either H_1 or H_2 in Figure 1.6. We consider each of these two cases separately. Without loss of generality, we assume that the vertices of these subgraphs are labeled as shown in Figure 1.6.

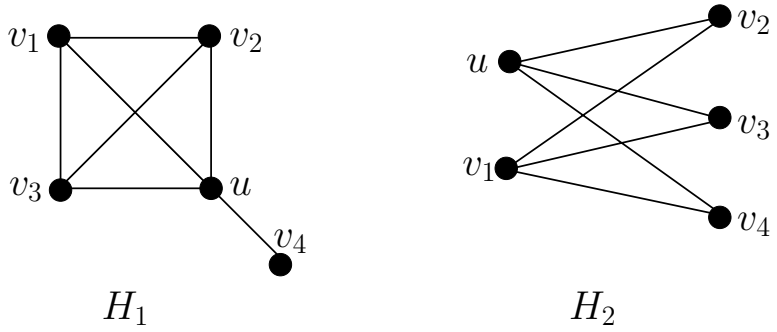

Figure 1.6: For each vertex, u, in an irreducible graph, G, the induced subgraph, G_u, must have a subgraph isomorphic to the graphs H_1 or H_2

We follow the proof by considering two cases:

Case 1: If H_1 is a subgraph of G_u: Let X be a subgraph of G induced by $S = \{v_1, v_2, v_3, u\}$ and let N be the set of vertices not in S that are adjacent to vertices in S. Note that $X = K_4$, $v_4 \in N$ and $1 \leq |N| \leq 4$.

- $|N| = 1$: then $N = \{v_4\}$ and v_4 is adjacent to all vertices of X. Hence G_u is isomorphic to K_5 and u is trapped.
- $|N| = 2$: then $N = \{v_4, w\}$ where w is not in G_u. To have the vertices v_4 and w irremovable, by Lemma 1.3.2, each of them must be adjacent to either exactly one or exactly three vertices of X. Without the loss of generality, we consider the case where v_4 is adjacent to exactly one vertex of X. The vertices v_4 and w can not be adjacent otherwise v_4 is removable. Therefore, w has a neighbour, x, which is not in H_1 and hence u is trapped.
- $|N| = 3$: then one vertex in N must be adjacent to exactly two vertices of X and by lemma 1.3.2, such a vertex is removable. Therefore, this case can not happen.
- $|N| = 4$: then it is easy to see u is trapped.

Case 2: If H_2 is a subgraph of G_u: Let $S = \{v_2, v_3, v_4\}$ and X be the subgraph in G induced by S. We have $0 \leq |E(X)| \leq 3$

- $|E(X)| = 0$: then v_2 is adjacent to vertices w_1 and w_2 in $V(G) \setminus V(G_u)$ and since there is no edge between v_1 and w_1 and no edge between u and w_2, the vertex v_2 is removable. Therefore, this case cannot happen.
- $|E(X)| = 1$: without loss of generality, we assume $E(X) = \{\{v_3v_4\}\}$.Hence, v_2 is adjacent to vertices w_1 and w_2 in $V(G) \setminus V(G_u)$ and similar to the case of $|E(X)| = 0$ the vertex v_2 is removable. Therefore, this case cannot happen either.

- $|E(X)| = 2$: suppose $E(X) = \{\{v_2v_3\}, \{v_3v_4\}\}$. Note that, in this case, v_2 and v_4 cannot share a neighbour, $w \in V(G) \setminus V(G_u)$, otherwise w is removable. Therefore, there must be two different vertices $w_1, w_2 \in V(G) \setminus V(G_u)$ where v_2 is adjacent to w_1 and v_4 is adjacent to w_2. Now, it is easy to see u is trapped.

- $|E(X)| = 3$: then $G_u = K_5$ and u is trapped.

□

An example of an irreducible quartic graph is shown in Figure 1.7. It is easy to verify this graph satisfies the conditions inZ Theorem 1.3.1.

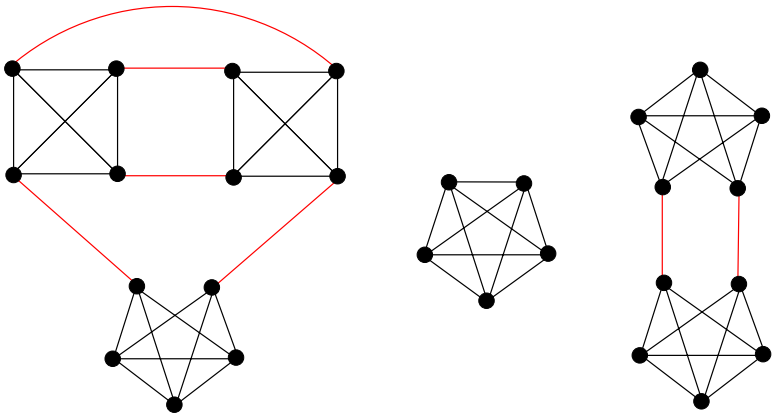

Figure 1.7: An example of an irreducible quartic graph.

1.3.2.1 Results

The programs developed during this research are time and storage efficient for generating all, connected and biconnected quartic graphs. Biconnectivity testing costs less than extra 50 % of the total time. These programs produce connected quartic graphs up to 18 vertices about 2 times more efficiently as the well-known software *GENREG* does.The program *GENREG* is developed by Meringer and generates the connected regular graphs [162, 164]. Table 1.1 demonstrates these results.

Number of Vertices	Number of QGs	Number of Connected QGs	Time/ Biconnected QGs (s)	Time/ connected QGs (s)	Time/ all QGs (s)	*GENREG* (s)	*GENREG* vs GCCP
12	1547	1544	0.020	0.021	0.021	0.028	1.33
13	10786	10778	0.080	0.071	0.070	0.120	1.69
14	88193	88168	0.590	0.432	0.430	0.930	2.15
15	805579	805491	5.290	3.700	3.691	8.500	2.29
16	8037796	8037418	54.370	36.792	36.717	84.610	2.29
17	86223660	86221634	611.840	404.880	404.410	924.760	2.28
18	985883873	985870522	8405.280	4823.910	4814.590	10803.88	2.23

Table 1.1: Running time of the programs developed based on the GCCP for producing quartic graphs (QG). The column before last contains the running time of *GENREG* for producing connected quartic graphs and the last column demonstrates the ratio of running time of *GENREG* to running time of GCCP for connected quartic graphs

1.3.3 Variations of GCCP

As we mentioned, GCCP has been used for generation of different combinatorial objects and graph class. Different extensions can be used in a GCCP method, such as adding at each step, a vertex, an edge or a specific sub-object such as dovis in our generation of quartic graphs. The choice of an extension to generate a given class of objects can be made based on the properties of that class in a way to have the generated objects remained in the class or to minimize the number of generated objects that must be ruled out. Also, a proper choice of an extension tries to maximise the efficiency of the generation process.

The method of GCCP we described in the earlier sections is sometimes called the generation by *strong* canonical construction path (strong canonical augmentation) as opposed to the other variation of the GCCP, weak canonical augmentation where calculating the automorphism group is avoided. In this variation, an object is extended in every possible way. Hence, equivalent extensions can be applied to each object and the isomorphic children can be generated. However, similar to the strong variation, after an object is generated, it is rejected if it is not generated by an extension whose inverse operation is not canonical. This means any two generated isomorphic copies are essentially siblings. Therefore, applying isomorph rejection only amongst siblings is sufficient to have an isomorph-free generation. In some situations, this version of GCCP can have more advantages.

Although GCCP traverses the search tree in a DFS fashion but in order to make our algorithm more efficient or useful under some circumstances, we can change the way the search tree is being traversed. For example, the search tree can be traversed in a combination of DFS and BFS fashion.

1.3.4 Parallelisation in GCCP

Since in GCCP no past memory is required, the computation can be distributed across a processor farm as follows:

All processors build the generation tree up to a certain level, L, and then each of them chooses a certain subset of nodes at level L and continues the generation process building only the subtrees rooted from those nodes. Examples of this parallelisation to generate cubic graphs can be found in [186]. The program we developed to generate quartic graphs can also apply this parallelisation technique.

1.4 Orderly Generation Versus GCCP

In both methods, the orderly generation and GCCP, larger graphs (children) are generated from smaller ones (parents) by applying some well-defined operations(extensions).

For a class of graphs, consider the search tree based on the parent-child relation deduced by extensions where each node of the tree is a labelled graph. The tree may contain isomorphic or even identically labelled graphs. It is necessary to prune this tree, so that from each class of isomorphism exactly one representative is output. The *GCCP* and the orderly methods have different approaches to prune this tree though they accept the very same number of nodes, that is the number of non-isomorphic graphs in the class. In the orderly method, these accepted representatives are the canonically labelled ones while in *GCCP*, the accepted representatives are those generated in a canonical way. We compare the two methods, *GCCP* and the orderly generation, in terms of their applications and efficiency as follows:

Applications of the Orderly Generation and GCCP

In the orderly generation, the set of canonical isomorphs of the graphs in the class to be generated must be closed under the considered extension. This means the extensions must be defined in a way that any canonically labelled graph in the class, apart from the irreducible ones that can be easily generated, is an extension of another canonically labelled graph in the class. Hence, the type of extensions and the canonical labellings must be compatible in this sense. Also, in this method, parents must be induced subgraphs of their children. Thus, in the orderly method, the limitations on the eligible extensions restrict the classes of graphs that can be generated in practice with the orderly method. In contrast, the *GCCP* can be applied to a wider range of extensions and no strict correlation is required between the types of extensions and the genuine reductions. Therefore, this method can be applied to a wide range of classes of graphs. In fact, almost any class of combinatorial objects for which an inductive construction process exists can be generated with this method.

Efficiency of the Orderly Generation and GCCP

While in the rest of the thesis, the generation tree and the search tree are used interchangeably, in this section, we consider them to be different. We define the search tree, to be the tree consists of all generated graphs, accepted and rejected ones while the generation tree only contains the accepted generated graphs. With these definitions, the generation tree is a subtree of the search tree consisting of only the accepted nodes. Note that, for a given class of graphs, the generation trees obtained by GCCP and the orderly generation have the same number of nodes that is the number of unlabelled graphs in that class. In GCCP, due to avoiding the equivalent extensions, the search trees, for many cases, are smaller than the corresponding ones in the orderly method. In those cases, the efficiency is increased in GCCP by generating fewer nodes. Although having smaller search trees in GCCP is at the cost of time spent on calculating the automorphism group to determine equivalent extension, one should

note that such calculations are not needed for the terminal nodes on the generation tree. Therefore, when the terminal nodes are a large portion of the generation tree, the GCCP can perform very efficiently. However, sometimes when the generated graphs are less symmetric, the search tree in GCCP is not much smaller than the one in the orderly method and hence the time spent on calculating the automorphism group may not pay for itself, specially when the terminal nodes are not a large portion of the generation tree. In this case, the orderly method may perform more efficiently than the GCCP. However, the weak variation of GCCP mentioned in Section 1.3.3 can serve more efficiently than the strong variation in such cases by avoiding calculating the automorphism group at cost of some isomorphism checking.

On the other hand, in the orderly generation, when a graph is generated, the process of deciding whether to accept or reject the graph is usually much less efficient and requires the computationally expensive calculation of the canonical labelling while by exploiting the flexibility of the GCCP in accommodating different definitions for genuine reductions and providing lookaheads, the computational cost of determining whether a generated graph must be rejected or accepted, can be dramatically reduced.

In the last chapter, a new method of generation, the *natural orderly generation*, is introduced that combines the benefits of the two methods we compared in this section.

1.5 Nauty; The Automorphism Group and a Canonical Labelling

To calculate the automorphism group, orbits of the automorphism group and the canonical labelling of graphs, we use the practically efficient software package *nauty* (No AUTomorphisms, Yes?). *Nauty* developed by McKay, is a set of procedures that in addition to efficiently determining the automorphism group of a vertex-coloured graph, implements one of the most powerful and the best known algorithms for graph isomorphism problem by providing canonically labelled isomorph of labelled graphs. Graph isomorphism problem is a fundamental problem in graph theory and is of a special interest in complexity theory. It is among the few problems that are known to be in the complexity class NP but not known whether it is solvable in polynomial time or it is NP-complete. The time complexity of this problem is known to be at most $exp\big((\log n)^{O(1)}\big)$ for graphs with n vertices [17, 16] while it can be solved in polynomial time for certain classes of graphs [18, 31, 82, 140, 165]. The complexity of this problem is discussed more extensively in [122, 143, 198].

In *nauty*, the orbits are obtained from the automorphisms of the graph and the automorphisms are calculated by a backtracking algorithm over the search tree of labellings of the graph. The basic idea of the algorithm is to build a search tree where each node is an ordered partition of the vertex set of the input graph and each leaf is

a discrete ordered partition that presents a relabelling of the input graph deduced by the order of the vertices in the discrete partition. Automorphisms of the input graph are found by noticing that two such relabelling give identical graphs. Let π_1 and π_2 be two relabellings that take the input graph to two identical graphs, then $\pi_1(\pi_2)^{-1}$ is an automorphism of the input graph. Thus, a complete set of generators for the automorphism group of the input graph can be deduced from the leaves of the search tree. The building block of the algorithm is that at each node, we take a partition of the vertices and try to divide a non-trivial cell at each step based on the number of neighbours of vertices in each cell. This process continues until the partition can not be furthered divided in such a way. At this stage, a cell, C, of the partition is chosen and for each vertex, $v \in C$, we branch the search tree and divide C into $C \setminus \{v\}$ and $\{v\}$. The canonical labelling of the input graph is then defined to be the relabelling, only among the relabellings deduced by the leaves of this search tree, that gives the greatest isomorph in the total ordering defined by Read and Faradzev [79, 182]. Note that, *nauty* also uses graph theoretical information to reduce the set of potential isomorphs from which the canonical isomorph is chosen.

The high efficiency of *nauty* is due to the clever pruning algorithm traversing the search tree. It uses some group calculations and the graph automorphisms that are found along the way to prune the search tree. Although *nauty* has exponential running time on some input graphs, it performs exceptionally well under most circumstances. The algorithms used in *nauty* are discussed in more detail in [103, 144, 147]. The complexity analysis of these algorithms is discussed in [167]. Examples of classes of graphs where the time complexity behaviour of this algorithm is polynomial or exponential can be found in [167, 90]. Over the years, substantial advances have been made to *nauty*, especially in the area of data structures. It can now process nontrivial graphs with more than a million vertices.

Traces developed by Piperno [178] is another program for determining the automorphism group of a vertex-coloured graph, and isomorphisms between graphs. Combining this program with *nauty* [151], a higher range of graphs can be covered for efficient calculation of the automorphism group and the canonical labelling. The package *nauty and Traces* is available at McKay's website [150]. It contains a set of tools suitable for processing files of graphs such as the *dreadnaut* program that provides sufficient functionality for simple purposes. *Nauty* can be also called within other programs for more complicated applications.

1.6 Overview of the Thesis

In this thesis, efficient isomorph-free generation of graph classes with the method of GCCP is discussed. In the current chapter, we explain this method and demonstrate how this method can be used to generate quartic graphs. The results are summarised

in Table 1.1 that shows the program based on GCCP generates quartic graphs with 18 vertices more than two times as efficiently as the well-known software *GENREG*.

The second chapter of this thesis—exhaustive generation of principal graph pairs—uses the art of generation of graphs to extend the classification of some mathematical objects known as *subfactors* in von Neumann algebra. The application of this work arises in design of quantum computers, quantum geometry and similar fields.

The third chapter—the Turán numbers for cycles—catalogues all small Turán graphs for collections of short cycles using advanced generation algorithms. The results are considerably in excess of the previous results of the many researchers who worked on similar problems.

And the last chapter—a hierarchical canonical labelling for graphs and its application in the generation of graphs—defines a new class of canonical labellings and introduces a canonical labelling in this class. There is a close relation between generation of graphs, graph isomorphism problem and the canonical labelling of graphs. The last chapter demonstrates how we can use the new canonical labelling to introduce a new method of generation that combines the advantages of the orderly generation and the method of GCCP.

Chapter 2

Exhaustive Generation of Principal Graph Pairs

2.1 Abstract

This chapter presents the exhaustive generation of principal graph pairs using the method of canonical construction path. The definition and importance of principal graph pairs come from the theory of subfactors where each subfactor can be modelled as a principal graph pair. The theory of subfactors has applications in the theory of von Neumann algebras, operator algebras, quantum algebras and knot theory as well as in design of quantum computers. It was by exhaustive classifications of subfactors at small index that the least understood quantum symmetries were discovered. While it was initially expected that the classification at index $3 + \sqrt{5}$ would be very complicated, using canonical construction path to exhaustively generate principal graph pairs was critical in completing the classification of small index subfactors to index $5\frac{1}{4}$ [9], which is beyond the first interesting composite index, $3 + \sqrt{5}$.

This is joint work with Scott Morrison and David Penneys but in this chapter, only the contribution of the author is mentioned. Morrison and Penneys have independently developed another generator for PGPs. The results of the two generators are compared and observed to be consistent.

2.2 Introduction

The generation of all principal graph pairs (PGPs) with a limit on the maximum eigenvalue of the graphs can help with the classification of subfactors that has applications in the theory of von Neumann algebras, the theory of subfactors and fusion categories. A subfactor is an inclusion $N \subset M$ of von Neumann algebras with trivial centres.

Subfactor standard invariants encode quantum symmetries. The classification of

small index subfactors is an essential part of the search for exotic quantum symmetries. A quantum symmetry is a non-commutative analogue of the representation category of a finite group. Since our knowledge of quantum symmetries is still primitive, understanding the range of examples is essential. But there are several instances of quantum symmetries that do not come from the basic examples. Indeed, the least understood of all known quantum symmetries were discovered in exhaustive classifications of subfactors at small index [102].

Jones [116] and Ocneanu [171] provided the full classification of subfactor standard invariants with index of at most 4 while many of the details are presented by others [96, 29, 110, 111, 120, 117]. Popa [96, 179, 113] developed the classification of subfactor at index exactly 4. Next, Haagerup classified principal graphs up to index $3+\sqrt{3}$ [102] with detail provided in [30, 13, 28]. Using some number theory-based techniques [50] and by applying the obstructions introduced by [118, 189, 175], the classification of subfactors was extended to index $5 = 3+\sqrt{4}$ [170, 168, 112, 176, 114, 119]. Unfortunately, with these techniques, the classification can not be extended efficiently to index larger than 5. In spite of the expectation that the classification at index $3+\sqrt{5}$ would be very complicated, this classification is recently provided in [138] and some progress is given towards the classification up to index $3+\sqrt{5}$ in [169], and up to index $6\frac{1}{5}$ in [139].

The generation of PGPs helped to complete the classification of small index subfactors beyond $3+\sqrt{5}$, which is the first interesting composite index, that is, a product of smaller allowed indices. A PGP is a combinatorial object consisting of two bipartite graphs with a duality function on the vertices satisfying some conditions. For each subfactor there is an associated PGP and each PGP can be associated to at most finitely many distinct subfactors. However, there are some combinatorial obstructions for a PGP to be associated to a subfactor.

We have developed a program to exhaustively generate PGPs. The algorithm used in this program is based on the generation method of canonical construction path, which is time and storage efficient for discarding the isomorphic copies. The high efficiency of the program enables producing PGPs with higher index limits compared to the previous attempts and hence helps with extending the classification of subfactors of small index [9] completing the classification of subfactor standard invariants to index $5\frac{1}{4}$.

2.2.1 Subfactors and Principal Graph Pairs

Given a PGP, an algebraic problem can be formulated with a one-to-one correspondence between the solutions of this problem and the (hyperfinite) subfactors associated with that PGP. This process is called *finding a flat connection* on a graph pair. Unfortunately, these algebraic problems are, in general, intractable. But on the other hand, for each subfactor, there is an associated PGP. While no combinatorial condition is known defining a bijection between subfactors and PGPs, there are several obstructions or

combinatorial conditions for a PGP to be associated to a subfactor. Although not satisfying these obstructing conditions is not sufficient for a PGP to be associated with a subfactor, it can nonetheless increase the probability to a good extent. Therefore, it is commonplace to enumerate all possible PGPs, satisfying certain combinatorial constraints and decide which generated PGPs can not be associated with a subfactor using some other techniques including the standard number theoretic approach based on [50]. The classification can then be completed considering only the remaining cases.

Unfortunately enumerating PGPs above index 4 gets difficult quickly. The generation methods used thus far to generate PGPs produce isomorphic copies of PGPs and removing the corresponding redundancies in the generation tree via pairwise isomorphism checking was unrealistically computationally expensive. In this chapter, we explain how we applied the method of Generation by Canonical Construction Path [146] and incorporated a number of obstructions into the algorithm to enumerate PGPs potentially associated to subfactors. The result played an important role in furthering the classification of subfactors [9]. More information on subfactors and PGPs can be found in [9].

2.3 Preliminaries

2.3.1 Definitions

In this section, we present the definitions we consider throughout this chapter. In order to define the class of PGPs we aimed to generate, we first define the following:

A *principal graph pair* (*PGP*), $P = (G_1, G_2, D)$ consists of two bipartite graphs, G_1 and G_2, and a duality function, D, where:

1. Each graph G_1 and G_2 has a number of distinct special vertices where there is a path between any other vertex in the graph to one of these special vertices. Each of these vertices is called a *starred* vertex or *root*. Each graph of a PGP is bipartite with its roots in the same part of bipartition. Also, each graph of a PGP has at least one root. In fact, in most cases, each graph of a PGP has exactly one root and that is the assumption in a number of the theorems and definitions stated in this chapter, however, they can be easily modified to accommodate multiple roots.

2. For each vertex, v, in a graph of a PGP, we define, *level[v]* to be the distance of v from the root (or the smallest distance of v from one root in case of multiple roots). The vertices of each graph are in fact, partitioned based on their level. We say v is *at level l* if it is at distance l from the root (or l is the smallest distance between v and one of the roots). Level 0 only consists of the roots of the graph, all the neighbours of a root are at level 1, and the level of other vertices can be defined inductively. There are no edges between the vertices at the same level.

3. The duality, D, is an involution of the vertices of the two graphs defining a correspondence between the vertices where:

   - If $u = v$ then u is a *self-dual* vertex. Otherwise $\{u, v\}$ is a pair of dual vertices.
   - The duality function preserves distance from the roots. That is if the vertices of u and v are dual of each other then they are at the same level.
   - If u is at an even level, then the dual of u is in the same graph as u.
   - If u is at an odd level, then the dual of u is not in the same graph as u. Hence there is no self-dual vertex at an odd level.

As we explained in Section 2.2, there are a number of obstructions for a PGP to be associated to a subfactor. One of these obstructions can be stated as a condition called *associativity*. A PGP satisfying the condition of associativity is called *associative*.

According to this condition for any vertex v, the multiset of the duals of the neighbours of the duals of the neighbours of v is equal to the multiset of the neighbours of the duals of the neighbours of the dual of v. That is, if S is a multiset containing a subset of vertices in one graph, and $D[S]$ and $N[S]$ are the multisets consist of duals and neighbours of vertices in S, respectively, then the associativity condition says $D[N[D[N[\{v\}]]]] = N[D[N[D[\{v\}]]]]$ for all vertices v in either of the two graphs. The associativity condition can be restated as follows:

> For any two vertices, v and u, the number of ways to get from v to u by following an edge, taking the dual, following an edge from there and taking the dual again, is the same as the number of ways to get from v to u by taking the dual, following an edge, taking the dual, and following an edge from there.

We define a *dual object* to be a self-dual vertex or a pair of dual vertices in a PGP. The *norm* of a PGP is the maximum of the largest eigenvalue of the adjacency matrix of either of the two graphs and the *index* of a PGP is the square of the norm of that PGP.

The adjacency matrices of the two graphs of an associative PGP have the same largest eigenvalues. The *depth of a graph* of a PGP is the largest level of a vertex in the graph and the *depth of a PGP*, P is the largest depth of its two graphs and is denoted by $depth_P$. The associativity condition compels the difference between the depth of the two graphs of a PGP to be at most one.

As a matter of fact, in PGPs, edges are weighted by positive integers. In this generality, the associativity condition is modified to respect edge multiplicities. However, the indices of PGPs rapidly increase with weights larger than one. Thus, the PGPs with weights larger than one that are within the index limit are rare and can be managed by hand. Therefore, for the sake of simplicity, in our construction, we assumed

the edges are not weighted.

2.3.2 Data Structures and Notations

In figures throughout this chapter, PGPs are demonstrated by two graphs drawn horizontally one beneath the other where the vertices are placed left to right in increasing order of their levels. Hence the roots are the left-most vertices and vertices of the two graphs at the same level are aligned vertically.

At each odd level, the vertices of the second graph are arranged from top to bottom in the same order of their corresponding dual vertices. That is, for each odd level, the top most vertex in the first graph is the dual of the top most vertex in the other graph, the second top most vertex in the first graph is the dual of the second top most vertex in the other graph, and so on. At even levels, on the other hand, each self-dual vertex is indicated by a short red spike while a pair of dual vertices are joined by a red line. Figure 2.1 demonstrates the PGP of the Haagerup subfactor [12] where vertices 0 and 1 are roots of the two graphs. The pairs $\{2,3\}$, $\{6,7\}$, $\{12,13\}$, $\{14,15\}$ and $\{16,17\}$ are pair of dual vertices and all other vertices are self-dual.

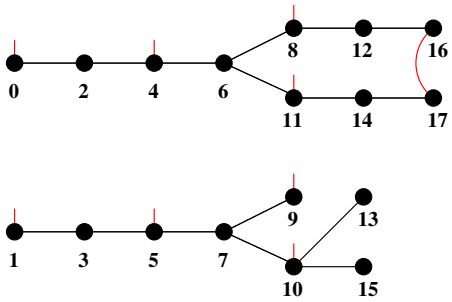

Figure 2.1: The PGP of the Haagerup subfactor [12]

2.3.2.1 Using Sparse Graph Data Structure to Store PGPs

Each PGP can be modeled as one single graph. Representing PGPs as single graphs facilitates the application of *nauty* in the generation process. The main data structure we used to produce and store PGPs is the *sparsegraph* format introduced by McKay. In this format, an adjacency list is stored for each vertex. A detailed description of this data structure can be found in [147]. We have designed efficient algorithms to convert graph pairs stored with this data structure to the common notation for PGPs as we explained above, and vice-versa. We show how to represent PGPs with single graphs using sparse graph data structure.

Consider a PGP consists of two bipartite graphs G_1 and G_2 with duality function D. We define a labelled graph G where $V(G) = V(G_1) \cup V(G_2)$, G_1 has n_1 roots labelled by $0, \ldots, n_1 - 1$, the graph G_2 has n_2 roots labelled by $n_1, \ldots, n_1 + n_2 - 1$, and $E(G) = E(G_1) \cup E(G_2) \cup E_D$ where $E_D = \{\{u, v\} | \, u \neq v$ and vertices u and v are dual of each other$\}$. The following theorem shows G_1, G_2 and D are reconstructable from (G, n_1, n_2).

Theorem 2.3.1. *Consider a PGP, P and the duality function D where P consists of graphs G_1 and G_2 with n_1 and n_2 roots, respectively. Let G be the graph constructed from G_1, G_2 and D as is explained above. The graphs G_1, G_2 and D and hence P, are reconstructable from (G, n_1, n_2).*

Proof. The roots of G_1 and G_2 can be determined just by their labels of $0, \ldots, n_1 - 1, n_1, \ldots, n_1 + n_2 - 1$. Since roots are self-dual vertices of degree one, each of them has only one neighbour in G that is their neighbour in G_1 and G_2, respectively. Hence the vertices at level one in G_1 and G_2 can be determined. These vertices are dual of each other and the edge between them in G shows the duality. We now inductively show how vertices at other levels G_1 and G_2 and the edges in G representing the duality can be determined by showing that for each i if the vertices at the level i and $i - 1$ are known, then the vertices at the level $i + 1$ and their duals can be determined.

For a vertex, u, at level i in G_1, consider the set of its neighbours in G. At most one of them can be at level i since there are no edges in G_1 or G_2, between the vertices at the same level. Therefore, if there is an edge in G, between u and a vertex, v where u and v are at level i in G_1 or G_2, then the edge $\{u, v\}$ does not exist in G_1 or G_2. This edge in G is, hence, representing the duality between u and v. The neighbours of u in G can be at level $i - 1$, i, or $i + 1$. And since by induction hypothesis the neighbours of u that are at level i and $i - 1$ are known, all the neighbours of u at level $i + 1$ can be determined. Since any vertex at level $i + 1$ in G_1 is adjacent in G to a vertex that is at level i in G_1, one can determine all vertices at level $i + 1$ by investigating the neighbours of all vertices at level i.

This way, while determining the set of vertices at each level and the neighbours of each vertex, all edges in G representing the duality are also determined. It is easy to see the graph obtained from G by removing these edges has two connected components, one is G_1 with roots labelled by $0, \ldots, n_1 - 1$ and the other one is G_2 with roots labelled by $n_1, \ldots, n_1 + n_2 - 1$. □

2.4 The Generation Algorithm

To produce all PGPs with indices below a given limit that correspond to subfactors, we produce a wider range of PGPs with indices below the given limit. We apply some pruning techniques to avoid generating a big portion of the PGPs which are not associative or their index is not within the given limit. Any PGP with at least two dual objects can be obtained from another PGP by deleting a dual object. Based on this

recursive definition, we define our extension to be adding a new dual object and the reduction to be removing a dual object, and we applied the method GCCP where each node of the generation tree is a PGP.

When generating a class of combinatorial objects by recursively extending smaller objects to construct larger ones, isomorphic copies appear in three ways :

- From the same parent and by extensions equivalent under the action of the automorphism group as depicted in Figure 2.2,
- From the same parent and by different extensions as depicted in Figure 2.3, and
- From different parents and by different extensions as depicted in Figure 2.4.

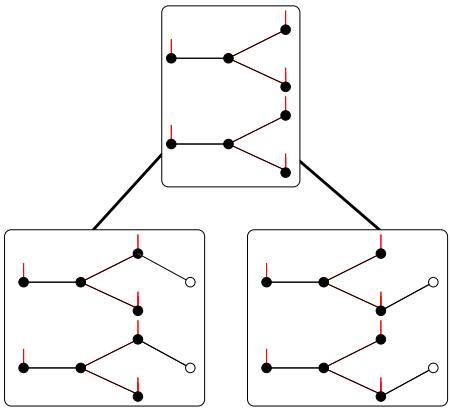

Figure 2.2: Isomorphic PGPs constructed from the same parent and by extensions equivalent under the action of the automorphism group .

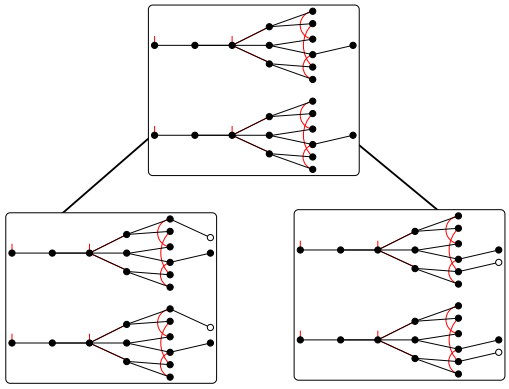

Figure 2.3: Isomorphic PGPs constructed from the same parent and by different extensions

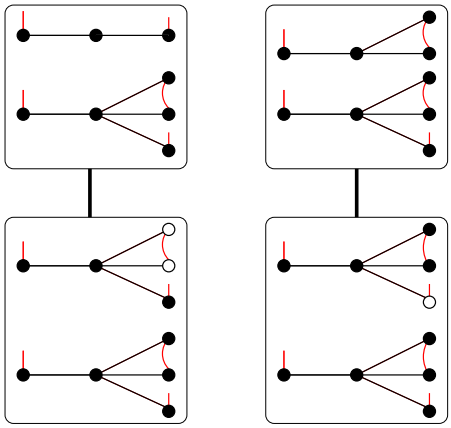

Figure 2.4: Isomorphic PGPs constructed from different parents and by different extensions.

The construction method of GCCP avoids isomorphic copies that can be generated by any of these three cases. To avoid the first case, in GCCP, for each PGP, we only apply extensions that are none-equivalent under the action of the automorphism group, and for this we need to calculate the automorphism group of each PGP. To avoid the last two cases, in GCCP method, we define genuine reductions and any generated PGP is rejected unless it is generated by an extension whose inverse operation is a genuine reduction. These are discussed in more detail in the following sections.

2.4.1 Extension

We define our extension to be adding one dual object to a PGP $P = (G, G', D)$ as follows:

- If an extension is adding a dual object at an odd level l, we insert a pair of dual vertices, one vertex to G and the other one to G', connecting each new vertex to some vertices at the previous level, $l-1$. This way we guarantee in the new PGP, as of the definition of a PGP, the dual of a vertex at an odd level is at the same level of that vertex but in a different graph.

- If an extension is adding a dual object at an even level, l, then the extension may add one self dual vertex or a pair of dual vertices and connects the self dual vertex or the pair of dual vertices to some vertices at level $l-1$ that are all in G or all in G'. This way we guarantee in the generated graph pair, the dual of a vertex at an even level is in the same graph and at the same level of that vertex and this matches to the definition of a PGP.

As it is explained in Section 2.3.2.1, we store a whole PGP as one graph. Therefore, when adding a pair of dual vertices, it is important to insert an edge between the new vertices to store the duality between them. Note that by having all neighbours of a new vertex at the same level, we guarantee each graph of the new PGP remains bipartite.

Given a PGP, each potential extension can be represented by the sets of neighbours of the one or the two vertices in the new dual object to be added to the PGP. To store a potential extension we used a data type called *setar*. As it is explained in Section 2.5.2 two subsets S_1 and S_2 of vertices of a PGP are stored in a setar $S = \{S_1, S_2\}$. Of course, S_1 and S_2 need to satisfy some conditions to define a valid prospective dual object and therefore, a valid potential extension. For example, the vertices in S_1 and S_2 must be all at the same level. When one of the subsets stored in S is empty, then S represents an extension where a self-dual vertex is added and is joined to the vertices in the non-empty subset in S. Otherwise S represents an extension where a pair of dual vertices is added and one vertex is joined to the vertices in S_1 and the other vertex is joined to the vertices in S_2.

Note that two extensions are equivalent when the neighbours of the prospective duals are equivalent, that is when their corresponding setars are equivalent. In the GCCP method, to avoid the isomorphic copies generated by the first case stated in Section 2.4, before extending a PGP, P, we first determine its automorphism group, $Aut(P)$, from which we calculate the orbits of the valid setars of P. And then for exactly one setar in each orbit, we extend P to a bigger PGP by adding a dual object and inserting regarding edges between the vertices in the dual object and the vertices in the representative setar.

2.4.2 Reduction

Based on the extensions we defined above, a reduction is the operation of deleting a dual object and the corresponding edges.

2.4.3 Genuine Reduction

According to the GCCP method, when we extend a PGP, X, to another PGP, Y, we only accept Y if it is constructed from X by an extension whose inverse operation is a genuine reduction. Here we define a reduction to be genuine if it reduces a winning dual object. Therefore, if X is extended to Y, we accept Y only if it is constructed from X by adding a winning dual object. This avoids the last two cases of generating isomorphic copies.

We define a dual object, d, to be winning if:

1. The vertices in d are at the last level of Y.
2. d is a pair of dual vertices unless there is no pair of dual vertices at the last level, i.e., d is a self-dual vertex only if there is no pair of dual vertices at the last level.
3. If d is a pair of dual vertices consisting of vertices u and v, then it has the largest total degree, $deg(u) + deg(v)$, among all other dual pairs at the last level. If d is a self-dual vertex u, then it has the largest degree, $deg(u)$, among all other self-dual vertices at the last level.
4. Let w be a vertex in a dual object, H, where w is labelled lexicographically last by the software *nauty* among all vertices that are part of a dual object that satisfies all of the conditions given above. Then one vertex of d is in the same orbit as w. That is:
   - When there is no dual pair of vertices at the last level, and H is a self-dual vertex with the largest canonical label among all self-dual vertices at the last level with the largest degree, then $d = \{u\}$ is in the same orbit as H.
   - When $d = \{u, v\}$ and $H = \{x, y\}$ are both pairs of dual vertices where x has the largest canonical label among all other vertices w_1 where $\{w_1, w_2\}$ is a pair of dual vertices at the last level with the largest total degree, then u or v is in the same orbit as x.

More precisely we define the canonical reduction by assign a 4-tuple $x(d) = (x_0(d), x_1(d), x_2(d), x_3(d))$ to each dual object, d, and the winning dual objects are those whose corresponding 4-tuples are lexicographically largest.

We aimed to define an efficient genuine reduction. Therefore, the values of x_0, x_1, x_2, x_3 are combinatorial invariants of increasing discriminating power and computational cost where:

- $x_0(d)$ is the level of the vertices of d,
- $x_1(d) = 0$ if d is a self-dual vertex otherwise $x_1(d) = 1$, and
- $x_2(d)$ is the degree of u when $d = \{u\}$ is a self-dual vertex, otherwise it equals to the total degree of u and v when $d = \{u, v\}$ is a pair of dual vertices.
- $x_3(d)$ is the largest label among the canonical labels of vertices in the same orbit as a vertex in d.

Each x_i is computed only if $(x_0(d'), \ldots, x_{i-1}(d'))$ is not the unique lexicographically largest with d' being the newly added dual object, that is, when the values of earlier elements of x fail to determine our newly added dual object is not winning while they also fail to distinguish it to be the unique winning dual object. Hence, if there are more than one dual objects with the largest value of (x_0, x_1, x_2) and our newly added dual object is one of them, then using *nauty*, we canonically label the vertices and calculate the orbits of the PGP, defining $x_3(d)$ to be the largest label among the canonical labels of vertices in the same orbit as a vertex in d.

Since the x_0, x_1 and x_2 are invariant under isomorphisms, the dual objects with vertices in same orbits of the automorphism group have the same values of (x_0, x_1, x_2). The value of x_3 has an even stronger property: the vertices of two dual objects are in same orbits if and only if they have the same value of x_3. This follows from the fact that for any two dual pairs of vertices $d_1 = \{u_1, v_1\}$ and $d_2 = \{u_2, v_2\}$, if the vertices u_1 and u_2 are in the same orbit then so are v_1 and v_2. Therefore, together with the definition of canonical labelling, we have the following lemma:

Lemma 2.4.1. *Let P_1 and P_2 be two isomorphic PGPs having at least two dual objects each and let γ be an isomorphism from P_1 to P_2. If d_1 and d_2 are, respectively, the dual objects in P_1 and P_2 having the largest 4-tuples x, then d_1^γ is a dual object in the same orbit as d_2 under the action of the automorphism group of P_2, $Aut(P_2)$. Furthermore, reducing d_1 in P_1 produces a graph isomorphic to the result of reducing d_2 in P_2.*

Although the generation algorithm would remain exhaustive and isomorph-free when only x_3 is computed, embedding x_0, x_1 and x_2 in the definition of winning dual objects is important for the efficiency purposes. While to compute the value of x_3, the orbits of the automorphism group and a canonical form must be computed, the earlier elements of the 4-tuples are only based on local properties of vertices that are much cheaper to compute. Furthermore, these properties provide lookaheads in the generation process that improve its efficiency. For example, when extending a PGP, P, by adding a new dual object to a level other than the last level, we already know that there will be other dual objects with larger x_0 and hence this new dual object is not going to be a winning in the new PGP. Therefore, our definition of the genuine reduction allows us to avoid the construction of a lot of children that would be rejected afterwards.

According to Lemma 2.4.1, any two isomorphic accepted PGPs are from the same parent and are generated by equivalent extensions. Since in the GCCP method, only extensions, that are not non-equivalent under the action of automorphism group, are applied to each PGP, no two isomorphic accepted PGPs can appear throughout the generation process. We use this fact to prove the following lemma where given a PGP r, the set $\mathcal{L}_r$ is the set of all labelled PGPs on the generation tree descendant from r in which the extension is adding a dual object. For each k, $\mathcal{L}_r^k$ is the set of all PGPs in $\mathcal{L}_r$ with k dual objects.

Lemma 2.4.2. *Let H_{n-1} be a set of PGPs consisting of exactly one representative for each isomorphism class of PGPs in $\mathcal{L}_r^{n-1}$. If for each PGP in H_{n-1}, the extensions of adding a dual object is applied to exactly one setar in each orbit of all valid setars, and a generated PGP in $\mathcal{L}_r^n$ is accepted if and only if the last added dual object has the lexicographically largest value of (x_0, x_1, x_2, x_3), then exactly one representative of each isomorphism class of PGPs in $\mathcal{L}_r^n$ is accepted.*

Proof. Let P_1 and P_2 be two isomorphic accepted graphs in $\mathcal{L}_r^n$. From Lemma 2.4.1, we know that P_1 and P_2 are generated from the same parent, P, in $\mathcal{L}_r^{n-1}$. Assume P_1 is extended from P by adding a dual object, d_1, and joining it accordingly to vertices stored in a setar, S_1, of P, and P_2 is also extended from P by adding a dual object, d_2, joining it accordingly to setar S_2 of P. According to Lemma 2.4.1, the newly added dual objects d_1 and d_2 must both have the lexicographically largest value of (x_0, x_1, x_2, x_3) in P_1 and P_2 respectively, otherwise P_1 or P_2 would not be accepted. Therefore there is an isomorphism, γ, from P_1 to P_2 that maps d_1 to d_2. This means the automorphism in P, deduced by γ, maps S_1 to S_2, showing that S_1 and S_2 are equivalent under the Automorphism group of P, $Aut(P)$ and that is in contrary to our procedure. Therefore no two isomorphic copies are accepted. Now, we prove the exhaustiveness, that is, for each isomorphism class of PGPs in $\mathcal{L}_r^n$, at least one representative is generated and accepted.

Let P_1 be an arbitrary PGP in $\mathcal{L}_r^n$. Consider a PGP, $P_2 \in \mathcal{L}_r^n$, that is isomorphic to P_1 in which the dual object, d containing the vertex labelled n is a winner. Let S be a setar in P_2 corresponding to d and P be the PGP obtained from P_2 by removing d. We have $P \in \mathcal{L}_r^{n-1}$. Therefore there should be $P' \in H_{n-1}$ isomorphic to P. Let γ be an isomorphism from P to P' and consider $S' = S^\gamma$. If P_3 is the PGP obtained from P' by adding a dual object, d' to S', then P_3 is isomorphic in P_2 and d' is a winning dual object in P_3. Hence P_3 is accepted if it is generated. On the other hand, if C is the set of all setars in P' that are equivalent to S' (C is the orbit of setars in P' containing S'), then P_3 is obtained from P' by any extension according to a setar in C and by assumption, P' is extended according to exactly one setar in C. Thus, P_3 is generated and accepted. This means $P_3 \in H_{n-1}$. Hence, we showed for any arbitrary $P_1 \in \mathcal{L}_r^n$, there is an isomorphic PGP, P_3 that is generated and accepted. □

Together, Lemma 2.4.1 and Lemma 2.4.2 give the following theorem:

Theorem 2.4.1. *When recursively applied, starting with a PGP, r, the algorithm described above constructs exactly one representative of every isomorphism class of PGPs in $\mathcal{L}_r$.*

An alternative proof that shows our generation algorithm is exhaustive and isomorph-free is presented in Section 2.6.

2.4.4 Pseudo-Code

Algorithm 1 takes a PGP, *root*, the number of vertices in the PGP, n_r, and some data embedded in a structure, *termin*, that stores the information regarding the terminating conditions including the maximum index and probably other conditions such as the maximum number of vertices or the maximum running time. It first checks if *root* is an eligible PGP by calling the function *Is_Good()* and then calls the function *P_Rec_Extend()* to generates all eligible PGPs within the limit specified in the termination conditions *termin*, that are rooted from *root* in the generation tree in the method of *generation by canonical construction path*. The functions *Is_Good()* and *P_Rec_Extend()* are presented in Algorithms 2 and 3.

Algorithm 1 GCCP_PGPs_Generation algorithm

```
1: procedure The GCCP_PGPs_Generation (PGP: root, int n_r, Struct: termin )
2:     if Is_Good(root, termin ) == false then      return
3:     end if
4:     P_Rec_Extend(root, n, termin, 1 )
5: end procedure
```

The function *Is_Good* as presented in Algorithm 2 takes a PGP, w, and a structure, *termin*, that stores the information regarding the terminating conditions including the maximum index and probably other conditions such as the maximum number of vertices or the maximum running time. It returns *false* when w satisfies one of the terminating conditions or when we can prove the subtree rooted from w on the generation tree does not contain any associative PGP. Otherwise this function returns *true*.

Algorithm 2 Is_Good algorithm

```
1: procedure Is_Good(PGP: w, Struct: termin )
2:     if w satisfies one terminating condition determined in termin then
3:         return false
4:     end if
5:     if it can be shown there is no associative PGP on the subtree rooted from w
   then
6:         return false
7:     end if
8:     return true
9: end procedure
```

Calling the procedure *P_Rec_Extend()*, recursively extends PGPs starting from a given PGP. This procedure is presented in Algorithm 3. It takes one PGP, w, the number of vertices, n, in that PGP, a structure, *termin*, that stores similar information as

given in Algorithm 1, and a boolean argument that presents whether w is considered the root of a search tree. If it is not, then the procedure first checks if w has been generated in a genuine way by calling the procedure *Is_Genuine()* depicted in Algorithm 4. It calls the function *Is_Good()* to check if w passes the filter of the termination conditions and satisfies some associativity tests. If w is associative, the function *Is_associative()* returns true and w is output. The procedure, then, considers all valid non-equivalent extensions of w and recursively extend each resulted PGP. The function *Change()* presented in Algorithm 5 modifies w to a new PGP according to a given extension.

Algorithm 3 P_Rec_Extend

```
1:  procedure P_Rec_Extend(PGP: w, int: n, Struct: termin , Boolean Is_root )
2:      if Is_root == false then
3:          if Is_Genuine(w, n) == false then      return
4:          end if
5:      end if
6:      if Is_Good(w, termin ) == false then      return
7:      end if
8:      if Is_associative(w) == true then
9:          add w to the output file
10:     end if
11:     make a list l of valid setars (upper objects)
12:     call nauty and calculate the orbits of setars in l
13:     for each orbit do
14:         choose one setar s as the representative
15:         w' = Change(w, n, s)
16:         let n' be the label of the last vertex in w'          ▷ n' = n + 1 or n + 2
17:         P_Rec_Extend(w', n', termin , 0)
18:     end for
19:      return
20: end procedure
```

The function *Is_Genuine()* as presented in Algorithm 4 takes a PGP, w, and an integer, n, that indicates the number of vertices in w which is the same as the label of the vertex last inserted to w during the generation process. This function checks if w has been generated in a genuine way from its parent by checking if the dual object, d, inserted last during the generation process (which is the dual object containing the vertex labelled n) is a winning dual object. That is, this function checks if d has the lexicographically largest 4-tuple, $x(d) = (x_0(d), x_1(d), x_2(d), x_3(d))$, among all dual objects in w. The definition of the 4-tuples $x() = (x_0(), x_1(), x_2(), x_3())$ is given in Section 2.4.3.

Algorithm 4 The Is_Genuine algorithm

```
1: procedure Is_Genuine(PGP: w, int: n)
2:     if dual[n] = n then                          ▷ The last inserted vertex is self-dual.
3:         if w contains a dual pair at the last level then
4:             return false
5:         end if
6:         if n is the unique vertex of the minimum degree at the last level then
7:             return true
8:         else
9:             if n is not a vertex of the minimum degree at the last level then
10:                return false
11:            else
12:                Let winningset be the set of all the vertices of the minimum degree at
    the last level
13:                if n is in the same orbit as the vertex in winningset with the largest
    canonical labelling calculated by nauty then
14:                    return true
15:                else
16:                    return false
17:                end if
18:            end if
19:        end if
20:    else                    ▷ dual[n] = n − 1 and the last inserted vertex is not self-dual .
21:        if {n, n − 1} is the unique dual pair with the minimum total degree at the
    last level then
22:            return true
23:        else
24:            if {n, n − 1} is not of the minimum total degree among dual pairs at the
    last level then
25:                return false
26:            else
27:                Let winningset be the set of all the vertices in a dual pair with mini-
    mum total degree at the last level
28:                if n or n − 1 is in the same orbit as the vertex in winningset with the
    largest canonical labelling calculated by nauty then
29:                    return true
30:                else
31:                    return false
32:                end if
33:            end if
34:        end if
35:    end if
36: end procedure
```

The function *Change()* as presented in Algorithm 5 takes a PGP, w, an integer, n, that indicates the number of vertices in w and a *setar*, s, that is a structure to store a potential extension and contains a subset or two subsets of vertices that can be valid neighbours of the vertices in a new dual object. Sections 2.4.1 and 2.5.2 contain more explanation about setars. The function *Change()* inserts one self-dual vertex of a pair of dual vertices to w and joins each new vertex to the vertices determined by s.

Algorithm 5 The Change algorithm

```
1:  procedure CHANGE(PGP: w, int: n, setar: s )
2:      w' = w
3:      Add vertex n + 1 to w', join it to the vertices in the first field of the setar.
4:      if the second field of the setar is empty then
5:          dual[n + 1] = n + 1 in w'
6:      else
7:          Add vertex n + 2 to w', join it to the vertices in the second field of the setar.
8:          dual[n + 1] = n + 2 in w'
9:          dual[n + 2] = n + 1 in w'
10:     end if
11:     return w'
12: end procedure
```

Note that although these algorithms give the basic idea of our generation process, our C code is implemented slightly differently.

2.5 Efficiency

The key point of our program being more efficient in producing PGPs, compared to the previous efforts, is the choice of the generation algorithm. In the previous efforts isomorphic copies were avoided by running the isomorphism check between the generated PGPs which is unrealistically computationally expensive as the size and number of PGPs grow. In contrast, as it is extensively discussed in other chapters, the method of Generation by Canonical Construction Path produces isomorph-free classes of graphs without checking the isomorphism between the generated graphs.

In this section, we discuss the different techniques and predictions we applied to increase the efficiency of the generation process as well as the mathematical theorems we used to better prune our generation tree and discard the nodes with no descendant matching the requirements for a PGP within the given index.

2.5.1 Using Common Computation

In general, since the GCCP method is a recursive algorithm and the child and the parent nodes only slightly differ, we can benefit from these similarities to avoid common computations between a node and its ancestors by storing the result for a node and then updating it for children. As we explain in the following sections, to decrease the running time, we benefit from this idea vastly, especially in computation regarding the associativity conditions.

2.5.2 Efficient Data Structure

The maximum degree of each vertex in a graph of a PGP is bounded from above by the index of the PGP. Therefore, the graphs representing PGPs are sparse and we can have *nauty* run more efficiently using sparse format to store PGP. More explanation about this format can be found in Section 2.3.2.1.

For intermediate computations, we store graphs using a modification of graph format introduced in [147]. In this format, the adjacency matrix of the graph is stored. To store the neighbours of each vertex, say v, we use a *setword* (consisting of 64 bits) when the number of vertices is at most 64 and we use an array of setwords otherwise. In the setword considered for v, each bit of represents the adjacency status between v and the vertex corresponding to that bit position. This is not only a storage efficient way to store graphs but also enables several operations, such as checking the adjacency, to be performed in a constant time instead of linear time using bit-wise operations.

We also use setwords to store and manipulate sets and subsets where for a given set or subset, each bit represents the inclusion/exclusion status of the member corresponding to that bit position. In other words, considering the primitive data type *int* be of size 8 bytes, then the binary representation of each integer represents a subset of a set of size at most 64 [124].

In Section 2.4.1, for each PGP, we defined our extension to be adding a dual object. Each potential extension can be represented by the prospective neighbours of the one or two vertices to be added as a new dual object. We know two extensions are equivalent when the neighbours of the prospective dual objects are equivalent. By definition, each new vertex should have all its neighbours in one graph and by our definitions of genuine reduction, we can limit the possible neighbours to be chosen only from the last level of the PGP.

To store a potential extension we use a data type called *setar* consisting of the neighbours of the prospective dual object to be added during the extension. Each setar has three fields: two setwords and a flag. Each of the first two fields is a setword storing an integer whose binary representation as we explained above shows the neighbours of one vertex in the prospective dual object. Note that for self dual vertices, the second field is simply empty. All the vertices stored in these two fields are at the same level, that is the last level or the level before last and the vertices stored in one field are all in the same graph. The value of the third field, flag, can have three values as follows:

- 0: This means the vertices in both fields are from the first graph of the PGP,
- 1: This means the vertices in both fields are from the second(dual) graph of the PGP and
- 2: This means the vertices in the first field are from the first graph and the vertices in the second field are from the second graph of the PGP.

When the value of the third field is 0 or 1, all the vertices of the setar are at the same odd level, indicating the new dual object being added to an even level. On the other hand, when the value of the third field is 2, all the vertices of the setar are at an even level, indicating the new dual object being added to an odd level. For example, the extension in Figure 2.5 is adding a pair of dual vertices where one vertex of the dual object is added to the higher graph and is joined to the vertices in positions $\{0,3,4\}$, for which the decimal value of the binary representation is $2^0+2^3+2^4=25$. The second new vertex is added to the other graph and is joined to vertices in positions $\{4,5\}$ for which the decimal value of the binary representation is $2^4+2^5=48$. Therefore, this extension can be stored as a setar with fields $(25,48,2)$. As other examples, consider extensions depicted by figures in Section 2.4. The setars corresponding to extensions in Figure 2.2 are $(1,1,2)$ and $(2,2,2)$, the setars corresponding to extensions in Figure 2.4 are $(1,1,0)$ and $(1,0,0)$ and the setars corresponding to extensions in Figure 2.3 are $(1,1,2)$ and $(32,32,2)$.

Storing extensions as setars that consist of two integers enables us to easily have a sorted list of all possible extensions and thus enables the application of binary search to increase the efficiency of some group calculations as we explain in Section 2.5.3.

Since the vertices stored in the two setwords of a setar are all at the same level, instead of defining a correspondence between bit positions and the label of the vertices

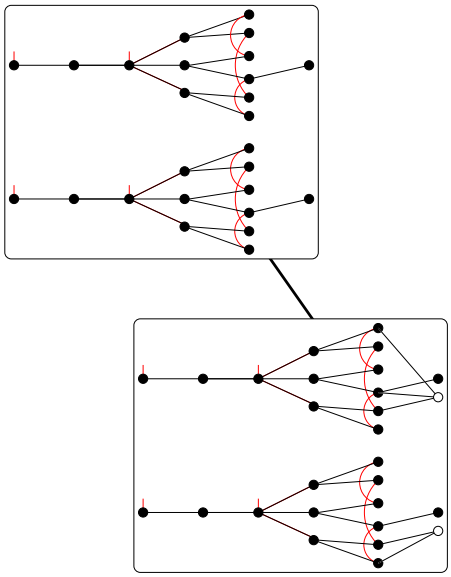

Figure 2.5: The value of the fields of setars that represent this extension is $(25, 48, 2)$.

to interpret each setword as a subset of all vertices, we consider a correspondence between bit positions and the position of the vertices at each level in a fixed arbitrary order and interpret each setword as a subset of all the vertices at a regarding level. This increases time and storage efficiency.

2.5.3 Group Calculations

Calculating the canonical labelling, orbits of a graph and in general, group calculations can be computationally expensive and inefficient in terms of storage. We used the software *nauty* to determine the canonical labelling and the automorphism group of the sparse graphs representing PGPs. In this section, we discuss the techniques we used to decrease the number of calls to *nauty* and increase the efficiency of the group calculations.

2.5.3.1 Calculating the Canonical Labelling

When determined by the user, we calculate the canonical labellings to detect isomorphism between the generated PGPs and some given PGPs such as those in the list of forbidden PGPs.

On the other hand, we compute the genuine reductions and determine winning dual objects to decide whether to accept or reject a generated PGP. Although these computations can be as expensive as calculating a canonical labelling, by defining a proper genuine reduction, in many cases we can decide, without calculating the canonical labelling, whether to reject or accept a PGP by calculating a number of

computationally easy invariants such as the total degree of vertices in dual objects. Also, with these invariants embedded in the definition of a genuine reduction, for many PGPs, we can predict and avoid extensions whose inverse operation is not a genuine reduction. With the definition of the genuine reduction in Section 2.4.3, we can avoid extensions where the new dual object is not at the last level because the obtained PGP would be rejected. In addition, when there is a pair of dual vertices at the last level, we avoid considering extensions where a self-dual vertex is added.

2.5.3.2 Avoiding Equivalent Extensions of a PGP

To avoid equivalent extensions and isomorphic children of a PGP, we extend the PGP for no more than one setar in each orbit of setars. The method we choose for this purpose affects the time and storage efficiency. We applied two main methods as follows:

1. **Storing the list of setars and calculate the orbit of setars:**
   In this method, we calculate the orbits of setars, that is, the action of the automorphism group of the vertices of the PGP, on the list of setars. In fact, an automorphism which acts on the vertices of the PGP deduces a permutation acting on the list of setars. Therefore, the list of setars must be constructed and stored. The program *nauty* can perform a given function whenever it finds an automorphism. Thus, we can calculate the orbits of setars using *nauty*.

   By the definition of setars, to enumerate all possible setars, we only need to enumerate all possible combinations of having two non-negative numbers that are bounded by the largest integer representing a valid subset of vertices at the last level or the level before last. In fact, we use a simple un-ranking function, and we consider all possible ranks for relevant subsets [124], excluding those we know in advance that do not give a valid and genuinely produced child. Here, considering a correspondence between bit positions and the position of the vertices at each level instead of the labels of the vertices as explained in Section 2.5.2 helps with time and storage efficiency.

2. **Storing a generating set of the automorphism group of the vertices:**

   In this method, instead of storing the setars, we store the automorphism group. When the number of setars is large, this method is storage efficient.

   In this method, before enumerating the list of valid setars, we call *nauty* to calculate the generators of automorphism group acting the vertices of the graph representing a PGP. The number of generators is at most $n - 1$ where n is the number of vertices. Using these generators, a procedure we call constructs a base and strong generating set for the group that allows rapid computation of

the whole group. Therefore we can store this generating set, instead of the whole group. The size of this set is at most $\frac{n^2}{2}$.

After calculating and storing the generating set, we enumerate all the valid setars without storing them. For each setar, we decide whether to reject and discard it or to accept it and extend the PGP based on it. The main point is we must guarantee the PGP is extended exactly for one representative of each orbit of setars. This representative can be any arbitrary member of the orbit. We considered these representatives to be the smallest member of each orbit, given an ordering using the integers stored at each setar. When a setar is enumerated, all the permutations acting on setars that are deduced by the automorphism group we stored are applied on the enumerated setar. If there was a permutation that takes the setar to a smaller one, we reject the setar otherwise, the setar is accepted and the PGP is extended based on this setar.

Note that if there is a permutation from one setar to another that is deduced from an automorphism, then these two setars are in the same orbit and are equivalent. Therefore, either both or none of these setars are in the valid list. For example, if one of the setars is rejected to be corresponded to a winning dual object or satisfies pruning criteria such index or associativity, so must be the other setar. Hence, it would produce an error in the program, if we fail to guarantee when a setar is rejected to be inserted to the valid list, all of the setars in the same orbit of this setar, are also rejected. Pruning techniques regarding index that are involved with floating point arithmetic and approximation can potentially create such errors.

The first method is more time efficient because it only considers the permutations corresponding to the generators while in the second method, the permutations corresponding to any automorphism in the group must be considered. On the other hand, the space complexity of the first method is linear in the number of setars while the space complexity of the second method can be as small as linear in the number of a generating set of the automorphism group.

The number of setars grows exponentially in terms of the number of vertices at one level. For example, if there are k vertices at the last level of each graph and the last level is even. Then the number of setars with vertices at the last level is $O(2^{2k})$. Therefore, it is not storage efficient nor even feasible to store all setars as the number of vertices at the two last levels increases. In the first method, using the binary search we can increase the time-efficiency and with the partitioning technique, we can increase the storage-efficiency. Below, these techniques are discussed in more detail.

• Using the binary search to find a setar

To compute the orbits of setars, we need to calculate permutations acting on setars that are deduced by automorphisms of the vertices. For this sake, we need to find each

setar, S, in the list of valid setars where S is the result of applying an automorphism on another setar. This can be done in $O(1)$ time with inverted indexing techniques. However, this technique is too storage inefficient for setars.

Let m_1 be the size of the list of valid setars, and m_2 be the number of the generators of the automorphism group acting on the vertices. To compute the orbits of setars, we need to run a search algorithm a total of $m_1 m_2$ times on the list of setars. Therefore, it is important to use an efficient search algorithm. With the definition of setars, using bit-vectors and integer representation of sets, it is easy to produce a sorted list of them to avoid the cost of a sorting algorithm and increase the efficiency using the binary search algorithm instead of a sequential search method.

• Partitioning the set of all valid setars

With this technique can decrease the time and especially space. We partition the set of all valid upper objects into parts where no two setars in two different parts are in one orbit. We call such partitioning *valid*.

The idea is if we know in advance that two setars are not in the same orbit then there is no automorphism of the vertices that deduces a permutation which takes one of these setars to the other. Therefore, we can partition the list of setars to a number of sublists where the setars in different sublists are in different orbits. We store only one sublist at a time and calculate the orbits of setars within each sublist. This way, we deal with smaller lists and hence, the space needed to store a list is decreased as well as the time needed to search the list.

Unless the automorphism group is stored, the number of calls to *nauty* in this algorithm increases from one to the number of sublists. In practice, we observed even with several calls to *nauty*, the total efficiency increases slightly when the list is partitioned to sublists according to the size of the subsets stored in a setar. This could be because of a decrease in the running of the search procedure. As we discuss in Section 2.5.5, for indices smaller than 6, we can assume the number of neighbours of a vertex is smaller than 6 and so is the size of subsets we stored in setars. That leaves us with no greater than 20 sublists to be considered according to the size of the subsets.

Note that one may consider other possible partitions as long as it can be proven that in these partitions, no two setars in two different parts are in one orbit. One general method to have a valid partitioning is the partitioning based on the orbit of the vertices. The list of setars is divided to some sublists where all setars in one sublist have the same number of vertices from each orbit of the vertices. This partitioning can be also considered for generation of other classes of objects.

Lemma 2.5.1. *The partitioning based on the number of vertices from each orbit, as described above, is a valid partitioning.*

Proof. If two setars are in two different sublists (parts), then there is at least one orbit of vertices where these setars share a different number of vertices with that orbit. This

means that these two setars are not equivalent and they can not be in the same orbit of setars. □

We used all of these ideas in our C implementation. In fact, we change our strategy based on the number of vertices at the two last levels, to have a better trade-off between time and space.

2.5.4 Associativity

Associativity is one of the conditions a PGP must satisfy in order to be correspondent to a subfactor. Therefore, we are only interested in associative PGPs. We can increase the efficiency by pruning the generation tree at nodes without any associative descendants.

As a reminder, according to the associativity condition, for any vertex v, the multiset of the duals of the neighbours of the duals of the neighbours of v is equal to the multiset of the neighbours of the duals of the neighbours of the dual of v. That is, if S is a multiset containing a subset of vertices in one graph, and $D[S]$ and $N[S]$ are the multisets consist of duals and neighbours of vertices in S, respectively, then the associativity condition says $D[N[D[N[\{v\}]]]] = N[D[N[D[\{v\}]]]]$ for all vertices v in either of the two graphs. The associativity condition can be also restated as follows:

Let $DNDN(u, v)$ be the number of ways to get from u to v by following an edge, taking the dual, following an edge from there and taking the dual again, and $NDND(u, v)$ be the number of ways to get from u to v by taking the dual, following an edge, taking the dual, and following an edge from there. A PGP, P, is associative if for all pairs of vertices $\{u, v\}$ in P we have $NDND(u, v) = DNDN(u, v)$.

It is trivial to see in any PGP, regardless of being associative or not, $NDND(u, v)$ is essentially the same as $DNDN(v, u)$ for any two vertices u and v.

In this section, we first present a lemma that says the structure of lower levels of a PGP remains unchanged in its descendants. We then present a number of lemmas that discuss the changes in the value of $DNDN(u, v)$ and $NDND(u, v)$ during the generation process. Based on these lemmas, we prune some nodes with no associative PGPs in their descendants.

Lemma 2.5.2. *Let P_1 and P_2 be two PGPs where P_1 is an ancestor of P_2 in the generation tree defined by our generation algorithm. For any $l < depth_{P_1}$, if P_1' and P_2' are the PGPs obtained from P_1 and P_2, respectively, by removing all the vertices at levels larger than l, then we have $P_1' = P_2'$.*

Proof. Based on our definition of the genuine reduction presented in Section 2.4.3, in an accepted PGP, the new dual object can only be at the last level. That is, if P is a PGP with depth $depth_P$, then any extension where a new dual object is added at a level other than $depth_P$ and $depth_P + 1$ is disregarded. That is the only difference between

a node P and a child of P is an extra dual object at the last level of the child. Thus during the generation process, dual objects are added level by level. □

In the following lemmas, u and v are two vertices of a given PGP, P and for each vertex, v of P, we define $level[v]$ to be the level of v in P, that is the smallest distance of v from a root as it is defined in Section 2.3.1.

Lemma 2.5.3. *We have $NDND(u,v) = DNDN(u,v)$ if at least one of the following holds:*

- $|level[u] - level[v]| \neq 0, 2$ *or*
- *u and v are in the same graph of the PGP.*

Proof. The value of $DNDN(u,v)$ is the number of ways to get from u to v by following an edge, taking the dual, following an edge from there and taking the dual again and the value of $NDND(u,v)$ is the number of ways to get from u to v by taking the dual, following an edge, taking the dual, and following an edge from there. That is we are taking two edges and two duals. Starting from vertex u, taking two edges, we can only get to vertices in $level[u]+2$, $level[u]-2$ or $level[u]$. Therefore, $DNDN(u,v) = NDND(u,v) = 0$ if $|level[u] - level[v]| \neq 0, 2$. Also, we take duals two times. Between these two times, we take an edge by which the level is changed by one. Therefore, we are taking a dual at an odd level and a dual at an even level. That is starting from vertex u, we can only get to vertices that are in the graph different from the graph containing the vertex u. Hence $DNDN(u,v) = NDND(u,v) = 0$ if u and v are in the same graph. □

Lemma 2.5.3, says we only need to take care of pairs of vertices, u and v where they are in different graphs of a PGP, both are at the same level or $|level[u] - level[v]| = 2$.

Lemma 2.5.4. *If $level[u] = level[v] = l$, then adding new dual objects at levels other than $l-1$ and $l+1$ does not change the values of $DNDN(u,v)$ and $NDND(u,v)$ in the new PGPs.*

Proof. If $level[u] = level[v] = l$, then apart from u, v and their duals, all the vertices on a route from u to v or v to u by following an edge, taking the dual, following an edge and taking the dual again are at levels $l-1$ and $l+1$. Therefore, the values of $DNDN(u,v)$ and $DNDN(v,u) = NDND(u,v)$ can not be affected by adding or removing vertices and dual objects at other levels. □

Lemma 2.5.5. *If $level[v] = level[u] \leq depth_P - 2$ in P, then the values of $DNDN(u,v)$ and $NDND(u,v)$ remain unchanged in all descendants of P.*

Proof. According to our definition of extension and genuine reduction, in an accepted child of P, the level of the new dual object can be only $depth_P$ or $depth_P + 1$. On the other hand, $level[v] = level[u] \leq depth_P - 2$, therefore, $l-1, l+1 \leq depth_P - 1$ and $l-1, l+1 \neq depth_P, depth_P + 1$. This means the new dual object is not inserted at level

$l-1$ or $l+1$ and hence by Lemma 2.5.4, the value of $DNDN(u,v)$ and $NDND(u,v)$ remain unchanged during an extension. Thus by induction in all descendants of P, the values of $DNDN(u,v)$ and $NDND(u,v)$ remain unchanged. □

Corollary 2.5.1. *If there are u and v in P such that $level[u] = level[v] \leq depth_P - 2$ and $DNDN(u,v) \neq NDND(u,v)$ in P, then no PGP in the subtree rooted from P in the generation tree is associative.*

One should note that Corollary 2.5.1 can not be generalised when $level[u] = level[v] = depth_P - 1, depth_P$. However, we have the following:

Corollary 2.5.2. *If there are u and v in P such that $level[u] = level[v] \leq depth_P - 1$ and $DNDN(u,v) \neq NDND(u,v)$ in P, then no PGP in a subtree rooted from P' in the generation tree is associative where P' is a PGP obtained from P by an extension that inserts a dual object at level $depth_P + 1$*

Proof. This can be inferred from Corollary 2.5.1, Noting that depth is now increased by one and we now have $level[u] = level[v] = depth_{P'} - 2$ in P'. □

Therefore, when $level[u] = level[v] = depth_P - 1$, the values of $DNDN(u,v)$ and $NDND(u,v)$ may only change by extensions that insert a dual object at level $depth_P$. Therefore, if $DNDN(u,v) \neq NDND(u,v)$ and $level[u] = level[v] = depth_P - 1$, then we can avoid the extensions in which a dual object is added at a new level of $depth_P + 1$ (joining to a number of vertices at level $depth_P$) because such an extension gives a PGP, P' where according to the definition of our genuine reduction, no further dual object can be added at level equals or less than $depth_P$ and hence we would have $DNDN(u,v) \neq NDND(u,v)$ in all accepted descendants of P'.

Note that when $level[u] = level[v] = depth_P$, then the values of $DNDN(u,v)$ and $NDND(u,v)$ can be changed only when the extension is adding a dual object at a new level of $depth_P + 1$ but in this case we cannot disregard the extensions where the dual object is added at $depth_P$ because although the resulted PGPs are not associative, they still can have associative descendants when new dual objects are added to at level of $depth_P + 1$ later along the generation tree.

Lemma 2.5.6. *If $level[v] = level[u] + 2$, then adding new dual objects at levels other than $level[u] + 1$ does not change the values of $DNDN(u,v)$ and $NDND(u,v)$ in the new PGPs.*

Proof. If $level[u] = l$ an $level[v] = l + 2$, then apart from u, v and their duals, all the vertices that lie on a route from u to v or v to u that follows a graph edge, taking the dual, following a graph edge and taking the dual again are at level $l + 1$. Therefore, the values of $DNDN(u,v)$ and $DNDN(v,u) = NDND(u,v)$ can not be affected by adding or removing vertices and dual objects at other levels. □

Lemma 2.5.7. *If $level[v] = level[u] + 2$ in P, then the values of $DNDN(u,v)$ and $NDND(u,v)$ remain unchanged in all descendants of P.*

Proof. According to our definition of extension and genuine reduction, in an accepted child of P, the level of the newly dual object can be only $depth_P$ or $depth_P + 1$. On the other hand, $level[v] = level[u] + 2 \leq depth_P$, therefore, $level[u] + 1 \leq depth_P - 1$ and $level[u] + 1 \neq depth_P, depth_P + 1$. This means the new dual object is not inserted at level $level[u] + 1$ and hence by Lemma 2.5.6, the values of $DNDN(u, v)$ and $NDND(u, v)$ remain unchanged during an extension. Thus by induction in all descendants of P, the values of $DNDN(u, v)$ and $NDND(u, v)$ remain unchanged during an extension. □

Therefore, if $level[v] = level[u] + 2$ and $DNDN(u, v) \neq NDND(u, v)$ in P, then we have $DNDN(u, v) \neq NDND(u, v)$ in all PGPs descendants of P. This is stated in Corollary 2.5.3 as follows:

Corollary 2.5.3. *If there are u and v in P where $level[v] = level[u] + 2$ and $DNDN(u, v) \neq NDND(u, v)$, then no PGP in the subtree rooted from P in the generation tree is associative.*

Lemma 2.5.8. *If $DNDN(u, v) \neq NDND(u, v)$ and at least one of the vertices of u and v are at a level other than the two last levels of P, then there is no associative PGP in the subtree rooted from P, in the generation tree of the generation algorithm we presented in Section 2.4.*

Proof. Since $DNDN(u, v) \neq NDND(u, v)$ in P, by Lemma 2.5.3 we have $|level[u] - level[v]| \in \{0, 2\}$ and since at least one of the vertices of u and v are at a level other than the two last levels of P, one can easily verify that u, v and P satisfy either conditions in Corollary 2.5.1 or conditions in Corollary 2.5.3. □

Based on these lemmas, to avoid subtrees without any associative PGP, we prune our generation tree as follows:

When a node P of depth $depth_P$ is generated, the newly added dual object, d, if a vertex $u \in d$ did not satisfy $DNDN(u, v) = NDND(u, v)$ for v at level $depth_P$, then according to Corollary 2.5.3, we can prune the generation tree at this node. If $depth_P$ is larger than the depth of the parent of P and $DNDN(u, v) = NDND(u, v)$ for u and v at level $depth_P - 2$, then according to Corollary 2.5.1, we can prune the generation tree at this node. Note depth of P and its parent is equal. This means P has already passed this filter for its vertices at level $depth_P - 2$ first time a dual object was added to level $depth_P$ and according to Lemma 2.5.4, P would pass this filter as well. Also adding d can change the value of $DNDN(u, v)$ or $NDND(u, v)$ for u and v both at level $depth_P - 1$. Hence we can update the value of them.

As given in Section 2.5.1, throughout our implementations in C, we try to predict and avoid common computation using the results stored and calculated in previous computations. For example, for each vertex, w, let $DN(w)$ be the set of duals of the neighbours of w and $ND(w)$ be the set of neighbours of the dual w. We have $NDND(u, v) = |ND(u) \cap DN(v)|$ and $DNDN(u, v) = |DN(u) \cap ND(v)|$. Therefore, instead of explicitly calculating $NDND(u, v)$ and $DNDN(u, v)$ for all u and v each time from scratch, we only compute, store and where necessary update the sets $ND(w)$

and $DN(w)$ for all vertices. Then we only compute the size of intersections $ND(u) \cap DN(v)$ and $DN(u) \cap ND(v)$ which can be calculated efficiently using the bit-vector representation of sets to store $DN(w)$ and $ND(w)$ for each vertex w.

2.5.5 Index of PGPs: Calculation and Pruning Techniques

The contribution of this chapter is in exhaustive classifications of subfactors at small indices and completing the classification of small index subfactors to index $5\frac{1}{4}$. For this sake, given an upper bound on the index of subfactors, we exhaustively generate all PGPs with indices within the given limit that are associated to subfactors. The index of a PGP is the maximum of the largest eigenvalue of the adjacency matrix of either of the two graphs of the PGP. Although the two graphs in an associative PGP have the same eigenvalue, it is not necessarily true for all PGPs in general. Therefore, the largest eigenvalue of both graphs of a PGP must be calculated for each generated PGP to determine the index. The definition of eigenvalues and some properties of indices and the largest eigenvalues are as follows:

- For a square matrix A with real entries, we say λ is an eigenvalue of A if there is a non-zero vector X such that $AX = \lambda X$

- Let Δ be the maximum degree of a graph with adjacency matrix A and λ_{max} be the largest eigenvalue of A. We have $\lambda_{max} \leq \Delta$.
  This clearly gives an upper bound on the degree of vertices of each graph of a generated PGP. However, later in this chapter, we present tighter upper bounds on degrees based on indices.

- For the adjacency matrix, A, of an undirected graph, G, if λ_A and λ_{A^2} are the largest eigenvalues of A and A^2, respectively, then we have $(\lambda_A)^2 = \lambda_{A^2}$.
  We use this property to reduce the cost of computations needed to calculate the index of each generated PGP.

- Let G and H be two graphs where G is a subgraph of H. Suppose λ_G and λ_H are the largest eigenvalue of G and H, respectively. We have $\lambda_G \leq \lambda_H$.

  With this property, only a finite number of PGPs are required to be enumerated. In addition, this property is also a great pruning tool because the two graphs of a PGP in a node of the generation tree are subgraphs of the two graphs of any PGP in a descendant node. Therefore, when the index of a generated PGP, P, is larger than the given index, the index of any node in the subtree rooted from P, is also larger than the given index and hence we can prune the generation tree at P.

In this section, we explain how we increase the efficiency of calculating the indices of generated PGPs. Given a matrix X, we denote the set of all eigenvalues of X by $\sigma(X)$ and the maximum eigenvalue of X by $\lambda_{max}(X)$ or simply by λ_{max} when there is no confusion.

2.5.5.1 Calculating Lower Bounds for Index of a PGP Using Power Iteration Method

The Power Iteration method, also known as the Von Mises iteration, is an iterative method to calculate the maximum eigenvalue and the corresponding dominant eigenvector [166]. It starts with a test vector b_0, which can be an approximation to the dominant eigenvector or a random vector and then iteratively calculates

$$b_{k+1} = \frac{Ab_k}{||Ab_k||}.$$

That is, at every iteration, the vector b_k is multiplied by the matrix A and normalized. The convergence is geometric, with ratio $|\frac{\lambda_{max-1}}{\lambda_{max}}|$ where λ_{max-1} is the second dominant eigenvalue.

Lemma 2.5.9. *Let $M_{n\times n}$ be a $(0,1)$-square matrix and X be a vector of size n where all entries of M and X are non-negative and there is at least one non-zero entry at each column and each row of M. Then we have*

$$||MX|| \geq ||X||.$$

The proof of this lemma is trivial. Based on this lemma we used the Power iteration method to calculate a lower bound on the maximum eigenvalue.

Lemma 2.5.10. *Let $M_{n\times n}$ be a $(0,1)$-square matrix where there is at least one non-zero entry at each column and each row of M,*

$$b_0 = \frac{1}{\sqrt{n}}\begin{bmatrix}1\\ \vdots \\ 1\end{bmatrix}, \text{and} \qquad\qquad b_{k+1} = \frac{Mb_k}{||Mb_k||}.$$

For all $k \geq 0$, we have $||Mb_k|| \leq \lambda_{max}(M)$.

Proof. The vectors b_ks are the test vectors in the method of Power Iteration and $||Mb_k||$ converges to $\lambda_{max}(M)$. On the other hand by Lemma 2.5.9 we know $||Mb_k||$ is non-decreasing. Therefore, for all $k > 0$ we have $||Mb_k|| \leq \lambda_{max}(M)$. □

The lemma above shows that $||Mb_k||$ is a lower bound for $\lambda_{max}(M)$.

In our implementation, to calculate the largest eigenvalues of the graphs in PGPs by the power method, we start from a test vector b_0 that is the normalisation of the vector $[1, 1, \ldots, 1]$. We stop the calculations whenever $||Mb_k||$ exceeds the index limit by a given relative error, we decide the corresponding PGP and hence its descendants are not within the given index limit and we disregard the generated PGP and prune the generation tree at that node. However, if this does not happen within a specific number of iterations (ten in our implementation), we can't decide whether the corresponding PGP is within the given index limit or not. Therefore, we accept the generated PGP.

In the following sections, we describe how we prune our generation tree based on the index-criterion in a more efficient way:

2.5.5.2 Efficient Calculations of the Maximum Eigenvalue of a Bipartite Graph

By definition, the two graphs in a PGP are bipartite. Here, we show how we reduce the time needed to calculate the maximum eigenvalue of these graphs using the fact that they are bipartite.

Lemma 2.5.11. *For any matrix $M_{n\times n}$ we have $\sigma(MM^T) = \sigma(M^TM)$.*

Proof. Let $\lambda \in \sigma(MM^T)$ then there is vector V such that $(MM^T)V = \lambda V$. Then we have:

$$(MM^T)V = \lambda V \Rightarrow M^T(MM^T)V = \lambda M^TV \Rightarrow (M^TM)(M^TV) = \lambda(M^TV).$$

This means $\lambda \in \sigma(M^TM)$. Therefore, $\sigma(MM^T) \subseteq \sigma(M^TM)$. Similarly, we can show $\sigma(M^TM) \subseteq \sigma(MM^T)$ and therefore, $\sigma(MM^T) = \sigma(M^TM)$. □

Lemma 2.5.12. *For any three square matrices M, M_1 and M_2 where*

$$M = \left[\begin{array}{c|c} M_1 & 0 \\ \hline 0 & M_2 \end{array}\right],$$

we have $\sigma(M) = \sigma(M_1) \cup \sigma(M_2)$.

Proof. Let $\lambda \in \sigma(M)$ then there is a non-zero vector V such that $MV = \lambda V$. On the other hand:

$$MV = \left[\begin{array}{c} M_1V_1 \\ \hline M_2V_2 \end{array}\right],$$

where

$$\left[\begin{array}{c} V_1 \\ \hline V_2 \end{array}\right] = V.$$

Thus we have $M_1V_1 = \lambda V_1$ and $M_2V_2 = \lambda V_2$. Since V is non-zero, either V_1 or V_1 must be non-zero. Therefore, $\lambda \in \sigma(M_1)$ or $\lambda \in \sigma(M_2)$, that is, $\lambda \in \sigma(M_1) \cup (M_2)$. This shows $\sigma(M) \subseteq \sigma(M_1) \cup \sigma(M_2)$. It can be shown a similar fashion that $\sigma(M_1) \cup (M_2) \subseteq \sigma(M)$ and hence the theorem is proved. □

Lemma 2.5.13. *If A and B be two matrices where A is a square matrix and we have:*

$$A = \left[\begin{array}{c|c} 0 & B \\ \hline B^T & 0 \end{array}\right],$$

then we have $\left(\lambda_{max}(A)\right)^2 = \lambda_{max}(BB^T)$.

Proof. First of all, BB^T and B^TB are square matrices. We also have:

$$A^2 = \left[\begin{array}{c|c} BB^T & 0 \\ \hline 0 & B^TB \end{array}\right].$$

Therefore, by Lemma 2.5.12, we have $\sigma(A^2) = \sigma(BB^T) \cup \sigma(B^TB)$. We know by Lemma 2.5.11 that $\sigma(BB^T) = \sigma(B^TB)$, hence $\sigma(A^2) = \sigma(BB^T) = \sigma(B^TB)$. This means $\lambda_{max}(A^2) = \lambda_{max}(BB^T) = \lambda_{max}(B^TB)$.

It is well-known that $\big(\lambda_{max}(X)\big)^2 = \lambda_{max}(X^2)$ for any square matrix X. Therefore, we have $\big(\lambda_{max}(A)\big)^2 = \lambda_{max}(BB^T)$. □

Theorem 2.5.4. *The index of a PGP with two graphs G_1 and G_2, is obtained by calculating the maximum eigenvalues of the two matrices H_1 and H_2 and taking the larger one between these two, where H_1 is the matrix with each row and column representing a vertex at an even level of G_1 and the value of $H_1[i][j]$ is the number of paths of length two form the ith vertex to the jth vertex in even level of G_1, and H_2 is defined in a similar way to represent the number of paths of length two between the vertices at even levels in G_2.*

Proof. First note that since G_1 is a bipartite graph with two parts of say k_1 and k_2 vertices, then G_1 can be represented with an adjacency matrix A of the form

$$A = \left[\begin{array}{c|c} 0 & B \\ \hline B^T & 0 \end{array}\right],$$

where the first k_1 rows and columns represent the vertices in one part of G_1, for example, vertices at even levels, and the last k_2 rows and columns represent the vertices in the other part of G_1, say vertices at odd levels and B is a $k_1 \times k_2$ matrix representing the edges from one part to another.

Therefore, by Lemma 2.5.13, we have $\big(\lambda_{max}(A)\big)^2 = \lambda_{max}(BB^T)$ while with our definition of matrix B here, one can easily see that BB^T, as submatrix of A^2, is the matrix of number of paths of length two between the vertices at even levels in G_1. Similar argument can be considered for G_2. □

Based on this theorem, to calculate the index of PGPs, instead of calculating the largest eigenvalues of the adjacency matrices, we calculate the largest eigenvalues of much smaller matrices. These smaller matrices representing the number of paths of length two between only the vertices at even levels (or at odd levels). This can dramatically increase the time efficiency considering the iterative algorithm of calculating the largest eigenvalues.

2.5.5.3 Upper Bound on the Degree of Vertices

We present upper bounds on the degree of vertices in a PGP based on the index of the PGP. This helps us to predict for some children if their index is going to be larger

than the given limit and hence we can prune the generation tree while avoiding the computation needed in the power method.

For each vertex, v, at level, l, of a graph of a PGP, we define the set of children of v to be the neighbours of v that are at level $l-1$ and the set of parents of v to be the neighbours of v at level $l+1$. We use the following lemma to calculate some bounds on the number of children of each vertex. This lemma is posed and proved by Brendan McKay.

Lemma 2.5.14. *Let G be a bipartite graph with a vertex that is adjacent to at least k vertices of degree at least two, and at least l other vertices. Then the maximum eigenvalue $\lambda_{max}(G)$ satisfies:*

$$\Big(\lambda_{max}(G)\Big)^2 \geq \frac{k+l+1+\sqrt{(k+l+1)^2-4l}}{2},$$

which is the largest root of $y^2-(k+l+1)y+l$.

Proof. Consider a graph H that has a vertex v with neighbour set W. Write W as a disjoint union $W = W_1 \cup W_2$. Form a new graph J from H by splitting v into two vertices v_1 and v_2, with v_1 adjacent only to W_1 and v_2 adjacent only to W_2. We know that

$$\lambda_{max}(J) = \max_x \frac{x^T J x}{x^T x};$$

choose an x that achieves the maximum. Form a vector y from x by combining elements x_{v_1}, x_{v_2} into a single element $\sqrt{x_{v_1}^2+x_{v_2}^2}$. Then $y^T y = x^T x$ and $y^T H y \geq x^T J x$. Therefore, $\lambda_{max}(H) \geq \lambda_{max}(J)$.

Now consider our bipartite graph G. Let a be a vertex given in the theorem and let B be the set of k neighbours of degree at least two. Split the neighbours of B (other than a) as we have described, until they have degree one. This operation cannot increase the maximum eigenvalue, as we have shown. Now G contains the tree $T_{k,l}$ which consists of k paths of length two and l paths of length of one, all joined together at one end (so there are $2k+l+1$ vertices altogether). By an induction, we find that the characteristic polynomial of $T_{k,l}$ is

$$x^{l-1}(x^2-1)^{k-1}(x^4-kx^2-lx^2-x^2+l),$$

and we can see that the largest eigenvalue is given by the last factor. □

This lemma restricts the possible values of k and l, as are defined in the theorem, based on the upper bound for the square of the maximum eigenvalue. For example, Table 2.1 shows the value of the function

$$f(k,l) = \frac{k+l+1+\sqrt{(k+l+1)^2-4l}}{2}$$

where the first row and column show different values of l and k respectively and 6^+ indicates $f(k,l) > 6$.

k/l	0	1	2	3	4	5^+
1	2	2.5 - 3	3 - 3.5	4 - 4.5	5 - 5.5	6^+
2	3	3.5 - 4	4.5 - 5	5 - 5.5	6^+	6^+
3	4	4.5 - 5	5.5 - 6	6^+	6^+	6^+
4	5	5.5 - 6	6^+	6^+	6^+	6^+
5	6	6^+	6^+	6^+	6^+	6^+
6^+	6^+	6^+	6^+	6^+	6^+	6^+

Table 2.1: $f(k,l) = \frac{k+l+1+\sqrt{(k+l+1)^2-4l}}{2}$ where 6^+ indicates $x > 6$.

Note that when we want to add a new vertex at a level $h > 2$, we first choose a set S of its neighbours at level $h-1$ in the graph. For any vertex, $v \in N$, the number of parents of v is equal to the number of neighbours of v at level $h-2$ and we know all parents of v are of degree at least two because they have a parent and at least one child, v. Therefore, if k is the number of parents of v, then v is adjacent to at least k vertices of degree at least two and Lemma 2.5.14 gives an upper bound on the maximum number of children of v so that the square of the maximum eigenvalue stays below the given limit. This way, Lemma 2.5.14 gives an upper bound the maximum number of children of the vertices of a graph and restrict our possible extensions to those satisfying this upper bound.

For example, when the given index-limit is six, i.e., $\left(\lambda_{max}(G)\right)^2 \leq 6$, Table 2.1 shows that a vertex with one parent, $k = 1$, can have at most 4 children, a vertex with two parents, $k = 2$, can have at most 3 children, and so on.

In Table 2.2, the first row is the number, k, of the parents of a vertex and the second row shows the maximum number of children each vertex with k parents can have while staying below the index-limit of $\left(\lambda_{max}(G)\right)^2 \leq 6$.

k	1	2	3	4	5
$l \leq$	4	3	2	1	0

Table 2.2: The maximum number of children of a vertex with k parents to have $\left(\lambda_{max}(G)\right)^2 \leq 6$.

Since the index of graphs and the degree of vertices is not decreasing in descendants of a PGP, Lemma 2.5.14 gives a powerful tool for pruning the generation tree without even calculating the maximum eigenvalue of a graph. For example, when the

upper bound on the index is 6, Table 2.2 says each vertex can have at most five parents and at most four children and we have $k + l \leq 5$. So we benefit from this to disregard ineligible setars before to avoid extensions that result in PGPs with too large indices.

The following ideas can help to increase the efficiency. These ideas are not implemented in our C program. However, exploiting the last two ideas in implementations by other authors has considerably improved the efficiency [9].

- Use and update A^2 from the parent node instead of calculating it for each generated PGP out of scratch.
- Developing some theorems to predict when the largest eigenvalues of the two graphs of a PGP are the same to avoid the extra calculations in such cases. For example, when a PGP is associative, the eigenvalues of the two graphs are the same and we can calculate it only for one of the graphs.
- Using another initial test vector b_0 that is a better estimate of the Perron vector, the dominant eigenvector, can increase the efficiency. For example, one might use the last vector b_k calculated for the parent of a PGP to construct an estimate for the graphs in the child PGP.
- Using the special structure of the graphs of PGP to make calculations of multiplications of matrices more efficient. For example, considering only the vertices in order of their level, then all entries in the sub-matrix or its powers (same for the sub-matrices of vertices at even levels) are all zero except those at super and sub-diagonal bands.

2.5.6 Giving Priority to One Graph of a PGP

In this section, we discuss an idea that can affect the efficiency at the cost of generating duplicated PGPs. This idea is not implemented in our C-code program but the implementation developed in Scala by Scott Morrison and David Penneys [9] benefited from this idea. We discuss how this idea could help with the efficiency and how it can destroy the exhaustiveness or uniqueness.

Let P be a PGP consisting of two bipartite graphs g_1 and g_2. We could avoid repeating the calculation of some functions on g_1 by calculating them once when we start adding dual objects to g_2 and storing and using them in the upcoming extensions as long as the level is not changed, if we prioritize g_1 to g_2 in our definition of genuine reduction so that a graph pair would be rejected if it is generated by adding a dual object to the last level of g_1 while g_2 is not empty in that level, then in the process of generation of each graph pair, as it is extended level by level, at each even level, all the dual objects in g_1 are added before any dual object in g_2 is added to that level. This means when we are adding a dual object to g_2 at the same level, g_1 remains unchanged. Since being in graph g_1 or g_2 is not an invariant under isomorphism, with this idea duplicated PGPs can be produced and hence the exhaustiveness or uniqueness in our generation algorithm can be destroyed. We discuss this in more detail as follows:

If we prioritise g_1 to g_2 and consider $\{g_1, g_2\}$ and $\{g_2, g_1\}$ isomorphic assigning same colour to the starred vertices, then:

- We may generate duplicated graph pairs if we consider the winning dual objects locally, i.e. for $i = 0, 1$, if the newly added dual object is in g_i, to decided whether it is winning or the generated graph pair must be rejected, the new dual object is only compared to other dual objects at the last level of g_i. For example, consider the graph pairs (g, h) and (h, g) that are generated from the two non-isomorphic graph pairs $(g, h \setminus x)$ and $(h, g \setminus y)$, by adding dual objects x and y respectively where x and y are the local winning dual objects in h and g, respectively. This can happen only if the PGP in the root of the generation tree consists of two isomorphic graphs.

- Our generation process may not be exhausted if the winning dual objects are selected globally, i.e. among all dual objects at the last level of g_1 and g_2. For example, let the PGP in the root of the generation tree consists of two non-isomorphic graphs and consider the PGP (g, h) where the global winning dual object, x, at the last level is in g. If we have the PGP $(g \setminus x, h)$ generated in our process, we know that $(h, g \setminus x)$ is isomorphic to $(g \setminus x, h)$ and hence is not generated. So the only way (g, h) can be generated is through extending $(g \setminus x, h)$ by adding x, but this extension is not accepted if there are other dual objects at the last level in h. Therefore, because of prioritising g to h, the pair

(g, h) generated this way is also rejected. Thus (g, h) is never accepted and our generation is not exhaustive.

Also we can have our generation process exhaustive while prioritising g_1 to g_2, having the winning dual objects being globally selected and not considering $\{g_1, g_2\}$ and $\{g_2, g_1\}$ isomorphic for non-isomorphic graphs g_1 and g_2 and assigning different colours to starred vertices in different graphs. Obviously both $\{g_1, g_2\}$ and $\{g_2, g_1\}$ can be produced in this approach if in the root of the generation tree, the PGP consists of two isomorphic graphs.

As given above, in our *C* implementation, we considered $\{g_1, g_2\}$ and $\{g_2, g_1\}$ as isomorphic graph pairs. We did not prioritise one graph to another in the definition of our genuine reduction and we calculated the winning dual objects globally. This implementation generates PGP exhaustively and uniquely.

2.6 Formal Proof

In this section, we prove that given a PGP r, our algorithm generates all PGPs rooted from r without producing isomorphic copies. The proofs are based on the theorems stated in [146] for the general method of GCCP to produce isomorph-free families of combinatorial objects. This method has been discussed more extensively in other chapters.

Given a PGP r, let $\mathcal{L}_r$ be the set of all labelled PGPs descendant from r where labels $\{1, 2, .., n\}$ are used to label a PGP with total of n vertices. For each $X \in \mathcal{L}_r$, we define $o(X)$ to be the number of dual objects in X. Let G be the group of all relabellings of labelled PGPs so that the PGPs in one orbit are all isomorphic. Hence $G = \mathcal{S}_1 \times \mathcal{S}_2 \times ...$, where the action on $\mathcal{L}_r$ is such that the factor $\mathcal{S}_n$ is the symmetric group of degree n permuting the labels of vertices of PGPs in $\mathcal{L}_r$ with n vertices. Let S be a setar in X, we denote by X_S, the PGP obtained from extending X by adding a dual object and joining it accordingly to the vertices in S. Let $X \in \mathcal{L}_r$ be an arbitrary PGP. We define the following:

- $L(X)$ is the set of lower objects of X where:

$$L(X) = \begin{cases} \{\langle X, d\rangle | d \text{ is a dual object in } X\}, & \text{where } o(X) > o(r) \\ \emptyset, & \text{Otherwise} \end{cases}$$

  therefore, $L(r) = \emptyset$.

- $U(X)$ is the set of upper objects of X where:

$$U(X) = \{\langle X, S\rangle | S \text{ is a setar in } X \text{ and } X_S \in \mathcal{L}_r\}.$$

- $\check{\mathcal{L}} = \cup_{X \in \mathcal{L}_r} L(X)$ and $\hat{\mathcal{L}} = \cup_{X \in \mathcal{L}_r} U(X)$.

- For each $\check{Y} = \langle Y, d\rangle \in \check{\mathcal{L}}$:

$$f(\check{Y}) = \{\hat{X}^g = \langle X, S\rangle^g |\ g \in G,\ X \text{ is obtained from } Y \text{ by removing } d \text{ and } S \text{ is the setar storing the neighbours of the vertices of } d \text{ in } Y\}.$$

  Clearly $\hat{X} = \langle X, S\rangle \in \hat{\mathcal{L}}$ and $Y = X_S$.

- For each $\hat{X} = \langle X, S\rangle \in \hat{\mathcal{L}}$:

$$f'(\hat{X}) = \{\check{Y}^g = \langle Y, d\rangle^g |\ g \in G,\ Y = X_S \text{ and } d \text{ is the dual object in } Y \text{ corresponding to setar } S\}$$

It is obvious that $\check{Y} = \langle Y, d\rangle \in \check{\mathcal{L}}$.

- According to our definition of the genuine reduction, the function $m : \mathcal{L}_r \to 2^{\check{\mathcal{L}}}$ given in [146] is defined as follows :

$$m(X) = \begin{cases} \{(X,d)|\ \ d \text{ is a winning dual object in } X\} & \text{if } o(X) > o(r) \\ \emptyset & \text{Otherwise} \end{cases}$$

As a reminder from Section 2.4.3, the dual object d in X is winning if there is no dual object d', in X, whose 4-tuple, $x(d')$, is lexicographically larger than the 4-tuple $x(d)$ where $x(k) = (x_0(k), x_1(k), x_2(k), x_3(k))$ for each dual object k.

Theorem 2.6.1. *Our definitions satisfy the constraints of axioms* $C1 - C7$ *stated in [146].*

Proof. The group G is the group of all relabellings of labelled PGPs in $\mathcal{L}_r$, so that the PGPs in one orbit are all isomorphic. This group acts on lower and upper objects as follows: For each $g \in G$, $\langle X, d\rangle \in \check{\mathcal{L}}$ and $\langle X, S\rangle \in \hat{\mathcal{L}}$, we have $\langle X, d\rangle^g = \langle X^g, d^g\rangle$ and $\langle X, S\rangle^g = \langle X^g, S^g\rangle$. Also, it is easy to see that an image, S^g of a setar S under g is also a setar in X^g and the property of being a valid setar is invariant under isomorphisms. This means $X_S \in \mathcal{L}_r$ if and only if $X^g_{S^g} \in \mathcal{L}_r$.

C1. We show G fixes each of $\mathcal{L}_r$, $\check{\mathcal{L}}$ and $\hat{\mathcal{L}}$ setwise.

1. For any $X \in \mathcal{L}_r$ and $g \in G$, the graph X^g is isomorphic to X and hence it is also in $\mathcal{L}_r$.
2. For any $\langle X, d\rangle \in \check{L}$ and $g \in G$ we have $\langle X, d\rangle^g = \langle X^g, d^g\rangle$ where X^g is isomorphic to X and in L_r. It is not hard to see d^g is a dual object of X^g and hence by our definitions $\langle X^g, d^g\rangle \in L(X^g)$. Therefore, $\langle X, d\rangle^g = \langle X^g, d^g\rangle \in \check{L}$.
3. For any $\langle X, S\rangle \in \hat{L}$ and $g \in G$ we have $\langle X, S\rangle^g = \langle X^g, S^g\rangle$ where X^g is isomorphic to X and hence is in L_r and as we given above S^g is a setar in X^g and $X^g_{S^g} \in L_r$. Therefore, $\langle X^g, S^g\rangle \in U(X^g)$ and consequently $\langle X, S\rangle^g = \langle X^g, S^g\rangle \in \hat{L}$.

C2. We show for each $X \in \mathcal{L}_r$, we have: $L(X^g) = L(X)^g$ and $U(X^g) = U(X)^g$:

1. $L(X^g) = L(X)^g$:

$$\begin{aligned} L(X) &= \{\langle X, d\rangle | d \text{ is a dual object of } X\} \\ \implies L(X)^g &= \{\langle X, d\rangle^g | d \text{ is a dual object of } X\} \\ &= \{\langle X^g, d^g, \rangle | d^g \text{ is a dual object of } X^g\} = L(X^g). \end{aligned}$$

2. $U(X^g) = U(X)^g$:

$$U(X) = \{\langle X, S\rangle | S \text{ is a setar in } X \text{ and } X_S \in \mathcal{L}_r\}$$

$$\Longrightarrow U(X)^g = \{\langle X, S\rangle^g | S \text{ is a setar in } X \text{ and } X_S \in \mathcal{L}_r\}$$
$$= \{\langle X^g, S^g\rangle | S^g \text{ is a setar in } X^g \text{ and } X^g_{S^g} \in \mathcal{L}_r\} = U(X^g).$$

C3. $\forall \check{Y} \in \check{\mathcal{L}}, f(\check{Y}) \neq \emptyset$:

For each $\check{Y} \in \check{\mathcal{L}}$, we have $\check{Y} = \langle Y, d\rangle$ where $Y \in \mathcal{L}_r$, $o(Y) > o(r)$ and d is a dual object in Y. Let S be the setar representing the neighbours of d in Y and X be the PGP obtained from Y by removing d. It is easy to see $\langle X, S\rangle \in f(\check{Y})$. Hence $f(\check{Y}) \neq \emptyset$.

C4. We show for any $\check{Y} \in \check{\mathcal{L}}$, $g \in G$, $\hat{X}_1 \in f(\check{Y})$ and $\hat{X}_2 \in f(\check{Y}^g)$, there exists $h \in G$ such that $\hat{X}_1^h = \hat{X}_2$.

Suppose $\hat{X}_1 = \langle X_1, S_1\rangle$, $\hat{X}_2 = \langle X_2, S_2\rangle$ and $\check{Y} = \langle Y, d\rangle$. Hence $\check{Y}^g = \langle Y^g, d^g\rangle$. Since $\hat{X}_1 \in f(\check{Y})$, there is $h_1 \in G$ satisfying $X_1 = A_1^{h_1}$ and $S_1 = B_1^{h_1}$ where A_1 is obtained from Y by removing the vertices in d and B_1 is the setar representing of neighbours of the vertices of d in Y. Also, we have $\hat{X}_2 \in f(\check{Y}^g)$, so there is $h_2 \in G$ satisfying $X_2 = A_2^{h_2}$ and $S_2 = B_2^{h_2}$ where A_2 is obtained from Y^g by removing the vertices in d^g and B_2 is the setar representing the of neighbours of the vertices of d^g in Y^g.
Consider $h = h_2(g(h_1^{-1}))$, Since $h_1, h_2, g \in G$, we have $h \in G$. We prove $\hat{X}_1^h = \hat{X}_2$:

1. $X_1^h = X_2$: We have $X_1^{h_1^{-1}} = A_1$ and $A_2^{h_2} = X_2$. It is easy to verify $A_1^g = A_2$. Hence $X_1^h = X_1^{h_2(g(h_1^{-1}))} = (X_1^{h_1^{-1}})^{h_2 g} = (A_1^g)^{h_2} = A_2^{h_2} = X_2$.

2. $S_1^h = S_2$: We have $S_1^{h_1^{-1}} = B_1$ and $B_2^{h_2} = S_2$. It is easy to verify $B_1^g = B_2$. So similar to above, one can verify $S_1^h = S_2$.

C5. We show for any $\hat{X} \in \hat{\mathcal{L}}$, $g \in G$, $\check{Y}_1 \in f'(\hat{X})$ and $\check{Y}_2 \in f'(\hat{X}^g)$, there exists $h \in G$ such that $\check{Y}_1^h = \check{Y}_2$.

Suppose $\check{Y}_1 = \langle Y_1, d_1\rangle$, $\check{Y}_2 = \langle Y_2, d_2\rangle$ and $\hat{X} = \langle X, S\rangle$. Hence $\hat{X}^g = \langle X^g, S^g\rangle$. We have $\check{Y}_1 \in f'(\hat{X})$, so there is $h_1 \in G$ satisfying $Y_1 = A_1^{h_1}$, $d_1 = x_1^{h_1}$ where $A_1 = X_S$ and x_1 is the dual object corresponded to S in A_1. Also, we have $\check{Y}_2 \in f'(\hat{X}^g)$, so there is $h_2 \in G$ satisfying $Y_2 = A_2^{h_2}$, $d_2 = x_2^{h_2}$ where $A_2 = X^g_{S^g}$ and x_2 is the dual object corresponded to S^g in A_2.
Consider $h = h_2(g(h_1^{-1}))$, Since $h_1, h_2, g \in G$, we have $h \in G$. We prove $\check{Y}_1^h = \check{Y}_2$:

1. $Y_1^h = Y_2$: We have $Y_1^{h_1^{-1}} = A_1$ and $A_2^{h_2} = Y_2$. It is easy to verify $A_1^g = A_2$. So similar to **C4**, one can easily show $Y_1^h = Y_2$.

2. $d_1^h = d_2$: We have $d_1^{h_1^{-1}} = x_1$ and $x_2^{h_2} = d_2$. It is obvious that $x_1^g = x_2$ and hence $d_1^h = d_2$.

C6. We show $o(X^g) = o(X)$:

For each $X \in \mathcal{L}_r$, we defined $o(X)$ to be the number of dual objects in X that is an invariant under isomorphisms. This means $o(X^g) = o(X)$.

C7. We show for each $\check{Y} \in \check{\mathcal{L}}$ and $\hat{X} \in f(\check{Y})$ we have $o(\hat{X}) < o(\check{Y})$:
According to our definitions for each $\check{Y} \in \check{\mathcal{L}}$ and $\hat{X} \in f(\check{Y})$ we have $o(\hat{X}) = o(\check{Y}) - 1 < o(\check{Y})$.

□

Lemma 2.6.1. *The function $m(X)$ defined above is well-defined and satisfies the requirements $M1 - M3$ stated in [146].*

Proof. It is obvious that $m(X)$ is well defined. We prove each requirement $M1$, $M2$ and $M3$ holds:

M1. If $L(X) = \emptyset$, then $m(X) = \emptyset$:
According to the definitions we presented for $m(X)$ and $L(X)$, we have $m(X) \subseteq L(X)$ and hence if $L(X) = \emptyset$, then $m(X) = \emptyset$.

M2. If $L(X) \neq \emptyset$, then $m(X)$ is an orbit of the action of $Aut(X)$ on $L(X)$:
Consider a PGP X with $L(X) \neq \emptyset$. Since the values of x_0, x_1 and x_2 are vertex invariants, all vertices in the same orbit of the action of $Aut(X)$ on $L(X)$ have same values for them. It is also clear that the value of x_3 for all vertices in one orbit is the same. So if for a dual object d we have $\langle X, d\rangle \in m(X)$ then for every dual object, d', that is in the same orbit as d we have $\langle X, d'\rangle \in m(X)$.
Furthermore, the value of x_3 is different for dual objects in different orbits, therefore, $m(X)$ is an orbit of action of $Aut(X)$ on $L(X)$.

M3. For each $X \in \mathcal{L}_r$ and $g \in G$ we have $m(X^g) = m(X)^g$:
Let $W(X) = \{d | \langle X, d\rangle \in m(X)\}$. We only need to show $W(X^g) = W(X)^g$. We proceed the proof in two steps:

1. For each $k \in W(X)^g$, we have $k \in W(X^g)$:

   If $k \in W(X)^g$ then $d = k^{g^{-1}} \in W(X)$ and hence d is a dual object with lexicographically largest 4-tuple for $x = (x_0, x_1, x_2, x_3)$ in X. Since the values of x_0, x_1, x_2 and x_3 are vertex invariants, then the 4-tuple $x(d^g)$ must also be lexicographically largest among all the 4-tuples of all dual objects in X^g. Therefore, $k = d^g \in W(X^g)$.

2. For each $k \in W(X^g)$, we have $k \in W(X)^g$:

   If $k \in W(X^g)$, then k is a dual object with lexicographically largest 4-tuple for $x = (x_0, x_1, x_2, x_3)$ in X^g. Since the values of x_0, x_1, x_2 and x_3 are vertex invariants, then the 4-tuple x of $d = k^{g^{-1}}$ must also be lexicographically largest among all the 4-tuples of all dual objects in X. Therefore, $d \in W(X)$ and hence $k = d^g \in W(X)^g$.

□

2.7 The C Code Program

A Mathematica wrapper has been prepared by Morrison for the C code program developed to exhaustively generate PGPs. Hence, this program can be run by loading "/development/afzaly-enumerator/Enumeration-setup.m" from the FusionAtlas repository in a Mathematica session (after first loading the FusionAtlas itself), and then using the command ExtendToDepth. Alternatively, it can be run directly as:

"enumerate-pgps *graphpair* –maxindex #*index* [–maxrank #*maxrank*] [–maxdepth #*depth*] [–ignore *ignorefilename*] [–check *weedsfilename* [–printNEW *newfilename*]] [–printPGP *pgpfilename*] [–printALL *allfilename*] [–maxtime#t *stopfilename*] ",

or "enumerate-pgps –resume *resumefilename*",

Where the parameters in brackets are optional and the terms start with # indicates a number. For example, one may run the program as follows:

"enumerate-PGPs –maxindex 5.0000090449 bwd1v1duals1v1,bwd1v1duals1v1 –ignore badweeds –check allweeds –maxrank 30".

Therefore, the C code program takes a graph pair, *graphpair* as the root of the generation tree and a real number,*index*, as the maximum index, and it generates all PGPs up to that maximum index. The program can consider a list of forbidden PGPs to avoid. It accepts other optional command line arguments that change the behaviour of the program as follows:

- **–ignore** *ignorefilename*:
  The internal nodes that appear in the input file *ignorefilename* are disregarded as well as all of their descendants.

- **–check** *weedsfilename*:
  The program checks if the set of the internal nodes are the same as the ones in the input file *weedsfilename*

- **–printNEW** *newfilename* :
  (only comes with –check) The program prints to the file *newfilename*, all the internal nodes not appearing in file *weedsfilename*.

- **–maxrank** #*maxrank* :
  The total number of vertices of each produced PGP is at most #*maxrank*.

- **–maxdepth** #*depth* :
  The depth of each generated graph pair is at most #*depth*.

- **–printPGP** *pgpfilename*:
  The program prints all generated PGPs to the file *pgpfilename*.
- **–printALL** *allfilename*:
  The program prints all generated PGPs to the file *allfilename*.
- **–maxtime** #*t stopfilename* :
  The program stops after *t* seconds and stores some information about the current node to the file *stopfilename*.
- **–resume** *resumefilename* :
  The program reads the root node along with certain data from the file *resumefilename*.

The C code program can be easily adapted to accommodate parallelism. In fact, this feature had been implemented in the earlier versions of the program but not being of the interest of the users, it was later removed.

2.7.1 The Features maxtime and resume

The two options *maxtime* and *resume* allow the user to stop and resume the program any number of times without producing repeated results. When the program is called with option **–maxtime**, it keeps track of the path from the root to the current node. For each node, *node*, on this path, it stores the information of the position of the setar in the valid setar list of the parent node based on which the parent is extended to *node*. When the list of all valid setars is divided to a number of sublists, then program keeps track of the information about the sublist the regarding setar is accommodated at as well as its position in that sublist.
This information, for the last generated node (that is the first node generated after *t* seconds) and for all its ancestors is stored at the file *stopfilename* as well and all command arguments of the user. When the program is called with option **–resume** *stopfilename*, all the information is restored. The program starts from the root. All nodes of the generation tree are now partitioned into three parts: nodes that had not been generated, nodes that are on the path from the root to the last node generated in the previous call to the program, and the nodes that were generated in the previous run but are not on the path from the root to the last node generated in the previous run. These are demonstrated in Figure 2.6 with white vertices, vertices within a circle, and black vertices respectively. The last generated node is depicted with a shaded circle around it. When extending a node, we skip generating all children in the third part while generating all other children. We avoid considering the nodes in the second part for calculating any statistics. As these nodes are previously generated and the only reason we are generating them is to access those children of them that had not been generated.

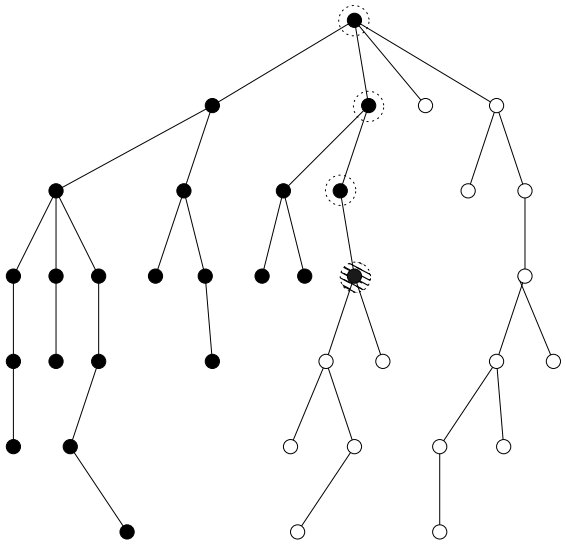

Figure 2.6: The partitioning of all the nodes of the generation tree into three parts: Nodes not been generated yet (white), nodes that are on the path from the root to the last node generated in the previous run (inside a circle), and the rest (black).

2.8 Conclusion

Exhaustive classifications of subfactors at small index help with the exploration into quantum symmetries and improvement of knowledge. Quantum symmetries have different applications including applications in designing quantum computers. These classifications had been reached up to index 5 through multiple steps taken by several researchers applying different techniques. Unfortunately enumerating possible graph pairs above index 4 gets difficult quickly and these techniques struggle beyond index 5. Most of the classification processes have consisted of two steps:

- Generation of all possible PGPs, satisfying certain combinatorial constraints, with graph index up to the square root of the index.
- Filtering the PGPs for which we have the knowledge that they cannot be associated to any subfactor and then proceed the classification with the remaining cases.

Over the years, a number of important obstructions to a graph pair being realized as the principal graph of a subfactor have been developed. Incorporating many of these obstructions into the algorithm for enumerating potential principal graphs of subfactors has helped with both steps given above.

Our contribution is a significant improvement to the first step, by applying, for the first time, a more sophisticated generation method to produce a wider range of PGPs

as opposed to the previous methods that were time-consuming and were based on pairwise checking to removing isomorphic copies. Applying the method of Generation by Canonical Construction Path increased the efficiency of the step of exhaustively generating PGPs by orders of magnitude and played a critical role in furthering the complete classification of subfactors up to index $5\frac{1}{4}$ which includes $3+\sqrt{5}$, the first interesting composite index. These results along with examples and information about running time of the program can be found in [9].
The program developed during this research can produce PGPs of higher indices and hence can be applied to extend even further the classification of subfactors, in particular, if new pruning and filtering techniques are introduced. Nevertheless, as the index gets higher, larger and denser PGPs appear. In this case, the program may need a number of modifications to accommodate denser PGPs, especially if the number of vertices at each level grows beyond certain limits. Also, the trade-off between the time and space as in Section 2.5.3 needs to be adjusted accordingly. Note that the ideas given in this section can be adapted to used in the generation of other objects, as such is the idea of valid partitioning introduced in Lemma 2.5.1.

To increase the efficiency even further, one may develop new pruning techniques and apply ideas such as those given at the end of Section 2.5.5 to reduce the cost of calculations needed.

Chapter 3

The Turán Numbers for Cycles

3.1 Abstract

For a given set of graphs, $\mathcal{H}$, the Turán Number of $\mathcal{H}$, $ex(n, \mathcal{H})$, is defined to be the maximum number of edges in a graph on n vertices without a subgraph isomorphic to any graph in $\mathcal{H}$. Denote by $EX(n, \mathcal{H})$, the set of all *extremal graphs* with respect to n and $\mathcal{H}$, i.e., graphs with n vertices, $ex(n, \mathcal{H})$ edges and no subgraph isomorphic to any graph in $\mathcal{H}$. We consider this problem when $\mathcal{H}$ is a set of cycles.

We introduce new results for $ex(n, \mathcal{C})$ and $EX(n, \mathcal{C})$ using a set of algorithms we have developed which are based on the method of *Generation by Canonical Construction Path*. Let $\mathcal{K}$ be an arbitrary subset of $\{C_3, C_4, C_5, \ldots, C_{32}\}$. For given n and a set of cycles, $\mathcal{C}$, these algorithms can be used to calculate $ex(n, \mathcal{C})$ and extremal graphs in $Ex(n, \mathcal{C})$ by recursively extending smaller graphs without any cycle in $\mathcal{C}$ where $\mathcal{C} = \mathcal{K}$ or $\mathcal{C} = \{C_3, C_5, C_7, \ldots\} \cup \mathcal{K}$ and $n \leq 64$.

This chapter contains joint work with Brendan McKay.

3.2 Introduction

The history of extremal graph theory starts in 1907, with the theorem of Mantel [141] that determines for a given n, the minimum number, e, such that any graph with n vertices and e edges contains a triangle (C_3 or K_3). Equivalently, this answers the question of determining the maximum number, e, for a given n, such that there exists a graph with n vertices and e edges without any triangle.Turán answered a more general version of this question. He proved that the maximum number of edges in a K_r-free graph with n vertices is $\frac{r-2}{r-1}.\frac{n^2}{2}$ [195, 196].

This problem can be further generalised to the problem of determining the maximum number, $ex(n, \mathcal{H})$, for a given n and a prescribed set of subgraphs $\mathcal{H}$, such

that there exists a graph with n vertices and $ex(n, \mathcal{H})$ edges without any subgraph in $\mathcal{H}$. Such problems that maximise the number of edges while avoiding some given subgraphs are called Turán type extremal problems and $ex(n, \mathcal{H})$ is called the *Turán number* of $\mathcal{H}$.

On the other hand, extremal graph theory includes a wider range of problems where for a graph property P, an invariant u, and a set of graphs A, we wish to find the minimum value of m such that every graph in A which has u larger than m has property P. The graphs in A with property P and the invariant u equal to m are called *extremal graphs*. Determining the structure and the uniqueness of extremal graphs are other important questions in extremal graph theory.

In a Turán type problem, P is the property of containing a subgraph in a given set of forbidden subgraphs, u is the number of edges and A is the set of all graphs with a given number of vertices. The problems in extremal graph theory fall into different categories based on the type of properties and the invariants they consider. Füredi and Simonovits [89] provided a complete survey on extremal graphs, especially, the Turán type problems for cycles. They described the importance of the Turán numbers as follows:

> Turán type extremal results (and Ramsey results as well) can often be applied in Mathematics, even outside of Combinatorics. Turán himself explained this applicability by the fact that – in his opinion – the extremal graph results were generalizations of the Pigeon Hole Principle.

In this chapter, we consider a set of Turán type problems where the set of forbidden subgraphs only contains cycles. Let $\mathcal{C}$ be a given set of cycles and $ex(n, \mathcal{C})$ be the maximum number of edges in a graph without any cycle in $\mathcal{C}$. Despite many efforts of different researchers, the exact value for $ex(n, \mathcal{C})$ and the number or structure of extremal graphs, for almost any set of cycles is rarely known. We introduce a method to practically attack this problem via graph generation. With this method, we furthered almost all known results on the exact value of Turán numbers of cycles and the set of all extremal graphs.

3.2.1 Definitions and Preliminaries

We define $\mathcal{Z} = \{C_3, C_4, C_5, \ldots\}$ to be the set of all cycles, $\mathcal{B} = \{C_3, C_5, C_7, \ldots\}$ to be the set of all odd cycles, and $\mathcal{G}$ to be the set of all unlabelled graphs. For each set of cycles, $\mathcal{C} \subseteq \mathcal{Z}$, we define $\mathcal{L}_{\mathcal{C}}$ and $\mathcal{G}_{\mathcal{C}}$ to be the set of all labelled and unlabelled graphs not containing any cycle in $\mathcal{C}$, respectively. Therefore, $\mathcal{G}_{\mathcal{B}}$ is the set of all unlabelled bipartite graphs and $\mathcal{G}_{\mathcal{C}\cup\mathcal{B}}$ is the set of all unlabelled bipartite graphs not containing any cycle in $\mathcal{C}$. For simplicity, when C is of size one, say $\mathcal{C} = \{C_i\}$, we may alternatively use $\mathcal{G}_{C_i}$ to denote $\mathcal{G}_{\mathcal{C}}$.

For each graph, X, if the set of vertices of minimum degree is independent, we say X is of type A and we denote this by $type(X) = A$. But if there is an edge between

two vertices of minimum degree, then we say X is of type B and we denote this by $type(X) = B$.

Let X be a graph. The set of vertices, the set of edges, the minimum degree and the set of all vertices of minimum degree in X are denoted by $V(X)$, E(X), $\delta(X)$ and $mins(X)$, respectively. For any vertex, $u \in V(X)$, we denote the degree of u in X by $deg_X(u)$ or $deg(u)$ when it is clear, and the set of neighbours of u in X by $N_X(u)$ or $N(u)$ when it is clear. The graph obtained from X by removing a vertex, u, and its incident edges is denoted by $X - u$.

Given a positive integer, n, and a set of cycles, C, we denote by $ex(n, C)$, the Turán number of n and C, that is the maximum number of edges in graphs with n vertices without any cycle in C. The set of all graphs with n vertices, $ex(n, C)$ edges and without any cycle in C is shown by $EX(n, C)$.

3.3 Literature Review

The Turán type extremal problems for several sets of cycles have been attacked by many different researchers with several approaches. Some used more theoretical approaches to determine the asymptotic behavior of $ex(n, \mathcal{C})$. Some others applied computer search techniques and other algorithms such as hill-climbing and simulated annealing techniques. In some approaches, known extremal graphs for other sets of cycles are considered and forbidden cycles are destroyed. Although there are still few known results for the exact value of Turán numbers for cycles, several bounds are obtained with such practical techniques. Nonetheless, there is still a large gap between upper and lower bounds for many sets of cycles. In general, it is more difficult to determine $ex(n, \mathcal{F})$ when $\mathcal{F}$ contains bipartite graphs [10]. This means determining $ex(n, \mathcal{C})$ when $\mathcal{C}$ contains an even cycle tends to be hard. In this section, we discuss the known results for different sets of cycles. Readers are referred to [89] for a complete survey by Füredi and Simonovits.

3.3.1 Turán Numbers for C_4

Since we have $C_3 = K_3$, the cycle C_4 is the smallest cycle for which the exact value of Turán number is not known for all n. Therefore, the problem of determining $ex(n, \{C_4\})$ and $EX(n, \{C_4\})$ is of special interest amongst other Turán type extremal problems. This problem was first posed by Erdős in 1938 and has been studied by many authors since then. But unfortunately, it did not appear to be simple.

In 1984, Bialostocki and Schonheim [27] presented the exact value of $ex(n, \{C_4\})$ for $n = 10, 11$. McCuaig calculated $ex(n, \{C_4\})$ for $n \leq 21$ (unpublished letter, 1985 [89]). In 1989, Clapham, Flockhart and Sheehan [53] determined all extremal graphs for $n \leq 21$. They also introduced recursive upper and lower bounds for $ex(n, \{C_4\})$ for

all n. Later, in 1992, Yuansheng and Rowlinson [210] determined $ex(n, \{C_4\})$ and all members of $EX(n, \{C_4\})$ for $n \leq 31$ using computer searches. In 2009, Shao and Xu and Xu [187] proved $ex(32, \{C_4\}) = 92$. Recently, Wu, Sun and Radziszowski [206] used these results to calculate some Ramsey numbers.

We extend the results on the exact value of $ex(n, \{C_4\})$ to $n \leq 39$ and obtain tight bounds for $n \leq 49$.

A Related Open Problem

Erdős and Simonovits [77] conjectured if G is any n-vertex graph with $ex(n, \{C_4\}) + 1$ edges then G must contain at least $n^{1/2} + o(n^{1/2})$ copies of C_4.

3.3.1.1 Some Bounds and Asymptotic Behavior

Asymptotically we have $ex(n, \{C_4\}) \sim \frac{n^{3/2}}{2}$. More precisely, it is known that $ex(n, \{C_4\}) \leq \frac{1}{2}n^{3/2} + o(n^{3/2})$ for every $n \geq 1$ (see [32]). We have $ex(n, K_{a,b}) \leq \frac{1}{2}\sqrt[a]{b-1}\ \ n^{2-(1/a)} + O(n)$ and $ex(n, K_{a,b}) \leq \frac{1}{2}\sqrt[a]{b-1}\ \ n^{2-(1/a)} + \frac{a-1}{2}n$ [89]. Since C_4 is $K_{2,2}$ these inequalities give $ex(n, \{C_4\}) \leq \frac{1}{2}n^{3/2} + O(n)$ and $ex(n, \{C_4\}) \leq \frac{1}{2}n^{3/4} + \frac{1}{2}n$. Also, $ex(n, \{C_4\}) = (\frac{1}{2} + o(1))n^{3/2}$ and $ex(n, \{C_4\}) = \frac{1}{2}n^{3/2} + O(n^{3/2-c})$ [74, 44, 123].

Reiman [183] provided the upper bound of $ex(n, \{C_4\}) \leq \frac{n}{4}(1 + \sqrt{4n-3})$ that is not very sharp. Nonetheless, with a construction known as the Erdős-Rényi graph [74, 73], Erdős, Rényi, and Sós proved this bound is asymptotically correct.

Known Results for $ex(q^2 + q + 1, \{C_4\})$

In 1996, Erdős, Rényi and Sós [74] and Brown [44], independently proved that for each prime power, q, we have $ex(q^2 + q + 1, \{C_4\}) \geq \frac{1}{2}q(q+1)^2$. They used polarity graphs of projective planes that were previously used in [73]. In [74], it is also shown that $ex(q^2 + q + 1, \{C_4\}) < \frac{1}{2}(q^2 + q + 1)(q + 1)$. Erdős [70, 72] conjectured there are no better constructions and the inequality $ex(q^2 + q + 1, \{C_4\}) \geq \frac{1}{2}q(q+1)^2$ is sharp for each prime power q. Füredi in [84, 85, 86] proved this conjecture and showed the equality holds for infinitely many q. He first proved if $q = 2^k$, then Erdős' conjecture holds [84]. Next, he showed [85, 86] if $q \neq 1, 7, 9, 11, 13$ then $ex(q^2 + q + 1, \{C_4\}) \leq \frac{1}{2}q(q+1)^2$ and if q is also a power of a prime, then $ex(q^2 + q + 1, \{C_4\}) = \frac{1}{2}q(q+1)^2$ and the only graphs with $q^2 + q + 1$ vertices and $\frac{1}{2}q(q+1)^2$ edges that do not contain 4-cycles are orthogonal polarity graphs of finite projective planes.
Thus, Füredi determined the first exact result of $ex(n, \{C_4\})$ for infinitely many n. His results along with the constructions in [44, 74] are the most important contributions to the 4-cycle Turán problem.

Other General Results

Abreu, Balbuena and Labbate [8] introduced some general lower bounds by deleting carefully some chosen vertices from the Erdős-Rényi graph. It is not known if any of these bounds are sharp. They proved that for any odd prime power, q, we have $ex(q^2-q-2,C_4) \geq \frac{1}{2}q^3-q^2-\frac{q}{2}+1$ and for any even prime power, q, we have $ex(q^2-q-2,C_4) \geq \frac{1}{2}q^3-q^2$. They conjectured these bounds are the best possible but in 2014 Tait and Timmons [191] improved this result for odd cases. They proved for any odd prime power, q, $ex(q^2-q-2,C_4) \geq \frac{1}{2}q^3-q^2-O(q^{3/4})$. Firke, Kosek, Nash and Williford [83] proved for even q ($q > q_0$), $ex(q^2+q,C_4) \leq \frac{1}{2}q(q+1)^2-q$ where equality holds for all q that are powers of 2. Recently, Yali, Yongqi, Rui and Radziszowski [207] proved that $ex(q^2+q+2,C_4) < \frac{1}{2}(q+1)(q^2+q+2)$.

Summary of the Previously Known Results About $ex(n,\{C_4\})$

The most important previously known results about the exact values of $ex(n,\{C_4\})$ are summarised in Tables 3.1 and 3.2. In all the tables in this chapter, except for Tables 3.2 and 3.9, the entry at a row lablled as i and a column labelled as j ($0 \leq i \leq 60$ and $0 \leq j \leq 9$) presents the relevant Turán where the number of vertices is $n = i + j$.

	0	1	2	3	4	5	6	7	8	9
0	0	0	1	3	4	6	7	9	11	13
10	16	18	21	24	27	30	33	36	39	42
20	46	50	52	56	59	63	67	71	76	80
30	85	90	92							

Table 3.1: The previously known results [27, 53, 210, 187] for the exact values of $ex(n,\{C_4\})$

n	Known Results
q^2+q-2	Lower bounds of $\frac{1}{2}q^3-q^2-O(q^{3/4})$ for odd prime power q [191] and $\frac{1}{2}q^3-q^2$ for even prime power q[8]
q^2+q	$\frac{1}{2}q(q+1)^2-q$ is an upper bound for $q > q_0$ even and is the exact value for q power of 2 [83]
q^2+q+1	The exact values of $\frac{1}{2}q(q+1)^2$ for all prime powers $q \neq 7,9,11,13$ [84, 85, 86], The upper bound of $\frac{1}{2}q(q+1)^2$ for $q \neq 7,9,11,13$ [85, 86], The lower bound of $\frac{1}{2}q(q+1)^2$ for prime power q [44, 74], The upper bound of $\frac{1}{2}(q^2+q+1)(q+1)$ [74].
q^2+q+2	The upper bound of $\frac{1}{2}(q+1)(q^2+q+2)$ [207]

Table 3.2: Some general results about $ex(n,\{C_4\})$

3.3.2 Turán Numbers for $\{C_3, C_4\}$

In the 1970s, Erdős conjectured $ex(n, \{C_3, C_4\}) = \frac{1}{2\sqrt{2}} n^{3/2} + o(n^{3/2})$ [71]. In the early 1990s, Garnik, Kwong, Lazebnik and Nieuwejaar [91, 92, 93] provided the exact value of $ex(n, \{C_3, C_4\})$ for $n \leq 30$ and $n = 50$. They also provided constructive lower bounds for $n \leq 200$ using computer search techniques such as hill-climbing. Moreover, they enumerated all members of $EX(n, \{C_3, C_4\})$ for $n \leq 10$. Dutton and Brigham [66] determined the sharp upper bound of $ex(n, \{C_3, C_4\}) \leq n\sqrt{n-1}/2$. In 2001, Wang, Dueck and MacMillan [202] improved the lower bounds of Garnik and others, and provided new lower bounds for $n \geq 6$ using a simulated annealing technique. Later in 2010, Abajo, Balbuena and Diánez [1] improved the lower bounds for some $100 < n < 180$. Two years later, further improvement for some $100 < n < 260$ was presented in [6] by Abajo and Diánez. Recently Balbuena and others [23] calculated the exact value for $n = 31, 32$ and improved lower bounds for some $69 \leq n < 230$. Table 3.3 contains the previously known exact values of $ex(n, \{C_3, C_4\})$. We extend these results to $n \leq 52$.

	0	1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	5	6	8	10	12
10	15	16	18	21	23	26	28	31	34	38
20	41	44	47	50	54	57	61	65	68	72
30	76	80	85							
40										
50	175									

Table 3.3: The previously known results [91, 92, 93, 23] for the exact values of $ex(n, \{C_3, C_4\})$

3.3.3 Turán Numbers for $\{C_4, C_5\}$

In 2009, Yongqi, Xiaohui, Yuansheng and Lei [209] calculated the exact values of $ex(n, \{C_4, C_5\})$ for $n \leq 21$ as presented in Table 3.4. We extend this to $n \leq 43$.

	0	1	2	3	4	5	6	7	8	9
0	0	0	1	3	4	6	7	9	10	12
10	14	16	18	20	23	25	28	30	33	35
20	38	42								

Table 3.4: The previously known results [209] for the exact values of $ex(n, \{C_4, C_5\})$

3.3.4 Turán Numbers for $\{C_3, C_4, C_5\}$

In 1996, Dong and Koh [65] gave a sharp upper bound for $ex(n, \{C_3, C_4, C_5\})$. In 2004, Yang, Lin, Dong and Zhao [208] calculated the exact values of $ex(n, \{C_3, C_4, C_5\})$ for

$n \leq 42$ and all the related extremal graphs. Abajo and Diánez [4], in 2010, provided sharp bounds for $43 \leq n \leq 61$ and calculated the exact value for $n = 62$. Table 3.5 contains these exact values of $ex(n, \{C_3, C_4, C_5\})$. We extend these results to $n \leq 63$.

	0	1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	4	6	7	9	10
10	12	14	16	18	21	22	24	26	29	31
20	34	36	39	42	45	48	52	53	56	58
30	61	64	67	70	74	77	81	84	88	92
40	96	100	105							
50										
60			186							

Table 3.5: The previously known results [208, 4] for the exact values of $ex(n, \{C_3, C_4, C_5\})$

3.3.5 Turán Numbers for $\{C_3, C_4, C_5, C_6\}$

In 1996, Dong and Koh [65] gave a sharp upper bound for $ex(n, \{C_3, C_4, C_5, C_6\})$. The exact value of $ex(n, \{C_3, C_4, C_5, C_6\})$ for $n \leq 16$ is deduced from the more general formulas presented by Abajo and Diánez in 2007 [3]. Tang, Lin, Balbuena and Miller in 2009[192] provided lower bounds for $n \leq 39$. In 2010, Abajo and Diánez[4] extended these lower bounds and provided upper bounds for $29 \leq n \leq 49$. They also calculated the exact values for $n \leq 28$. Also, in collaboration with Balbuena [1], they furthered the lower bounds for $50 \leq n \leq 300$. These results are presented in Table 3.6. We extend the known results for the exact values of $ex(n, \{C_3, C_4, C_5, C_6\})$ to $n \leq 53$.

	0	1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	4	5	7	8	9
10	11	12	14	15	17	18	20	22	23	25
20	27	29	31	33	36	37	39	41	43	

Table 3.6: The previously known results [3, 4] for the exact values of $ex(n, \{C_3, C_4, C_5, C_6\})$

3.3.6 Turán Numbers for $\{C_3, C_4, C_5, C_6, C_7\}$

The exact value of $ex(n, \{C_3, C_4, C_5, C_6, C_7\})$ for $n \leq 19$ can be deduced from the more general formulas presented by Abajo and Diánez in 2007 [3]. Tang, Lin, Balbuena and Miller in 2009 [192] provided lower bounds for $n \leq 39$. In 2010, Abajo and Diánez[4] improved and extended these lower bounds and provided upper bounds for $n \leq 79$. They also calculated the exact values for $n \leq 36$ and $n = 80$. Also, in 2012 and in collaboration with Balbuena [2], they further improved the lower bounds for $96 \leq n \leq 193$ and determined $ex(170, \{C_3, ..., C_7\}) = 425$. These results for exact

values of $ex(n, \{C_3, C_4, C_5, C_6, C_7\})$ are presented in Table 3.7. We extend these results to $n \leq 60$.

	0	1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	4	5	6	8	9
10	10	12	13	14	16	18	19	20	22	24
20	25	27	29	30	32	34	36	38	40	42
30	45	46	47	49	51	53	55			
. . .										
80	160									
. . .										
170	425									

Table 3.7: The previously known results [3, 4, 2] for the exact values of $ex(n, \{C_3, C_4, C_5, C_6, C_7\})$

3.3.7 Extremal Graphs for Single Cycles, $ex(n, \{C_i\})$

Bondy in [33] proved a conjecture of Erdős that every graph of order n and size at least $\frac{1}{2}(n^2 - 5n + 14)$ has a cycle of length $n - 1$. This means $ex(n, \{C_{n-1}\}) < \frac{1}{2}(n^2 - 5n + 14)$.

3.3.7.1 Even Cycles; $ex(n, \{C_{2k}\})$

In 1965, Erdős [69] stated that for every k there is a c such that any graph on n vertices with $cn^{1+1/k}$ edges has a C_{2k}. This was proved by Bondy and Simonovits [34] as a more general theorem that states any graph on n vertices with $100kn^{1+1/k}$ edges has a C_{2h} for every integer h where $k \leq h \leq kn^{1/k}$.

Reiman [183, 188] and Brown [44] constructed graphs on n vertices with $\Omega(n^{3/2})$ edges without C_4. Benson [26] constructed graphs on n vertices with $\Omega(n^{4/3})$ edges without C_6, and with $\Omega(n^{5/4})$ edges without C_{10}. Reiman, Brown, and Benson used finite projective geometry in their construction. For infinitely many n, Wenger in [203] presents a new construction for graphs with n vertices, $(n/2)^{3/2}$ edges without any C_4, $(n/2)^{4/3}$ edges without C_6, and $(n/2)^{6/5}$ edges without C_{10}. This gives lower bounds for $ex(n, \{C_4\})$, $ex(n, \{C_6\})$ and $ex(n, \{C_{10}\})$ for infinitely many n.

In 1982, Erdős and Simonovits [76] conjectured $ex(n, \{C_{2k}\})$ is asymptotically $\frac{1}{2}n^{1+1/k}$ as n tends to infinity. Lazebnik, Ustimenko and Woldar [134] showed $ex(n, \{C_6\}) \geq \frac{1}{2}n^{3/4} + O(n)$ for infinitely many n by constructing dense C_6-free graphs. In 2005,

Füredi, Naor and Verstraëte [88] constructs a counterexample to this conjecture of Erdős for hexagon(C_6).

In 2005, Kühn and Osthus [126] proved $ex(n, \{C_{2k}\}) \leq 2(k-1)ex(n, \{C_4, C_{2k}\})$.

Hexagon

As stated above, it was shown that $ex(n, \{C_6\}) \geq \frac{1}{2}n^{3/4} + O(n)$ for infinitely many n [134]. Yuansheng and others [211] determined the graphs in $EX(n, \{C_6\})$ for $n \leq 26$.

3.3.7.2 Odd Cycles; $ex(n, \{C_{2k+1}\})$

The value of $ex(n, \{C_{2k+1}\})$ can be determined from the works of Bondy [33], Woodall [205], Bollobás [32] and Dzido [68]. Füredi and Gunderson [87] completed the list of extremal graphs in $EX(n, \{C_{2k+1}\})$ and provided new proof for determining the value of $ex(n, \{C_{2k+1}\})$.

Balister, Bollobás, Riordan and Schelp in [24] considered the extremal graphs without a given odd cycle among graphs with a given maximum degree. They define $f(n, \Delta; C_{2k+1})$ to be the maximal number of edges in a graph of order n and maximum degree Δ that contains no cycles of length C_{2k+1}. They showed that for $\frac{n}{2} \leq \Delta \leq n-k-1$ and sufficiently large n we have $f(n, \Delta; C_{2k+1}) = \Delta(n-\Delta)$ and a complete bipartite is the unique extremal graph. This gives a lower bound for $ex(n, \{C_{2k+1}\})$ since $ex(n, \{C_{2k+1}\}) \geq f(n, \Delta; C_{2k+1})$.

3.3.8 Extremal Graphs with Bounded Girth

The set $EX(n, \{C_3, C_4, \ldots, C_k\})$ contains the extremal graphs with girth at least $k+1$. This type of extremal graphs can be used in studying cages and related problems. Cages are widely studied [7]. They are the smallest regular graphs with given degree and girth. In [5], Abajo and Diánez constructed infinitely many extremal graphs for sets of forbidden cycles of form $\{C_3, C_4, \ldots, C_k\}$. They have also determined the exact value of $ex(n, \{C_3, C_4, \ldots, C_k\})$ for $n \leq \lfloor (16k-15)/5 \rfloor$ [3].

On natural question about this type of extremal graphs is whether the girth of graphs in $EX(n, \{C_3, C_4, \ldots, C_k\})$ is $k+1$? The answer for some cases is affirmative and for some others is negative. This question is studied by different authors [21, 135, 2]. In these papers, as well as the girth, some other properties of the extremal graphs avoiding all cycles of length at most k have been established.

A number of upper bounds are determined for the maximum number of edges, e, in a graph with given number of vertices ,n, and girth, g. As such is the one deduced from $g \leq 2 + 2\log_k(n/4)$ when $k = \lfloor e/n \rfloor \geq 2$ [66]. In 1991, Dutton and Brigham [66]

determined an upper bound for $ex(n, \{C_3, C_4, \ldots, C_k\})$ for all $n \geq k \geq 6$.

Also, several lower bounds such as $ex(n, \{C_3, C_4, \ldots, C_k\}) = \Omega(n^{1+\frac{1}{k-1}})$ and $ex(n, \{C_3, C_4, \ldots, C_{2k+1}\}) = \Omega(n^{1+\frac{2}{2k+3}})$ have been determined by different authors [26, 130, 131, 132, 188, 197, 203, 204]. Many of them are constructive lower bounds such as the lower bound of $ex(n, \{C_3, C_4, \ldots, C_{2k+1}\}) = \Omega(n^{1+\frac{2}{3k-3+\epsilon}})$ determined by Lazebnik, Ustimenko and Woldar [133].

3.3.9 Extremal Bipartite Graphs

Erdős and Simonovits have made several conjectures on Turán numbers for bipartite graphs[76]. As such they conjectured that for every finite family $\mathcal{F}$ of graphs, there exists k such that $ex(n, \mathcal{F} \cup \{C_k\}) \sim ex(n, \mathcal{F} \cup \mathcal{B})$ as $n \to \infty$. That is extremal $\mathcal{F}$-free graphs should be near-bipartite if $\mathcal{F}$ contains a long enough odd cycle.

This conjecture is proved when $\mathcal{F}$ consists only of non-bipartite graphs [75] and for $\mathcal{F} = \{C_4, C_6, \ldots, C_{2l}\}$ when $l \in \{2, 3, 5\}$ [121]. Moreover for large enough n, it is shown in both cases the extremal $\mathcal{F} \cup \{C_k\})$-free graphs are exactly bipartite. In the latter the extremal graphs are the bipartite incidence graphs of rank two geometries called generalized polygons [193]. The case $\mathcal{F} = \{C_4\}$ with $k = 5$ was also established by Erdős and Simonovits in [76], i.e., they showed $ex(n, \{C_4, C_5\}) \sim ex(n, \{C_4\} \cup \mathcal{B})$.

Allen and others [10] proved this conjecture for some other families of bipartite graphs using a sparse version of Szemeredi's Regularity Lemma. They have also proved $ex(n, \mathcal{F} \cup \{C_k\}) \sim ex(n, \mathcal{F} \cup \mathcal{B})$ for any $k \geq 5$ and the extremal graphs are nearly bipartite.

3.3.9.1 $ex(n, \{C_4\} \cup \mathcal{B})$

Since C_4 can also be considered as $K_{2,2}$, Turán numbers for bipartite graphs avoiding C_4 are closely related to Zarankiewicz numbers—the maximum number of edges a $K_{s,t}$-free bipartite graph of size (m, n) can have—denoted by $Z_{s,t}(m, n)$. In fact, $ex(n, \{C_4\} \cup \mathcal{B}) = \max_{a+b=n} Z_{2,2}(a, b)$. Zarankiewicz numbers can be used to bound bipartite Ramsey numbers, $b(m, n)$, that is the least positive integer b such that if the edges of $K(b, b)$ are coloured with red and blue, then there always exists a blue $K(m, m)$ or a red $K(n, n)$ [94].

Erdős proved the upper bound of $3n\sqrt{n}$ for the size of a bipartite graphs without C_4 where the two parts have an equal number of vertices, n.

The numbers in Table 3.8 are the exact values of $ex(n, \{C_4\} \cup \mathcal{B})$ calculated in [94] or are deduced from Zarankiewicz numbers [58, 67]. We extend these results to $n \leq 63$. In Section 3.9, we introduce new results on the exact values of Zarankiewicz numbers that are obtained by investigating the extremal graphs in $EX(n, \{C_4\} \cup \mathcal{B})$ we generated.

	0	1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	4	6	7	9	10
10	12	14	16	18	21	22	24	26	29	31
20	34	36	39	42	45	48	52	53	56	58
30	61	64	67	70	74	77	81	84	88	92
40	96	100	105	106						

Table 3.8: The previously known results [94, 58, 67] for the exact values of $ex(n, \{C_4\} \cup \mathcal{B})$

3.3.10 Extremal Bipartite Graphs with a Bounded Girth

The results on extremal bipartite graphs with bounded girth can be used in studying bi-regular cages that is a well-studied problem [20]. In [22], Balbuena, García-Vázquez, Marcote and Valenzuela considered extremal bipartite graphs with fixed parts and a bounded girth where $EX(m, n; \{C_4, \ldots, C_{2k}\})$ is the family of bipartite graphs of maximum size with m and n vertices in each part, that contain no cycle of length less than or equal to $2k$, and $ex(m, n; \{C_4, \ldots, C_{2k}\})$ is the number of edges of a graph in $EX(m, n; \{C_4, \ldots, C_{2k}\})$. They presented some results about the girth and connectivity of graphs in $EX(m, n; \{C_4, \ldots, C_{2k}\})$ and determined some exact values of $ex(m, n; \{C_4, \ldots, C_{2k}\})$. This problem has been studied by Lam [128, 129]. He determined some upper bounds for $ex(m, n; \{C_4, \ldots, C_{2(2k+1)}\})$.

3.4 The Generation Algorithm

Given a positive integer, n, and a set of cycles, $\mathcal{C}$, we aim to calculate $ex(n,\mathcal{C})$ and $EX(n,\mathcal{C})$. By definition, $ex(n,\mathcal{C})$ is the maximum number of edges in a graph in $\mathcal{G}_\mathcal{C}$ with n vertices, and $EX(n,\mathcal{C})$ is a subset of $\mathcal{G}_\mathcal{C}$ where the number of vertices is n and the number of edges is $ex(n,\mathcal{C})$. Therefore, we can generate all graphs in $\mathcal{G}_\mathcal{C}$ with n vertices and then consider those with the maximum number of edges.

The graphs in $\mathcal{G}_\mathcal{C}$ with n vertices can be generated from graphs in $\mathcal{G}_\mathcal{C}$ with $n-1$ vertices, as we explain later in this section. Thus, we can generate all graphs in $\mathcal{G}_\mathcal{C}$ up to n vertices recursively using the method of *Generation by Canonical Construction Path*, (*GCCP*), starting from the trivial graph of K_1 where any generated node on the generation tree is in $\mathcal{G}_\mathcal{C}$. The method *GCCP* is described in details in other chapters.

Unfortunately, this naive generation process is not practically efficient without proper pruning since too many graphs in $\mathcal{G}_\mathcal{C}$ are produced that are far from being extremal or having an extremal descendant. We have developed some tools to extensively prune the search tree where through a reverse engineering technique, we avoid construction of many nodes with no extremal graphs in the subtree rooted from them. This technique is explained in Section 3.4.3.

In this section, we discuss the definition of the extension, reduction and the genuine reduction that we used to generate graphs in $\mathcal{G}_\mathcal{C}$ using the *GCCP* method.

3.4.1 Extensions and Reductions

Our extension is adding a vertex and joining it to a subset of vertices without creating a forbidden cycle. A reduction, therefore, is the operation of removing a vertex and incident edges.

In order to extend a graph, $G \in \mathcal{G}_\mathcal{C}$, to larger graphs in $\mathcal{G}_\mathcal{C}$, we first determine all subsets of vertices that can share a new neighbour without creating a forbidden cycle in $\mathcal{C}$. We then calculate the orbits of these subsets of vertices. For each orbit, we choose one subset as the representative and the extensions are applied only for these representatives. The following lemmas state that all graphs generated this way are in $\mathcal{G}_\mathcal{C}$.

Lemma 3.4.1. *Let $G = (V,E)$ be a bipartite graph and G' be a graph obtained from G by adding a vertex and joining it to vertices in $S \subseteq V$. The graph G' is bipartite iff there is no path in G of an odd length between any two vertices in S.*

Lemma 3.4.2. *Let $G = (V,E)$ be a graph with no cycle of length k and G' be a graph obtained from G by adding a vertex and joining it to vertices in $S \subseteq V$. The graph G' has a cycle of length k iff there is a path in G of length $k-2$ between two vertices in S.*

Therefore:

Theorem 3.4.1. *Let $C \subseteq \mathcal{Z} = \{C_3, C_4, C_5, \ldots\}$ be a set of cycles, $G = (V, E) \in \mathcal{G}_C$, $S \subseteq V$ and G' be a graph obtained from extending G by adding a new vertex and joining it to vertices in S. We have $G' \in \mathcal{G}_C$ iff for each k, where $C_k \in C$, there is no path in G of length $k-2$ between any two vertices in S.*

According to the theorem above, if $\mathcal{B} = \{C_3, C_5, C_7, \ldots\} \subseteq C$, then there must not be any path in G of an odd length between any two vertices in S to have $G' \in \mathcal{G}_C$,.

3.4.2 Genuine Reductions

According to the *GCCP* method, when we extend a graph, X in $\mathcal{G}_C$, to another graph, Y, we only accept Y if it is constructed from X by an extension whose inverse operation is a genuine reduction. Here we define a reduction to be genuine if it reduces a winning vertex. Therefore, if X is extended to Y, we accept Y only if it is constructed from X by adding a vertex that is winning in Y. We define a vertex, v, to be winning if:

1. v is in D where D consists of all vertices of minimum degree, d, in Y,
2. v is in $H \subseteq D$ where H is the set of vertices of maximum degree in the subgraph induced by D,
3. v is in K where $K \subseteq H$ consists of all vertices in H that share a neighbour of degree $d+1$ with the greatest number of vertices (counting itself) in H, and
4. v is in the same orbit as the vertex w where $w \in K$ is the vertex labelled lexicographically last by the software *nauty* among all vertices in K.

More precisely, we define the canonical reduction by assigning a 4-tuple $x(v) = (x_0(v), x_1(v), x_2(v), x_3(v))$ to each vertex, v, and the winning vertices are those whose corresponding 4-tuples are lexicographically largest.We aimed to define an efficient genuine reduction. Therefore, the values of x_0, x_1, x_2 and x_3 are combinatorial invariants of increasing discriminating power and computational cost. The definition of these vertex invariants are as follows:

Let $G \in \mathcal{L}_C$ be a graph with n vertices, minimum degree d and S be the set of all vertices in $mins(G)$ having most number of neighbours of degree d among the vertices in $mins(G)$. For every vertex v in $V(G)$, we define:

- $x_0(v) = n - deg(v)$,
- $x_1(v)$ is the number of vertices of degree d that are adjacent to v,
- $x_2(v)$ is the number of vertices in S sharing a neighbour of degree $d+1$ with v, and

- $x_3(d)$ is the largest canonical label of a vertex in the orbit containing the vertex v.

Each x_i is computed only if $(x_0(v'), \ldots, x_{i-1}(v'))$ is not the unique lexicographically largest with v' being the last vertex added to G during the generation process of G. This means each x_i is computed only if the values of earlier elements of x fail to determine v' is not winning while they also fail to distinguish it as the unique vertex with the largest value of x. Hence, if there are more than one vertex with the largest value of (x_0, x_1, x_2) and our newly added vertex is one of them, then using *nauty*, we compute orbits and a canonical labelling of the vertices of G and define $x_3(v')$ to be the largest label among the canonical labels of vertices in the same orbit as v'.

Since x_0, x_1 and x_2 are invariant under isomorphisms, all vertices in one orbit of the automorphism group have the same values of (x_0, x_1, x_2). The value of x_3 has an even stronger property: two vertices are in a same orbit if and only if they have the same value of x_3. Together with the definition of canonical labelling, this implies the following Lemma:

Lemma 3.4.3. *Let $X_1, X_2 \in \mathcal{L}_\mathcal{C}$ be two isomorphic graphs having at least two vertices each and let γ be an isomorphism from X_1 to X_2. If v_1 and v_2 are, respectively, the vertices in X_1 and X_2 having largest 4-tuples x, then $v_1\gamma$ is in the same orbit as v_2 under the action of the automorphism group of X_2, $Aut(X_2)$. Furthermore, removing v_1 from X_1 produces a graph isomorphic to the graph obtained from X_2 by removing v_2 .*

Although the generation algorithm would remain exhaustive and isomorph-free when only x_3 is computed, embedding x_0, x_1 and x_2 in the definition of winning vertices is important for efficiency purposes. While the orbits of the automorphism group and a canonical form must be computed to calculate the value of x_3, the earlier elements of the 4-tuples are only based on local properties of vertices that are much cheaper to compute. Furthermore, these properties provide lookaheads in the generation process that improve its efficiency. For example, when extending a graph, $X \in \mathcal{L}_\mathcal{C}$, by adding a new vertex and joining it to a subset of vertices, the value of x_0 for this vertex in the extended graph can be easily determined on the level of X. Also, we can decide, on the level of X, whether a new vertex would be the only vertex with the largest value of x_0 in the extended graph. Therefore, with our definition of the genuine reduction, we avoid the construction of a lot of children that would be rejected afterwards.

To extend a graph, X, we first construct the list of all valid subsets, that is, the subsets of vertices that can share a new neighbour without creating a forbidden cycle. This list is constructed based on Theorem 3.4.1. We then determine the automorphism group of X, $Aut(X)$ and the orbits of valid subsets which is the result of the action of the automorphism group on the list of valid subsets. In fact, an automorphism, which acts on the vertices of X, deduces a permutation acting on the list of valid subsets. We

extend X to bigger graphs, for exactly one subset in each orbit where we add a vertex to X and join the new vertex to the vertices in that subset.

According to Lemma 3.4.3, any two isomorphic accepted graphs are from the same parent and are generated by equivalent extensions. Since in the method of *GCCP*, only non-equivalent extensions are applied to each graph (as we explained above), no two isomorphic accepted graphs can appear throughout the generation process. We use this fact to prove the following lemma where $\mathcal{L} = \mathcal{L}_C$ for an arbitrary set of cycles. Given a graph $X \in \mathcal{L}$, we define $\mathcal{L}(X)$ to be the set of all labelled graphs on the generation tree descendant from X in which the extension is adding a vertex and for each k, we define $\mathcal{L}^k(X)$ to be the set of all graphs in $\mathcal{L}(X)$ with k vertices.

Lemma 3.4.4. *Let H_{n-1} be a set of graphs consisting of exactly one representative for each isomorphism class of graphs in $\mathcal{L}^{n-1}(X)$. If for each graph in H_{n-1}, the extensions of adding a vertex is applied to exactly one subset of vertices in each orbit of the list of valid subsets of vertices, and a generated graph in $\mathcal{L}^n(X)$ is accepted if and only if the vertex added last has the lexicographically largest value of (x_0, x_1, x_2, x_3), then exactly one representative of each isomorphism class of graphs in $\mathcal{L}^n(X)$ is accepted.*

Proof. Let G_1 and G_2 be two isomorphic accepted graphs in $\mathcal{L}^n(X)$. From Lemma 3.4.3, we know G_1 and G_2 are generated from the same parent, G, in $\mathcal{L}^{n-1}(X)$. Assume G_1 is extended from G by adding a vertex, v_1, and joining it to the vertices in $S_1 \subseteq V(G)$, and G_2 is also extended from G by adding a vertex, v_2, and joining it to the vertices in $S_2 \subseteq V(G)$. According to Lemma 3.4.3, the newly added vertices v_1 and v_2 must both have the lexicographically largest value of (x_0, x_1, x_2, x_3) in G_1 and G_2 respectively, otherwise G_1 or G_2 would not be accepted. Therefore, there is an isomorphism, γ, from G_1 to G_2 that maps v_1 to v_2. This means the automorphism in G, deduced by γ, maps S_1 to S_2, showing that S_1 and S_2 are equivalent under the Automorphism group of G, $Aut(G)$ and that is in contrary to our procedure. Therefore, no two isomorphic copies are accepted. In the following, we prove the exhaustiveness, that is, for each isomorphism class of graphs in $\mathcal{L}^n(X)$, at least one representative is generated and accepted.

Let G_1 be an arbitrary graph in $\mathcal{L}^n(X)$. Consider a graph, $G_2 \in \mathcal{L}^n(X)$, that is isomorphic to G_1 in which the largest labelled vertex, v_n, is winning. Let $S = N_{G_2}(n)$ and G be the graph obtained from G_2 by removing v_n. We have $G \in \mathcal{L}^{n-1}(X)$. Therefore, there should be $G' \in H_{n-1}$ isomorphic to G. Let γ be an isomorphism from G to G' and $S' = S^\gamma$. If G_3 is the graph obtained from G' by adding a vertex, v' to S', then G_3 is isomorphic in G_2 and v' is winning in G_3. Hence, G_3 is accepted if it is generated. On the other hand, if K is the set of all subsets of vertices in G' that are equivalent to S' (K is the orbit of subsets in G' containing S'), then G_3 is obtained from G' by any extension that joins the new vertex to vertices of a subset in K and by assumption, G' is extended for exactly one subset in K. Thus, G_3 is generated and accepted. This means

$G_3 \in H_{n-1}$. Hence, we showed for any arbitrary $G_1 \in \mathcal{L}^n(X)$, there is an isomorphic graph, G_3 that is generated and accepted. □

Together, Lemma 3.4.3 and Lemma 3.4.4 give the following theorem:

Theorem 3.4.2. *When recursively applied, starting with the trivial graph, K_1, the algorithm described constructs exactly one representative of every isomorphism class of graphs in $\mathcal{L}_\mathcal{C}$.*

An alternative proof that shows our generation algorithm is exhaustive and isomorph-free is presented in Section 3.5.

3.4.3 Pseudo-Code

As we explained in Section 3.4, starting from the root, K_1 , and recursively generating all graphs in $\mathcal{G}_\mathcal{C}$ is not very efficient because this way many nodes are generated that are far from being extremal or having an extremal descendant. We used a technique called reverse engineering to avoid producing such nodes. In this method, we first recursively create a set of templates of graphs that are needed, starting from those with n vertices down to graphs with fewer vertices. We then recursively generate these graphs starting from smaller graphs up to graphs with n vertices.

Hence, the idea is, before applying any extension, we partition the graphs in $\mathcal{G}_\mathcal{C}$ into a set of classes based on the number of vertices, the number edges, the minimum degree, the number of vertices of minimum degree and the independence of the vertices of minimum degree. And then using a number of theorems and lemmas, we consider only a subset of these classes so that all extremal graphs with n vertices and all their ancestors are within these classes. We then proceed to generate all graphs in these classes. Although with this method, there are still some graphs generated that are not extremal and have no extremal descendant, the generation tree becomes dramatically smaller using the lemmas and theorems in Section 3.7 to rule out unnecessary classes.

In this section the set *FAM* presents a set of forbidden cycles. If $\mathcal{B} \in FAM$, then all odd cycles are forbidden. We define the set of 5-tuples U to be:

$$U = \{(n, e, d, m, t) |\ n > 0,\ e \geq 0,\ 0 \leq d < n,\ 0 < m \leq n \text{ and } t \in \{A, B\}\ \ \}.$$

A 5-tuple, $c \in U$, is called a *class*. For any class $c = (n_c,\ e_c,\ d_c,\ m_c,\ t_c) \in U$, we say a graph is in class c if it is of type t_c and has n_c vertices, e_c edges and m_c vertices of minimum degree d_c. The set of graphs in c is denoted by S_c and is defined as follows:

$$S_c = \{g \in \mathcal{G} |\ |V(g)| = n_c,\ |E(g)| = e_c,\ \delta(g) = d_c,\ \ |mins(g)| = m_c,\ \ \text{and } type(g) = t_c\ \}.$$

For a given number of vertices, n_0, and a number of edges, e_0, we define:

$$K[n_0,\ e_0] = \{(n_0, e_0, d, m, t) |\ (n_0, e_0, d, m, t) \in U\ \}.$$

$$M_{FAM}[n_0,\ e_0] = \{(n_0, e_0, d, m, t) |\ \exists g \in \mathcal{G}_{FAM} \cap S_{(n_0, ex_0, d, m, t)}\ \}.$$

We say a class, $p \in U$, is a parent of another class, $c \in U$, if there is a graph in S_c that can be generated in a genuine way from a graph in S_p. It is not always possible to determined the exact set of the parents of a class. However, in Section 3.7.2, we present a number of necessary conditions for a class to be a parent of a given class. For each class c, these conditions help to find a smaller superset of the parents of c. Each class in this superset is called a *feasible parent* of c.

Let $c, p \in U$ be two classes where p is a feasible parent of c. We define c_p to be a

subclass of c. The set of graphs in subclass c_p is denoted by $S_{c,p}$. It consists of graphs in S_c that can be generated from a graph in S_p.

In Section 3.7.3, we explain how introducing the concept of subclasses helped to increase the efficiency.

Given a number of vertices, n, and a set of cycles, FAM, we construct the set of classes, A, needed to be generated to have all graphs in $EX(n, FAM)$. To construct A, we start by inserting to A, the classes for extremal graphs with n vertices. We then recursively add to A, any class that can be a feasible parent to a class already in A.

Algorithm 6 takes an integer n and FAM that presents a set of forbidden cycles. It produces all graphs in $EX(n, FAM)$ and hence, the Turán number, $ex(n, FAM)$, can be obtained. In Section 3.7.1, we present a number of lemmas based on which given n, e and FAM, we can rule out a good portion of implausible classes from $K[n, e]$ and find a set S where $M_{FAM}[n, e] \subseteq S \subseteq K[n, e]$.

Algorithm 6 Generate_Extremal

```
1:  procedure GENERATE_EXTREMAL (int: n, Set: FAM)
2:      Let e = e_max where e_max is an upper bound for ex(n, FAM)
3:      while true do
4:          Find a set S where M_FAM[n, e] ⊆ S ⊆ K[n, e]
5:          Generate(S, FAM)                    ▷ or Parallel_Generate( S, FAM)
6:          if a graph with n vertices and e edges in G_FAM is generated then
7:              return e
8:          else  e = e − 1
9:          end if
10:     end while
11: end procedure
```

The procedure *Generate()* as depicted in Algorithm 7, takes a set of classes, $S \subset U$ and *FAM* that presents a set of forbidden cycles. For all $h \in S$, the algorithm generates all graphs in $S_h \cap \mathcal{S}_{FAM}$ by first generating their ancestors in the generation tree. The procedure *Make_Classes* as explained it the following, constructs a set of classes, A, where any ancestor of a graph in a class in S is either in a class in A or it is already available from previous runs. After calling *Make_Classes*, Algorithm 7 constructs a set of classes, *HOT*, that is a subset of the set A calculated by *Make_Classes*. The set *HOT* contains a number of classes in A where all the graphs belong to these classes are already available. The classes in *HOT* are, therefore, ready to have their graphs extended by procedure *Extend_Cycle_Free*. The set *HOT* is updated as the algorithm proceeds.

Algorithm 7 Generate

```
1:  procedure GENERATE(Set: S, Set: FAM)
2:
3:      A = Make_Classes(S)
4:      HOT = {a ∈ A | Missing_parents[a] == ∅}
5:      while HOT ≠ ∅ do
6:          h = extract(HOT)
7:          Out_Classes = {a ∈ A| h is a feasible parent of a}
8:          for each graph g ∈ S_h do
9:              Output = Extend_Cycle_Free(g, FAM, Out_Classes)
10:             for each g' ∈ Output do
11:                 h' = (|V(g')|, |E(g')|, δ(g'), |mins(g')|, type(g'))
12:                 S_h' = S_h' ∪ {g'}
13:             end for
14:         end for
15:         for each class a ∈ A do
16:             if s ∈ Missing_parents[a] then
17:                 Missing_parents(a)/{s}
18:                 if Missing_parents[a] == ∅ then HOT = HOT ∪ {a}
19:                 end if
20:             end if
21:         end for
22:     end while
23: end procedure
```

It is easy to see when the set *HOT* is empty, the graphs in classes constructed by *Make_Classes* are available. Therefore, all graphs in classes in the input set S are generated and the procedure terminates.

The procedure *Make_Classes()*, as presented in the Algorithm 8, takes a set of classes, S, and a set of forbidden cycles presented by *FAM*. It returns a set of classes, S', where for each graph g in a class in S, all the graphs on the path from g to the root, K_1, on the generation tree, are in a class in S' or are already available. In other words, this procedure gives all the classes whose corresponding graphs are needed to be generated. The set *existed* consists of subclasses whose corresponding graphs are all available. We can simply have $existed = \{x\}$ where x is the only subclass of the class $(1, 0, 0, 1, A)$ where K_1, is the only graph in x. Nonetheless, the efficiency increases, using the subclasses whose corresponding graphs are calculated and stored during the previous runs of the program. At the end of the algorithm, for each class, $c \in S'$, the set, $Missing\_parents[c]$, contains those parents of c that are not available and are yet to be generated. The function *Remove_Useless()* uses some lookahead techniques to rule out a portion of classes that can be disregarded from S'. It predicts some of the subclasses whose corresponding set of graphs will be empty. Also, based on a number of lemmas mentioned in Section 3.7.3, this function determines a number of subclasses including no graph that, on the generation tree, is a parent of a parent of a graph, say g_c, in a class in S'.

Algorithm 8 Make_Classes

```
1:  procedure MAKE_CLASSES(Set: S, Set FAM)
2:      T = S
3:      Let g be the trivial graph of one vertex and c = (1,0,0,1,A), k = (0,0,0,0,A)
4:      S_{c,k} = {g}
5:      existed = {c_k}          ▷ Or existed = {c_k| c,k ∈ U,  k is a parent of c and S_{c,k} is
    available }
6:      while T ≠ ∅ do
7:          c = extract(T)                                   ▷ c = (n_c, e_c, d_c, m_c, t_c)
8:          if c is not feasible according to FAM then S' = S'/{c}
9:          else
10:             Missing_parents[c] = ∅
11:             for each p ∈ U that is a feasible parent of c where c_p ∉ existed do
12:                 Missing_parents[c] = Missing_parents[c] ∪ p
13:                 if p not in S' then
14:                     S' = S' ∪ {p}
15:                     T = T ∪ {p}
16:                 end if
17:             end for
18:         end if
19:     end while
20:     Remove_Useless()
21:     return S'
22: end procedure
```

The procedure *Extend_Cycle_Free()* in Algorithm 9, takes a graph, g, a set of forbidden cycles presented by *FAM*, and a set of classes, *Out_Classes*, where output graphs should be in a class in *Out_Classes*. The algorithm outputs all graphs in a class in *Out_Classes* that are extended from g avoiding the given forbidden cycles. In this algorithm we consider a subset of vertices X of a graph to be *valid* according to a given set of cycles, *FAM*, if we can join all vertices in X to a new vertex without creating a forbidden cycle in *FAM*. The function *Weed_out (A, Out_Classes)* removes from A, a number of subsets of vertices where the graphs obtained by the extensions based on them are not in *Out_Classes*.

Algorithm 9 Extend_Cycle_Free

```
1: procedure EXTEND_CYCLE_FREE(Graph: g, Set: FAM, Set: Out_Classes)
2:     Output = ∅
3:     A = {X ⊆ V(g) | X is a valid subset according to FAM}
4:     Weed_out( A, Out_Classes)
5:     call nauty and calculate the orbits of subsets of vertices in A
6:     for each orbit do
7:         choose one subset X as the representative
8:         Let g' be the graph obtained from g by adding a vertex and joining it to
   vertices in X
9:         let n be the label of the vertex just added to g'          ▷ n − 1 = |V(g)|
10:        if Is_Genuine(g', n, X)  &  g' is in a class in Out_Classes then
11:            Output = Output ∪ {g'}
12:        end if
13:    end for
14:    Return Output
15: end procedure
```

The function *Is_Genuine()* as presented in Algorithm 10 takes a graph, g, an integer, n, where $|V(g)| = n$ and a subset, X, of vertices of g where $N(n) = X$. This means during the last step of the generation of g, the vertex labelled n is added and joined to the vertices in X. This function checks if g has generated in a genuine way from its parent by checking if the vertex n is the vertex of g that is supposed to be inserted last according to the rules we have defined for genuine reductions. These rules are mentioned in Section 3.4.2 and are implemented in Algorithm10.

Algorithm 10 The Is_Genuine algorithm

procedure Is_GENUINE(Graph: g, int: n, Set: X)
  $d = deg(n)$
  $set_1 = \emptyset$
  for each $v \in V(g)$ **do**
    if $deg(v) < d$ **then** return ***false***
    else if $deg(v) == d$ **then** $set_1 = set_1 \cup \{v\}$
    end if
  end for
  if $set_1 == \{n\}$ **then** return ***true***
  end if
  $d2 = |\{u|\ u \in set_1 \quad \& \quad \{n, u\} \in E(g)\}|$
  $set_2 = \emptyset$
  for each $v \in set_1$ **do**
    $k = |\{u|\ u \in set_1 \quad \& \quad \{v, u\} \in E(g)\}|$
    if $k > d2$ **then** return ***false***
    else if $k == d2$ **then** $set_2 = set_2 \cup \{v\}$
    end if
  end for
  if $set_2 == \{n\}$ **then** return ***true***
  end if
  $S^+ = \{u|\ deg(u) = d + 1\}|$
  $d3 = |\{u|\ u \in set_2 \quad \& \quad \exists w \in S^+ \ where\ \{n, w\}, \{w, u\} \in E(g)\}|$
  $set_3 = \emptyset$
  for each $v \in set_2$ **do**
    $k = |\{u|\ u \in set_2 \quad \& \quad \exists w \in S^+ \ where\ \{v, w\}, \{w, u\} \in E(g)\}|$
    if $k > d3$ **then** return ***false***
    else if $k == d3$ **then** $set_3 = set_3 \cup \{v\}$
    end if
  end for
  if $set_3 == \{n\}$ **then** return ***true***
  end if
  if n is in the same orbit as the vertex in set_3 with the largest canonical labelling calculated by *nauty* **then**
    return ***true***
  else
    return ***false***
  end if
end procedure

3.5 Formal Proofs

Avoiding a given set of cycles, $\mathcal{C}$, is a vertex-hereditary property of a graph, i.e, if graph X does not contain any cycle in $\mathcal{C}$, then neither is any induced subgraph of X and obviously this property is invariant under isomorphism. Therefore, according to section 4 of [146] our procedure works correctly Nonetheless, we present an explicit proof here. We prove that given a set of cycles $\mathcal{C}$, our algorithm generates all members of $\mathcal{G}_{\mathcal{C}}$ without producing repeated isomorphic copies. Our proofs are based on the theorems stated in [146] for a more general method of isomorph-free generation of families of combinatorial objects presented in . This method is discussed more extensively in other chapters.

Given a set of cycles C, the set $\mathcal{L}_{\mathcal{C}}$ is the set of all labelled graphs isomorphic to a graph in $\mathcal{G}_{\mathcal{C}}$, using labels $\{1, 2, .., n\}$ to label a graph with n vertices. For each $X \in \mathcal{L}_{\mathcal{C}}$, we define $o(X) = n$ if it has n vertices. Let G be the group of all relabellings of labelled graphs in $L_{\mathcal{C}}$, so that the graphs in one orbit are all isomorphic. Hence, $G = S_1 \times S_2 \times ...$, where the action on $\mathcal{L}_{\mathcal{C}}$ is such that the factor S_n is the symmetric group of degree n permuting the labels on graphs of order n in $\mathcal{L}_{\mathcal{C}}$. Let X be an arbitrary graph in $\mathcal{L}_{\mathcal{C}}$. We define the following:

- For $S \subseteq V(X)$, we denote by X_S, the graph constructed from X by inserting a new vertex to and joining it to the vertices in S.

- $L(X)$ is the set of lower objects of X where:

$$L(X) = \begin{cases} \{\langle X, v\rangle \mid v \in V(X)\}, & \textit{where } o(X) > 1, \\ \emptyset, & \textit{Otherwise.} \end{cases}$$

  Therefore, $L(K_1) = \emptyset$. In fact the isomorphism class of the trivial graph K_1 is the only irreducible unlabelled graph in $G_{\mathcal{C}}$.

- $U(X)$ is the set of upper objects of X where:

$$U(X) = \{\langle X, S\rangle \mid S \subseteq V(X) \textit{ where } X_S \in \mathcal{L}_{\mathcal{C}}\}.$$

- $\check{\mathcal{L}} = \cup_{X \in \mathcal{L}_r} L(X)$ and $\hat{\mathcal{L}} = \cup_{X \in \mathcal{L}_r} U(X)$.

- For each $\check{Y} = \langle Y, v\rangle \in \check{\mathcal{L}}$:

$$f(\check{Y}) = \{\hat{X}^g = \langle X, S\rangle^g \mid g \in G,\ X = Y - v \textit{ and } S = N(v) \textit{ in } Y\}$$

  Clearly $\hat{X} = \langle X, S\rangle \in \hat{\mathcal{L}}$ and $Y = X_S$.

- For each $\hat{X} = \langle X, S\rangle \in \hat{\mathcal{L}}$:

$$f'(\hat{X}) = \{\check{Y}^g = \langle Y, v\rangle^g \mid g \in G,\ \ Y = X_S \textit{ and } v \in V(Y) \textit{ where } N(v) = S\}$$

  It is obvious that $\check{Y} = \langle Y, v\rangle \in \check{\mathcal{L}}$.

- According to our definition of the genuine reduction, the function $m : \mathcal{L}_{\mathcal{C}} \to 2^{\check{\mathcal{L}}}$ mentioned in [146] is defined as follows :

$$m(X) = \begin{cases} \{(X, v) |\ \ v \textit{ is a winning vertex of } X\} & \textit{if } o(X) > 1 \\ \emptyset & \textit{Otherwise} \end{cases}$$

  As a reminder from Section 3.4.2, a vertex v in $V(X)$ is winning if there is no vertex v', in $V(X)$, whose 4-tuple, $x(v')$, is lexicographically larger than the 4-tuple $x(v)$ where $x(k) = (x_0(k), x_1(k), x_2(k), x_3(k))$ for each vertex k in $V(X)$.

Theorem 3.5.1. *Our definitions satisfy the constraints of axioms C1 − C7 stated in [146].*

Proof. The group G is the group of all relabellings of labelled graphs in $\mathcal{L}_{\mathcal{C}}$, so that the graphs in one orbit are all isomorphic. This group acts on lower and upper objects as follows: For each $g \in G$, $\langle X, v\rangle \in \check{\mathcal{L}}$ and $\langle X, S\rangle \in \hat{\mathcal{L}}$, we have $\langle X, v\rangle^g = \langle X^g, v^g\rangle$ and $\langle X, S\rangle^g = \langle X^g, S^g\rangle$. Also, it is easy to see that an image, S^g of a subset of vertices $S \subseteq V(X)$ under g is also a subset of vertices in X^g and the property of being a valid subset (in terms of not creating a forbidden cycle when joined to a new vertex) is invariant under isomorphisms. This means $X_S \in \mathcal{L}_{\mathcal{C}}$ if and only if $X^g_{S^g} \in \mathcal{L}_{\mathcal{C}}$.

C1. We show G fixes each of $\mathcal{L}_{\mathcal{C}}$, $\check{\mathcal{L}}$ and $\hat{\mathcal{L}}$ set-wise.

1. For any $X \in \mathcal{L}_{\mathcal{C}}$ and $g \in G$, the graph X^g is isomorphic to X and hence, it is also in $\mathcal{L}_{\mathcal{C}}$.

2. For any $\langle X, v\rangle \in \check{L}$ and $g \in G$ we have $\langle X, v\rangle^g = \langle X^g, v^g\rangle$ where X^g is isomorphic to X and in $\mathcal{L}_{\mathcal{C}}$. It is not hard to see v^g is a vertex of X^g and hence, by our definitions $\langle X^g, v^g\rangle \in L(X^g)$. Therefore, $\langle X, v\rangle^g = \langle X^g, v^g\rangle \in \check{L}$.

3. For any $\langle X, S\rangle \in \hat{L}$ and $g \in G$, we have $\langle X, S\rangle^g = \langle X^g, S^g\rangle$ where X^g is isomorphic to X and hence, is in $\mathcal{L}_{\mathcal{C}}$ and as we mentioned above S^g is a subset of vertices of X^g, and $X^g_{S^g} \in \mathcal{L}_{\mathcal{C}}$. Therefore, $\langle X^g, S^g\rangle \in U(X^g)$ and consequently $\langle X, S\rangle^g = \langle X^g, S^g\rangle \in \hat{L}$.

C2. We show for each $X \in \mathcal{L}_{\mathcal{C}}$, we have: $L(X^g) = L(X)^g$ and $U(X^g) = U(X)^g$:

1. $L(X^g) = L(X)^g$:

$$L(X) = \{\langle X, v\rangle |\ v \in V(X)\}$$
$$\implies L(X)^g = \{\langle X, v\rangle^g |\ v \in V(X)\}$$

$$= \{\langle X^g, v^g, \rangle |\ v^g \in V(X^g)\} = L(X^g).$$

2. $U(X^g) = U(X)^g$:

$$U(X) = \{\langle X, S\rangle |\ S \subseteq V(X) \text{ where } X_S \in \mathcal{L}_\mathcal{C}\}$$

It is obvious for each subset $S \subseteq V(X)$, we have $X_S \in \mathcal{L}_\mathcal{C}$ if and only if $X^g_{Wg} \in \mathcal{L}_\mathcal{C}$. Therefore:

$$\Longrightarrow U(X)^g = \{\langle X, S\rangle^g | S \subseteq V(X) \text{ where } X_S \in \mathcal{L}_\mathcal{C}\}$$
$$= \{\langle X^g, S^g\rangle |\ S^g \subseteq V(X^g) \text{ where } X^g_{Sg} \in \mathcal{L}_\mathcal{C}\} = U(X^g).$$

C3. $\forall \check{Y} \in \check{\mathcal{L}}, f(\check{Y}) \neq \emptyset$:
For each $\check{Y} \in \check{\mathcal{L}}$, we have $\check{Y} = \langle Y, v\rangle$ where $Y \in \mathcal{L}_\mathcal{C}$, $o(Y) > 1$ and $v \in V(Y)$. Let $S = N(v)$ in Y, and X be the graph obtained from Y by removing v. It is easy to see $\langle X, S\rangle \in f(\check{Y})$. Hence, $f(\check{Y}) \neq \emptyset$.

C4: We show for any $\check{Y} \in \check{\mathcal{L}}$, $g \in G$, $\hat{X}_1 \in f(\check{Y})$ and $\hat{X}_2 \in f(\check{Y}^g)$, there exists $h \in G$ such that $\hat{X}_1^h = \hat{X}_2$.

Suppose $\hat{X}_1 = \langle X_1, S_1\rangle$, $\hat{X}_2 = \langle X_2, S_2\rangle$ and $\check{Y} = \langle Y, v\rangle$. Hence, $\check{Y}^g = \langle Y^g, v^g\rangle$. Since $\hat{X}_1 \in f(\check{Y})$, there is $h_1 \in G$ satisfying $X_1 = A_1^{h_1}$ and $S_1 = B_1^{h_1}$ where A_1 is obtained from Y by removing v and $B_1 = N(V)$ in Y. Also, we have $\hat{X}_2 \in f(\check{Y}^g)$, so there is $h_2 \in G$ satisfying $X_2 = A_2^{h_2}$ and $S_2 = B_2^{h_2}$ where A_2 is obtained from Y^g by removing the vertices in v^g and B_2 is the subset representing the of neighbours of the vertices of v^g in Y^g.
Consider $h = h_2(g(h_1^{-1}))$, Since $h_1, h_2, g \in G$, we have $h \in G$. We prove $\hat{X}_1^h = \hat{X}_2$:

1. $X_1^h = X_2$: We have $X_1^{h_1^{-1}} = A_1$ and $A_2^{h_2} = X_2$. It is easy to verify $A_1^g = A_2$. Hence, $X_1^h = X_1^{h_2(g(h_1^{-1}))} = (X_1^{h_1^{-1}})^{h_2 g} = (A_1^g)^{h_2} = A_2^{h_2} = X_2$.

2. $S_1^h = S_2$: We have $S_1^{h_1^{-1}} = B_1$ and $B_2^{h_2} = S_2$. It is easy to verify $B_1^g = B_2$. So similar to above, one can verify $S_1^h = S_2$.

C5: We show for any $\hat{X} \in \hat{\mathcal{L}}$, $g \in G$, $\check{Y}_1 \in f'(\hat{X})$ and $\check{Y}_2 \in f'(\hat{X}^g)$, there exists $h \in G$ such that $\check{Y}_1^h = \check{Y}_2$.

Suppose $\check{Y}_1 = \langle Y_1, v_1\rangle$, $\check{Y}_2 = \langle Y_2, v_2\rangle$ and $\hat{X} = \langle X, S\rangle$. Hence, $\hat{X}^g = \langle X^g, S^g\rangle$. We have $\check{Y}_1 \in f'(\hat{X})$, so there is $h_1 \in G$ satisfying $Y_1 = A_1^{h_1}$, $v_1 = x_1^{h_1}$ where $A_1 = X_S$ and x_1 is the vertex where $N(x_1) = S$ in A_1. Also, we have $\check{Y}_2 \in f'(\hat{X}^g)$, so there is $h_2 \in G$ satisfying $Y_2 = A_2^{h_2}$, $v_2 = x_2^{h_2}$ where $A_2 = X^g_{Sg}$ and x_2 is the vertices where $N(x_2) = S^g$ in A_2.
Consider $h = h_2(g(h_1^{-1}))$, Since $h_1, h_2, g \in G$, we have $h \in G$. We prove $\check{Y}_1^h = \check{Y}_2$:

1. $Y_1^h = Y_2$: We have $Y_1^{h_1^{-1}} = A_1$ and $A_2^{h_2} = Y_2$. It is easy to verify $A_1^g = A_2$. So similar to **C4**, one can easily show $Y_1^h = Y_2$.

2. $d_1^h = v_2$: We have $v_1^{h_1^{-1}} = x_1$ and $x_2^{h_2} = v_2$. It is obvious that $x_1^g = x_2$ and hence, $v_1^h = v_2$.

C6: We show $o(X^g) = o(X)$:

For each $X \in \mathcal{L}_\mathcal{C}$, we defined $o(X)$ to be the number of vertices of X, which is an invariant under isomorphisms. This means $o(X^g) = o(X)$.

C7: We show for each $\check{Y} \in \check{\mathcal{L}}$ and $\hat{X} \in f(\check{Y})$ we have $o(\hat{X}) < o(\check{Y})$:

According to our definitions for each $\check{Y} \in \check{\mathcal{L}}$ and $\hat{X} \in f(\check{Y})$ we have $o(\hat{X}) = o(\check{Y}) - 1 < o(\check{Y})$.

□

Lemma 3.5.1. *The function $m(X)$ defined above is well-defined and satisfies the requirements $M1 - M3$ stated in [146].*

Proof. It is obvious that $m(X)$ is well defined. We prove each requirement $M1$, $M2$ and $M3$ holds:

M1. If $L(X) = \emptyset$, then $m(X) = \emptyset$:

According to the definitions we presented for $m(X)$ and $L(X)$, we have $m(X) \subseteq L(X)$ and hence, if $L(X) = \emptyset$, then $m(X) = \emptyset$.

M2. If $L(X) \neq \emptyset$, then $m(X)$ is an orbit of the action of $Aut(X)$ on $L(X)$:

Consider a graph X with $L(X) \neq \emptyset$. The orbits of the action of the automorphism group of the vertices, $Aut(X)$, on $L(X)$, are the same as orbits of the action of $Aut(X)$ on the vertices of X and the values of x_0, x_1 and x_2 are vertex invariants. Therefore, all vertices in a same orbit of the action of $Aut(X)$ on $L(X)$ have same values for them. It is also clear that the value of x_3 for all vertices in one orbit is the same. So if for a vertex v, we have $\langle X, v\rangle \in m(X)$ then for every vertex, u, in the same orbit as v, we have $\langle X, u\rangle \in m(X)$.
Furthermore, the value of x_3 is different for vertices in different orbits, therefore, $m(X)$ is an orbit of action of $Aut(X)$ on $L(X)$.

M3. For each $X \in \mathcal{L}_\mathcal{C}$ and $g \in G$ we have $m(X^g) = m(X)^g$:

Let $W(X) = \{v | \langle X, v\rangle \in m(X)\}$. We only need to show $W(X^g) = W(X)^g$. We proceed the proof in two steps:

1. For each $u \in W(X)^g$, we have $u \in W(X^g)$:

   If $u \in W(X)^g$ then $v = u^{g^{-1}} \in W(X)$ and hence, v is a vertex with lexicographically largest 4-tuple for $x = (x_0, x_1, x_2, x_3)$ in X. Since the values of x_0, x_1, x_2 and x_3 are vertex invariants, then the 4-tuple $x(v^g)$ must also be lexicographically largest among all the 4-tuples of all vertices in X^g. Therefore, $u = v^g \in W(X^g)$.

2. For each $u \in W(X^g)$, we have $u \in W(X)^g$:

If $u \in W(X^g)$, then u is a vertex with lexicographically largest 4-tuple for $x = (x_0, x_1, x_2, x_3)$ in X^g. Since the values of x_0, x_1, x_2 and x_3 are vertex invariants, then the 4-tuple x of $v = u^{g^{-1}}$ must also be lexicographically largest among all the 4-tuples of all vertices in X. Therefore, $v \in W(X)$ and hence, $u = v^g \in W(X)^g$.

□

3.6 Parallelisation

In the method of *GCCP*, each graph can be extended independently from the other graphs in the search tree. This enables parallel implementation of the generation process. Parallelisation dramatically accelerated our generation process in search for extremal graphs where we have partitioned the graphs into classes in $U = \{(n, e, d, m, t) | \ n > 0,\ e \geq 0,\ 0 \leq d < n,\ 0 < m \leq n \text{ and } t \in \{A, B\}\ \}$. The extensions of graphs in each class are independent of other classes. However, we extend a graph in each class, only when all graphs in that class are available. In our algorithms, such classes are stored in a set called *HOT*. The graphs in this set can be just generated in the current run or have been retrieved from the stored results of the previous runs.

As it is depicted in Algorithm 11, a set of processes are run simultaneously. The graphs in each class may be extended by different processes. In fact, we partition the set of possible extensions of the graphs in each class into several parts based on the obtained graph, as explained in Algorithm 12. And each set of extensions is proceeded by one process. Therefore, a process is responsible for conducting a set of extensions on a set of graphs. Each process applies these extensions using several threads simultaneously where the set of graphs assigned to a process for extensions is further divided into several sections and the graphs in each section are extended by a separate thread.

The procedure *Parallel_Generate()* depicted in Algorithm 11 is a parallel version of the procedure *Generate()* presented in Algorithm 7. Similar to that procedure, *Parallel_Generate()* takes a set of classes, $S \subset U$ and a set of forbidden cycles, *FAM*. For all $h \in S$, the algorithm generates all graphs in $S_h \cap \mathcal{S}_{FAM}$ by first generating all of their ancestors in the generation tree. The extensions during the generation process are conducted simultaneously by a number of processes and threads. The set of extensions is partitioned into several parts by calling function *Make_Parallel_Sections()* so that the graphs obtained from the extensions in one part share a number of invariants. This function is implemented in Algorithm 12. Each part is conducted by one separate process. After a process finishes applying the extensions in one part and adding the newly generated graphs to their corresponding sets or files, Algorithm 11 updates the information about available classes and subclasses.

Algorithm 11 Parallel_Generate

```
1: procedure PARALLEL_GENERATE(Set: S, , Set: FAM)
2:     A = Make_Classes(S)
3:     HOT = {a ∈ A | Missing_parents(a) == ∅}
4:     Process_Idle = Process_Set
5:     while true do
6:         while HOT ≠ ∅ do
7:             s = extract(HOT)
8:             section = Make_Parallel_Sections(s, A, FAM)
9:             for k = 1, 2, ..., section.num do
10:                if Process_Idle == ∅ then
11:                    Wait until a process in Process_Set/Process_idle is idle
12:                    Choose an idle process q ∈ Process_Set/Process_idle
13:                    Process_idle = Process_idle ∪ {q}
14:                    Let j be such that id[q] = j
15:                    for each class a ∈ proc[j].child do
16:                        Missing_parents[a] = Missing_parents[a]/{proc[j].class}
17:                        if Missing_parents[a] == ∅ then HOT = HOT ∪ {a}
18:                        end if
19:                    end for
20:                end if
21:                Let p be an idle process
22:                Make p run Parallel_Extend(s, section.parent[i], section.child[i], FAM)
23:                i = extract(Process_Idle)
24:                poc[i].id = Id_Process(p)
25:                proc[i].class = s
26:                proc[i].child = section.child[k]
27:                proc[i].parent = section.parent[k]
28:            end for
29:        end while
30:        if Process_Idle == Process_Set then return
31:        else
32:            Wait until a process in Process_Set/Process_idle is idle
33:            Choose an idle process q ∈ Process_Set/Process_idle
34:            Process_idle = Process_idle ∪ {q}
35:            Let j be such that id[q] = j
36:            for each class a ∈ proc[j].child do
37:                Missing_parents[a] = Missing_parents[a]/{proc[j].class}
38:                if Missing_parents[a] == ∅ then HOT = HOT ∪ {a}
39:                end if
40:            end for
41:        end if
42:    end while
43: end procedure
```

The procedure *Make_Parallel_Sections()* depicted in Algorithm 12 takes a specific set of classes, s, the set of classes, $A \subset U$, whose graphs are going to be generated, and a set of forbidden cycles, *FAM*. The algorithm partitions the extension of graphs in class s. It first considers the set C of all classes in A where s is a feasible parent of them. Then C is divided into several parts where the graphs in classes in one part are of the same type and have the same number of vertices, number of edges and minimum degree, and the values of the number of vertices of minimum degree of these graphs are contiguous. For each part of C, say $ch[i]$, we consider the set $par[i]$ that contains all subclasses of s that can be feasible parents to a class in $chi[i]$. The function *Is_gpc()* checks this feasibility based on lemmas in Section 3.7.3. Now the set of extensions of graphs in s is partitioned according to $ch[i]$s, so that, for each i, all the extensions that extend a graph in $par[i]$ to a graph $chi[i]$ are in one part. Each part is then managed by a separate process in Algorithm 11.

Algorithm 12 Make_Parallel_Sections

```
1: procedure MAKE_PARALLEL_SECTIONS(Class: s, Set: A, Set: FAM)
2:     C = ∅
3:     for each a ∈ A where s ∈ Missing_parents[a] do   C = C ∪ {a}
4:     end for
5:     Partition C into parts children = {ch[1], ch[2], ... ch[k]} for some k, where
       graphs in classes in one part, say ch[i], are of the same type and have the same
       number of vertices, number of edges and minimum degree. Furthermore, there
       are two integers m_1 and m_2 where {m_x | x ∈ ch[i]} = {m_1, m_1 + 1, m_1 + 2, ... , m_2}
       where m_x is the number vertices of minimum degree in graphs in class x.
6:     for each i = {1, 2, ..., k} do
7:         par[i] = ∅
8:         for each a ∈ A that is a feasible parent of s do
9:             if There is x ∈ ch[i] where Is_gpc(a, s, x) then
10:                par[i] = par[i] ∪ {a}
11:            end if
12:        end for
13:    end for
14:    for each i = {1, 2, ..., k} do
15:        section.child[i] = ch[i]
16:        section.parent[i] = par[i]
17:    end for
18:    section.num = k
19:    return section
20: end procedure
```

Note that a subclass of s can be a feasible parent to several subclasses that are in

different parts of C. This means the graphs in one subclass of s can be considered by several processors for extensions. Although this can waste some time to perform repeated computations, the overall efficiency is increased since the graphs obtained from extensions in one part share a number of invariants and are distributed fewer subclasses. This way, each process need to manage reading and writing on fewer files.

The procedure *Parallel_Extend()* as shown in Algorithm 13 is run by a distinct process to conduct a set of extensions that are in one part of the partitioned obtained in Algorithm 12. These extensions are determined in the input by a specific set of classes, a, a set of its subclasses, *parents*, and a set of classes for output graphs, *children*. The algorithm also takes a set of forbidden cycles, *FAM* and it activated a number of threads to distribute the burden of computations needed for extensions and reading and writing to and from files on a secondary memory. To to avoid collisions in reading the input graph and writing the output graphs from and to files, one single thread manages writing and one single thread manages reading tasks. On the other hand, extending the input graphs are independent. Therefore, we use multiple threads for the extensions where input graphs are partitioned and each tread is responsible for extending graphs in one part.

Algorithm 13 Parallel_Extend

1: **procedure** PARALLEL_EXTEND(Class: a, Set: *parents*, Set: *children* , Set: *FAM*)
2: Activate the thread *writer* that calls *Parallel_Writer*(a)
3: Activate maximum possible number of extender threads that call *Parallel_Extender*$(a,$ *children*, *FAM*$)$
4: Activate the thread *reader* that calls *Parallel_Reader*$(a,$ *parents*, *FAM*$)$
5: **end procedure**

The procedure *Parallel_Reader()* as shown in Algorithm 14 is run by a distinct thread. It takes a specific set of classes, s, a set of its subclasses, *parents* and a set of forbidden cycles, *FAM*. It considers a set of files that are corresponding to subclass of a that are in *parents*. It reads to its buffer, one by one, all graphs in such files and whenever its buffer gets full, it sends the graphs on its buffer to an idle extender thread waiting for input graphs.

Algorithm 14 Parallel_Reader

```
1: procedure PARALLEL_READER(Class: a, Set: parents, Set: FAM)
2:     G = ∅, input = ∅
3:     for each class p ∈ parents do
4:         input = input ∪ a_p
5:     end for
6:     while There is a graph, g, to be read that is in a file corresponding to a subclass
       in input do
7:         G = G ∪ g
8:         if |G| == Max_G or g is the last input graph then
9:             while No extender thread is idle do
10:                                        ▷ wait for an extender thread to become idle
11:            end while
12:            Choose one idle extender thread, t
13:            Send G to t G = ∅
14:        end if
15:    end while
16: end procedure
```

The procedure *Parallel_writer()* as shown in Algorithm 15 is run by a distinct thread. It takes a specific set of classes, a as input and waits to receive a set of output from an extender thread. Whenever an extender thread sends a set of output, Algorithm 15 writes each output graph on a file corresponding to the proper subclass.

Algorithm 15 Parallel_Writer

```
1: procedure PARALLEL_WRITER(Class: a)
2:     while There is an active extender thread do
3:         Wait until an extender thread offers input
4:         Choose one extender thread, t, that offers input
5:         Let S be the set of input given by t
6:         while S ≠ ∅ do
7:             k = extract(S)
8:             Write graph k.graph on the file corresponding to the subclass k.subclass
9:         end while
10:    end while
11: end procedure
```

The procedure *Parallel_Extender()* depicted in Algorithm 16 are called in each extender thread. This procedure takes a specific set of classes, *a* a set of classes, *children*, for output graphs, and as well as the set of forbidden cycles, *FAM*. It receives the input graphs from the thread *reader* and extend them to graphs that are in classes in *children* while storing the obtained graphs to its buffer. When this buffer is full, the procedure waits for the thread *writer* to be idle to transfer the output graphs in the buffer.

Algorithm 16 Parallel_ Extender

```
1: procedure PARALLEL_EXTENDER(Class: a, Set: children, Set: FAM)
2:     S = ∅
3:     while The thread reader is active do
4:         Announce this thread is idle
5:         Wait until this thread is chosen by the thread reader or the thread reader is
   inactive
6:         if This thread is chosen by the thread reader then
7:             Announce this thread is busy
8:             Let G_in be the set of graphs received from the thread reader
9:             for each g ∈ G_in do
10:                A = Extend_Cycle_Free(g, FAM, children)
11:                for each h ∈ A do
12:                    Let c ∈ children be the class where h ∈ S_c
13:                    k.graph = h,   k.subclass = c_a
14:                    S = S ∪ {k}
15:                    if |S| == MAX_S or this is the last graph to be generated by this
   thread then
16:                        Wait until this thread is chosen by the thread writer
17:                        Give S to the thread writer
18:                        S = ∅
19:                    end if
20:                end for
21:            end for
22:        end if
23:    end while
24: end procedure
```

3.7 Efficiency and Pruning the Generation Tree

In general, the method of *GCCP* is a time and storage efficient method to generate combinatorial objects. It provides lookaheads that increase the efficiency of the generation process. Parents and children have a very similar structure and many common computations can be avoided for children, using the results that are calculated and stored for parents. Using *nauty* to calculate canonical labellings and automorphisms of graphs, and considering a proper data structure to store graphs, are other factors of the high efficiency of our programs. More on such data structures and *nauty* can be found in [147].

In this chapter, introducing the concept of subclasses helps with further increasing the efficiency. Extending one subclass p, may produce graphs in multiple classes, $a, b, \ldots, h$. Therefore, to produce graphs in subclasses, $a_p, b_p, \ldots, h_p$, we only need to produce graphs in S_p once, store them on the main memory or on a secondary storage, and retrieve them next time a subclass whose parent is p, is needed. Also, as we explain in Section 3.7.3, with the concept of subclasses, we can prune the generation tree by reducing the number of classes and subclasses whose graphs are required to be generated. Also, the set of graphs of a subclass, a_b, is empty when the set of graphs of a or b is empty, or when b is not a parent of a. We provide a number of theorems in Sections 3.7.1 and 3.7.2 that can help to determine, based on the specifications of classes, whether the set of graphs of a class or subclass is empty. On the other hand, for each subclass, c_p, if the sets of graphs of all subclasses of p, i.e., $\{p_{h_1}, p_{h_2}, \ldots, p_{h_k}\}$, are known to be empty, then we can conclude S_{c_p} remains empty and we can avoid the calculations towards generating the graphs of c_p. This information can be also recursively used to determine other subclasses whose set of graphs are empty. The procedure *Remove_Useless()* uses these facts and a number of lemmas mentioned in Section 3.7.3 to determine the classes that can be disregarded.

Note that, all these theorems and calculations to prune the generation tree are actually computed before extending any graph. These pre-processings leave us with a much smaller generation tree and are crucial for increasing the efficiency of the generation process.

Later, in Section 3.8, we provide some theorems based on which we further increase the efficiency by reducing the number of calls to *nauty* even when group information is required to determine whether to accept or disregard a generated graph.

3.7.1 Determining the Empty Classes

In this section, we provide a number of theorems based on which we can rule of classes with no graph belongs to them.

Theorem 3.7.1. *Let $\mathcal{C}$ be a set of cycles and $n \geq 2$. For any graph $X \in \mathcal{G}_{\mathcal{C}}$ with n vertices we have: $|E(X)| - \delta(X) \leq ex(n-1, \mathcal{C})$.*

Proof. Let $X \in \mathcal{G}_{\mathcal{C}}$ be a graph with n vertices where $|E(X)| - \delta(X) > ex(n-1, \mathcal{C})$. By removing a vertex of minimum degree from X, we obtain a graph in $\mathcal{G}_{\mathcal{C}}$ with $n-1$ vertices and e edges where $e = |E(X)| - \delta(X) > ex(n-1, \mathcal{C})$ which is a contradiction. □

Then Theorem 3.7.1 says for a class $c' = (n_{c'}, e_{c'}, d_{c'}, m_{c'}, t_{c'}) \in U$, if $e_{c'} - d_{c'} > ex(n_{c'} - 1, \mathcal{C})$, then $\mathcal{G}_{\mathcal{C}} \cap \mathcal{S}'_c \emptyset$.

Theorem 3.7.2. *Let $\mathcal{C}$ be a set of cycles and $n \geq 3$. The following holds:*

1. *If $e - 2d + 1 > ex(n-2, \mathcal{C})$, then $\mathcal{G}_{\mathcal{C}} \cap S_{(n,e,d,m,B)} = \emptyset$.*
2. *If $e - 2d > ex(n-2, \mathcal{C})$ and $m \geq 2$, then $\mathcal{G}_{\mathcal{C}} \cap S_{(n,e,d,m,A)} = \emptyset$.*

Proof. Let $X \in \mathcal{G}_{\mathcal{C}} \cap S_{(n,e,d,m,B)}$, then by removing two adjacent vertices of degree d, we have a graph in $\mathcal{G}_{\mathcal{C}}$ with $n-2$ vertices and $e - 2d + 1$ edges. Therefore, we must have $e - 2d + 1 \leq ex(n-2, \mathcal{C})$.
Let $X \in \mathcal{G}_{\mathcal{C}} \cap S_{(n,e,d,m,A)}$, then by removing two vertices of degree d, we have a graph in $\mathcal{G}_{\mathcal{C}}$ with $n-2$ vertices and $e - 2d$ edges. Therefore, we must have $e - 2d > ex(n-2, \mathcal{C})$. □

Theorem 3.7.3. *For given n, e, d, m, the set $S_{(n,e,d,m,A)}$ is empty if one of the following holds:*

1. *There is no bipartite graph with m vertices in one part and $n - m$ vertices in another part, and md edges where at least one vertex in the first part is of degree d.*
2. *$n - m < d$.*

Proof. The proofs are trivial. □

Theorem 3.7.4. *Let $C_3 \in \mathcal{C}$ and for a given graph X, $K(X) = \{\{x, y\} | \quad x, y \notin mins(X)$ & $N(x) \cap N(y) \cap mins(X) = \emptyset\}$. We have $\mathcal{G}_{\mathcal{C}} \cap S_{(n,e,d,m,A)} = \emptyset$ if $e - md > |K(X)|$ for any graph, $X \in S_{(n,e,d,m,A)}$.*

Proof. We show that for any graph $X \in \mathcal{G}_{\mathcal{C}} \cap S_{(n,e,d,m,A)}$, we have $e - md \leq |K(X)|$. Consider a graph $X \in S_{(n,e,d,m,A)}$ where $e - md > |K(X)|$. Since $mins(G)$ is independent, $e - md$ is the number of edges with both ends in $V(X) \setminus mins(X)$. Therefore, if $e - md > |K(X)|$, there is at least one edge between two vertices in $V(G) \setminus mins(G)$ that share a vertex of degree d and hence, G contains a C_3. □

Theorem 3.7.5. *We have $\mathcal{G}_{\mathcal{C},\mathcal{B}} \cap S_{(n,e,d,m,A)} = \emptyset$ if $e > md$ and in every bipartite $(m, n-m)$-graph with md edges where there is at least one vertex of degree d in the first part, we have all the vertices of the second part to be in one component.*

Proof. The proof is trivial. □

Theorem 3.7.6. *We have $S_{(n,e,d,m,A)} \cup S_{(n,e,d,m,B)} = \emptyset$ if one of the following holds:*

1. $m < n(d+1) - 2e$.
2. $d = n-1$ *and* $m \neq n$
3. $d < n-1$ *and* $m > (n(n-1) - 2e)/(n-1-d)$.

Proof. If there is a graph $X \in S_{(n,e,d,m,A)} \cup S_{(n,e,d,m,B)}$ ($|V(X)| = n$, $|E(X)| = e$, $\delta(X) = d$ and $|mins(X)| = m$), then:

1. It is easy to see $2e \geq dm + (n-m)(d+1) = n(d+1) - m \Rightarrow m \geq n(d+1) - 2e$.
2. If $d = n-1$ then $m = n$.
3. For $d < n-1$, let X' be the complement graph of X. If X' has e' edges and k vertices of maximum degree D, then $kD \leq 2e'$. On the other hand we have $e' = \binom{n}{2} - e$, $D = (n-1) - d$ and $k = m$. So we have $m(n-1-d) \leq n(n-1) - 2e$ and thus, $m \leq (n(n-1) - 2e)/(n-1-d)$

□

The first part of Theorem 3.7.6 does not provide a tight bound when d is smaller and e is closer to $\binom{n}{2}$ but not to $nd/2$.

Theorem 3.7.7. *Let $\mathcal{C}$ be a set of cycles. For any number of vertices $n > 2$, we have $S_{(n,e,d,m,A)} \cup S_{(n,e,d,m,B)} = \emptyset$ if $e > \frac{n.\, ex(n-1,\mathcal{C})}{n-2}$.*

Proof. If there is a graph $X \in \mathcal{G}_{\mathcal{C}} \cap (S_{(n,e,d,m,A)} \cup S_{(n,e,d,m,B))}$ ($|V(X)| = n$, $|E(X)| = e$, $\delta(X) = d$ and $|mins(X)| = m$), then by Theorem 3.7.1, we have $e - d \leq ex(n-1,\mathcal{C})$. Therefore:

$$d \geq e - ex(n-1,\mathcal{C}) \Rightarrow nd/2 \geq n(e - ex(n-1,\mathcal{C}))/2.$$

On the other hand, we have $e \geq nd/2$. Hence:

$$e \geq n(e - ex(n-1,\mathcal{C}))/2 \Rightarrow (n-2)e/2 \leq n.\, ex(n-1,\mathcal{C})/2$$

Therefore, $e \leq n.ex(n-1,\mathcal{C})/(n-2)$ □

Theorem 3.7.7 says $ex(n,\mathcal{C}) \leq \frac{n.\, ex(n-1,\mathcal{C})}{n-2}$. The following theorem introduces an upper bound on number of edges in a bipartite graphs with no C_4.

Theorem 3.7.8. *For any graph in $\mathcal{G}_{\{C_4\} \cup \mathcal{B}}$ with $(n_1 + n_2)$ vertices and e edges and for all $t \leq n_1$, we have:*

$$e \leq n_2 + \binom{t}{2} + (n_1 - t)\frac{n_2 + \binom{t}{2}}{t}.$$

Proof. Let X be a bipartite graph without C_4, with $(n_1 + n_2)$ vertices and e edges where $V(X) = V_1 \cup V_2$, $|V_1| = n_1$ and $|V_2| = n_2$. For $t \leq n_1$, consider a t-subset $H = \{u_1, u_2, \ldots, u_t\} \subset V_1$. Let $A_i \subset V_2$ be the set of the neighbours of vertex u_i, for each $i = 1, \ldots, t$. By inclusion-exclusion principle have:

$$n_2 \geq |A_1 \cup A_2 \cup \ldots \cup A_t| \geq \sum_{i=1,\ldots,t} |A_i| - \sum_{i<j<t} |A_i \cap A_j|.$$

On the other hand, for all $i, j \in \{1, \ldots, t\}$ where $i \neq j$, we have $|A_i \cap A_j| \leq 1$ otherwise G contains a C_4. Therefore,

$$n_2 \geq \sum_{i=1,\ldots,t} |A_i| - \binom{t}{2} = \sum_{i=1,\ldots,t} deg(u_i) - \binom{t}{2}.$$

Hence, for any $t \leq n_1$, we have $n_2 + \binom{t}{2} \geq \sum_{i=1,\ldots,t} deg(u_i)$.

This inequality holds for every t-subset including the subset with t largest degree vertices. Let $V1 = \{v_1, v_2, \ldots, v_{n_1}\}$ where $deg(v_1) \geq deg(v_2) \geq \ldots \geq deg(v_{n_1})$. Then we have:

$$e = \sum_{i=1,\ldots,t} deg(v_i) + \sum_{i=t+1,\ldots,n_1} deg(v_i) \leq n_2 + \binom{t}{2} + \sum_{i=t+1,\ldots,n_1} deg(v_i).$$

Since $deg(v_t) \geq deg(v_{t+1}) \geq \ldots \geq deg(v_{n_1})$, we have:

$$e \leq n_2 + \binom{t}{2} + (n_1 - t) deg(v_t).$$

On the other hand, $deg(v_t) \leq deg(v_{t-1}) \leq \ldots \leq deg(v_1)$. So:

$$deg(v_t) \leq \frac{\sum_{i=1,\ldots,t} deg(v_i)}{t} \leq \frac{n_2 + \binom{t}{2}}{t},$$

and

$$e \leq n_2 + \binom{t}{2} + (n_1 - t)\frac{n_2 + \binom{t}{2}}{t}.$$

□

3.7.2 Parental Rules

Given the definitions in Section 3.4.3, in this section, we provide a number of theorems that determine necessary conditions for a class, p, to be a parent of another class, c. The efficiency increases using these theorems by having a smaller generation tree, ruling out classes that are not required.

Theorem 3.7.9. *If $G \in \mathcal{G}_{C_3}$ is of type B, then the parent of G is of type A.*

Proof. Let G' be the parent of G and u be a winning vertex of G that is inserted last during the generation process of G. We have $V(G') = V(G) \setminus \{u\}$. Since G is of type B, we have $mins(G') \subseteq N_G(u)$. Therefore, any two vertices in $mins(G')$ are independent, otherwise G contains a C_3 that includes u. Hence, $mins(G')$ is independent and G' is of type A. □

The two following theorems, determines some specifications of parents of a class of type A.

Theorem 3.7.10. *Let $p, c \in U$ be two classes where $p = (n' = n-1, e' = e-d, d', m', t')$ is a parent of class $c = (n, e, d, m, t = A)$.*

1. *If $m = 1$, then:*
   - *we have $d' \geq 1$ when $d = 0$, and*
   - *we have $d' \geq d$ when $d > 0$*
2. *If $m = 1$ and $d' = d$, then $m' \leq d$.*
3. *If $m > 1$, then $d' = d$.*
4. *If $m > 1$ and $t' = B$, then $m' \geq m$.*

Proof. Let $G \in S_c$ and $G' \in S_p$ be two arbitrary graphs where G' is the parent of G and we have $G' = G - \{u\}$. We prove each part of the theorem, as follows:

1. The vertex u is the only vertex in G of degree d and the degree of any other vertex in G is at least $d+1$. When $d = 0$, removing u does not change the degree other vertices. So any vertex in the parent graph is of degree at least $d + 1 = 1$. When $d > 0$, removing u can decrease the degree of vertices in $N_G(u)$ by 1 and the degree of other vertices remain unchanged. Hence, any vertex in G' is of degree at least $d + 1 - 1 = d$.

2. Here again, the vertex u is the only vertex in G of degree d and the degree of any other vertex in G is at least $d+1$. Since $d' = d$, we have $\delta(G') = d = \delta(G)$. Therefore:
   $$mins(G') = \{v \mid deg_G(v) = d + 1 \ \$ \ v \in N_G(u)\}.$$
   Hence, $mins(G') \subseteq N_G(u)$ and $m' = \left|mins(G')\right| \leq \left|N_G(u)\right| = d$. That is, $m' = \leq d$.

3. This one is trivial.

4. Since $t = A$, the set $mins(G)$ is independent. Therefore, by removing $u \in mins(G)$, from G, the degree of all other vertices in $mins(G)$ remains d and since $m > 1$, we have $d' = d$ and $mins(G') = mins(G) \setminus \{u\} \cup D$ where:

$$D = \big\{v \mid v \in N_G(u) \quad \& \quad deg_G(v) = d + 1\big\}.$$

Since G' is of type B and $mins(G) \setminus \{u\}$ is independent, there must be at least on vertex in D that is adjacent to another vertex in $mins(G')$. Hence, $|D| \geq 1$ and $m' = |mins(G')| \geq |mins(G)| - 1 + 1 = m$.

□

Theorem 3.7.11. *If $G \in \mathcal{G}_{\mathcal{C}_3} \cap S_{(n,e,d,1,A)}$ is a child of $G' \in \mathcal{G}_{\mathcal{C}_3} \cap S_{(n'=n-1,e'=e-d,d'=d,m',t')}$, then $t' = A$.*

Proof. If the parent graph, G', is of type B, then there are adjacent vertices, v and w, of degree d in G'. Since $m = 1$, we have $deg_G(v) = deg_G(w) = d + 1$ and $v, w \in N_G(u)$. Therefore, u, v and w form a C_3 which is a contradiction. □

Let $p, c \in U$ be two classes where $p = (n', e', d', m', t')$ is a parent of class $c = (n, e, d, m, t)$. The theorem above states that if C_3 is in the set of forbidden cycles and $m = 1$, $t = A$, $d' = d$, then $t' = A$.

3.7.2.1 Graphs of Type A with Parents of Type A

In this section, we provide a number of lemmas, based on which, we decide if a class with graphs of type A can have a parent whose graphs are also of type A, when $m > 1$. Given a class, $c = (n, e, d, m, A)$, we want to determine all parents of c with graphs of type A. Let $p = (n', e', d', m', A)$ be a parent of c. Clearly, we have $n' = n - 1$, $e' = e - d$ and $d' = d$. Therefore, we only need to determine the possible values for m'. Throughout this section, we assume $G' \in S_{(n',e',d',m',A)}$ is parent of $G \in S_{(n,e,d,m,A)}$ where $m > 1$, i.e., G can is obtained by adding a vertex of minimum degree to G'. It is easy to verify the following lemma:

Lemma 3.7.1. *Let G and G' be two graphs defined above, then we have $n' = n - 1$, $e' = e - d$, $d' = d$ and $m - 1 \leq m' \leq m - 1 + d$.*

By this lemma, we know $G' \in S_{(n-1,e-d,d,m',A)}$, but we need to determine if such a graph exists, i.e., if we have $S_{(n-1,e-d,d,m',A)} \neq \emptyset$. Our approach is to exhaustively consider all possible values for m'. And for each value, we consider all possible structures of graphs in $S_{(n-1,e-d,d,m',A)}$ based on some invariants such as the number edges within the set of vertices of minimum degree, or the number of vertices of degree $d + 1$ connected to a winning vertex. The lemmas in this section, rule out some impossible structures and hence, some impossible values for m'.

Let G be a graph. We use the following definitions in the lemmas in this section:

- $X = mins(G) = \{u \in V(G) : deg(u) = d\}$, therefore, $m = |X|$,
- $Y = \{u \in V(G) : deg(u) = d+1\}$ and $q = |Y|$,
- $Z = \{u \in V(G) : deg(u) \geq d+2\}$ and $r = |Z|$,
- $X_X = \{\{uv\} \in E(G) : u, v \in X\}$,
- $Y_Y = \{\{uv\} \in E(G) : u, v \in Y\}$ and $n_{yy} = |Y_Y|$,
- $Z_Z = \{\{uv\} \in E(G) : u, v \in Z\}$ and $n_{zz} = |Z_Z|$,
- $X_Y = \{\{uv\} \in E(G) : u \in X \& v \in Y\}$ and $n_{xy} = |X_Y|$,
- $X_Z = \{\{uv\} \in E(G) : u \in X \& v \in Z\}$ and $n_{xz} = |X_Z|$,
- $Y_Z = \{\{uv\} \in E(G) : u \in Y \& v \in Z\}$ and $n_{yz} = |Y + Z|$, and
- For every $v \in X$ we define:

$$f(v) = \left|\left\{x' \in X \mid \exists y \in Y \text{ such that } \{vy\}, \{x'y\} \in E(G)\right\}\right|.$$

It is obvious that:

- $|X_X| = 0$ (Since $type(G) = A$),
- $n_{xy} + n_{xz} = md$,
- $n_{xy} + 2n_{yy} + n_{yz} = q(d+1)$,
- $n_{xz} + n_{yz} + 2n_{zz} \geq r(d+2)$,
- $2e = md + q(d+1) + (n_{xz} + n_{yz} + 2n_{zz})$.

Let u be a winning vertex of G that is inserted last during the generation process of G. We have $V(G') = V(G) \setminus \{u\}$ and we define:

$$d_1 = \left|\left\{y \in Y \mid y \text{ is adjacent to } u\right\}\right|.$$

Lemma 3.7.2. $n_{xy} \leq q$.

Proof. Suppose $n_{xy} > q$. Then there is a vertex, w_1, in Y that is adjacent to at least two vertices in X, say x_1 and x_2. So we have $f(x_1) \geq 2$. Since G is of type A and u is a winning vertex of G that is inserted last, i.e., $V(G') = V(G) \setminus \{u\}$, by our definition of genuine reduction in Section 3.4.2, we have $f(u) \geq f(x_1) \geq 2$. That means there is a vertex $w_2 \in Y$ where w_2 is adjacent to u and some other vertex in X, say v. But then after deleting u. The vertices w_2 and v are both of the minimum degree and they are adjacent to each other and G' would be of type B which is a contradiction. □

The following lemma can be proven similarly:

Lemma 3.7.3. *No pair of vertices in X can have a common neighbour in Y. That is, each vertex in Y have at most one neighbour in X.*

Lemma 3.7.4. *If $d_1 = 0$ then $n_{xy} = 0$.*

Proof. Having $d_1 = 0$, implies $f(u) = 0$. If $n_{xy} > 0$ then there will be a vertex $v \in X$ other than u, that has a neighbour in Y. But then, we have $f(v) \geq 1 \geq f(u)$ which is a contradiction with our definition of genuine reduction . □

Lemma 3.7.5. *We have $2e - md - q(d+1) \geq n_{xz}$.*

Proof. If $2e - md - q(d+1) < n_{xz}$, then we have:

$$2e < md + q(d+1) + n_{xz} < md + q(d+1) + n_{xz} + n_{yz} + 2n_{zz}$$

$$= \sum_{x \in X} deg(x) + \sum_{y \in Y} deg(y) + \sum_{z \in Z} deg(z) = \sum_{u \in V(G)} deg(u) = 2e.$$

That is, $e2 < 2e$, which is a contradiction. □

Lemma 3.7.6. *We have $2e - md - q(d+1) \geq r(d+2)$.*

Proof. If $2e - md - q(d+1) < r(d+2)$, then we have:

$$2e < md + q(d+1) + r(d+2) \leq \sum_{u \in X} deg(u) + \sum_{u \in Y} deg(u) + \sum_{u \in Z} deg(u) = \sum_{u \in V(G)} deg(u) = 2e.$$

That is, $2e < 2e$, which is a contradiction. □

Lemma 3.7.7. *A graph has an odd cycle if it contains a closed walk of an odd length.*

Proof. Let G be a graph containing a closed walk, W, of an odd length and let $C = W$. Consider a vertex, v, in the walk and traverse G starting from v, following the edges of W. Whenever we visit a vertex y that has been already visited, if the walk is not finished yet, we discard from the sequence of edges in C all the edges (in the cycle) we last traversed since the previous time we visited y. If we face any odd cycle, then we are done. Otherwise, the total number of edges discarded from C is even. Since C had initially an odd number of edges, at the end of our traversal, the number of edges remained in C is odd form a cycle containing v. □

Lemma 3.7.8. *If $|X_Y|, |X_Z|, |Y_Z| > 0$ and there is an even path between any two vertices that are both in X, both in Y, or both in Z, then G has an odd cycle and is not bipartite.*

Proof. Let $\{x_1, y_1\} \in X_Y$, $\{y_2, z_2\} \in Y_Z$, $\{z_3, x_3\} \in X_Z$ and P_1, P_2 and P_3 be even paths from x_3 to x_1, from y_1 to y_2, and from z_2 to z_3, respectively. Then $P_1 \cup P_2 \cup P_3 \cup \Big\{\{x_1, y_1\}, \{y_2, z_2\}, \{z_3, x_3\}\Big\}$ is a closed walk in G of an odd length. So by Lemma 3.7.7, G has an odd cycle. □

3.7.2.2 Number of Edges Between Vertices in $mins(G)$

In this section, we provide some theorems that help calculate a lower bound for the number of edges in the subgraph of a graph, G that is induced by the set of vertices of minimum degree, $mins(G)$. Having this lower bound larger than zero shows G is of type B and when this lower bound is negative, a graph with those specification does not exist, i.e., the set of graphs of a class with those specifications is empty. Also, this lower bound gives a lower bound for the minimum degree in the subgraph of G induced by $mins(G)$ and hence, when G is of type B, it gives a lower bound for other invariants of graphs and hence, reduces our search space.

The following theorem, gives upper bounds for this lower bound and hence, restricts our search space to exhaustively look for a lower bound. The next three lemmas, help to rule out some values for $R(G)$ by checking whether there can be a bipartite graph between $mins(G)$ and $V(G)/mins(G)$.

Theorem 3.7.12. *For any graph $G \in \mathcal{G}_{\mathcal{C}} \cap S_{(n,e,d,m,A)}$, let $R(G)$ be the number of edges in the subgraph of G induced by $mins(G)$. Then we have:*

$$R(G) \leq \min\Big(md/2,\ \ ex(m,\mathcal{C}),\ ex(n-m,\mathcal{C}) + md - e\Big).$$

Proof. Let G_S be the subgraph of G induced by $mins(G)$. G_S has m vertices and the degree of each vertex is at most d. Therefore, the total degree of vertices in G_S is at most md and we have $R(G) \leq md/2$.
It is also obvious that $R(G) \leq ex(m,\mathcal{C})$ otherwise G_S and hence, G contains a forbidden cycle.
We also have $e = e_1 + e_2 + R(G)$ where e_1 is the number of edges with no end in $mins(G)$ and e_2 is the number of edges with exactly one end in $mins(G)$. On the other hand, we have $e_2 + R(G) = md - R(G)$. Therefore, $e = e_1 + md - R(G)$ and $R(G) = e_1 + md - e \leq ex(n-m,\mathcal{C}) + md - e$. □

Lemma 3.7.9. *In a bipartite graph, if the vertices of one part are all in one connected component, then adding an edge between two vertices of that component will make the graph non-bipartite.*

Proof. Since the graph is bipartite, any path between any two vertices in one part is even. Now if the vertices of one part are all in one connected component, then there is an odd path between any two vertices. Therefore, adding an edge between any two vertices in that part forms an odd cycle. □

Lemma 3.7.10. *Let G be a bipartite graph where $V(G) = V_1 \cup V_2$ and for $i = 1, 2$, $uc_i(G)$ be the number of pairs of vertices, $\{u, v\}$, in V_i where u and v do not share a neighbour. Then adding at least $uc_i(G) + 1$ edges between the vertices in V_i forms a triangle.*

Proof. Obviously adding an edge between a pair of vertices in V_i that share a neighbour, forms a C_3. If we add more than $uc_i(G)$ edges between the vertices in V_i, then at least one of these edges is added to between two vertices that share a neighbour and hence, a C_3 is created. □

The proof of the following lemma is trivial.

Lemma 3.7.11. *For each class $c = (n, e, d, m, t) \in U$ where $n = m$, if $e \neq \frac{md}{2}$ then $S_c = \emptyset$, otherwise for each $G \in S_c$ there are exactly $e = \frac{md}{2}$ edges with both ends in $mins(G)$.*

3.7.2.3 Bipartite Graphs with no C_4

For a given class, $c = (n, e, d, m, t) \in U$, we have $S_c \cap \mathcal{G}_{\{C_4\} \cup \mathcal{B}} = \emptyset$ if there for all n_1 and m_1, there is no bipartite graph with $n_1 + n - n_1$ vertices, m_1 and $m - m_1$ vertices of degree d in the first and second part, respectively. Therefore, Theorem 3.7.13, is a tool to rule out classes with no graph in $\mathcal{G}_{\{C_4\} \cup \mathcal{B}}$. The following three lemmas are used to prove this theorem, where we define a pair of vertices $\{x, y\}$ to be *covered by a vertex*, z, if z is adjacent to both x and y. Let $ncov(z)$ be the number of pairs of vertices that are covered by z. We say a pair of vertices is *covered*, if it is covered by at least one vertex. For a bipartite graph, G where $V(G) = V_1 \cup V_2$, we define $tcov_2(G) = \sum_{z \in V_1} ncov(z)$.

Lemma 3.7.12. *Let $G \in \mathcal{G}_{\mathcal{C}_4 \cup \mathcal{B}}$ be a bipartite graph with $n = n_1 + n_2$ vertices. We have $tcov_2(G) \leq \binom{n_2}{2}$.*

Proof. If $tcov_2(G) > \binom{n_2}{2}$, then by the pigeonhole principle, there is at least one pair of vertices, say $\{x, y\}$, in V_2 that is covered by at least two vertices, say w and z, in V_1. Hence, we have $\{w, x\}, \{x, z\}, \{z, y\}, \{y, w\} \in E(G)$ and there is a C_4 between vertices x, y, z and w, which is a contradiction. □

Lemma 3.7.13. *Let S be the set of all sets consisting of h non-negative integers with total value of R and $a = \{a_1, \ldots, a_h\}$ be the set in S where for all $1 \leq i, j \leq h$, we have $|a_i - a_j| \leq 1$. If $f(s) = \sum_{k=1,\ldots,h} \binom{s_k}{2}$, then we have $f(a) \leq f(s)$ for each $s \in S$.*

Proof. First note that there can only be one set in S with the same property as a. We show that for any $s \in S/\{a\}$, there is $s' \in S$ where $f(s') < f(s)$ and therefore, $f(a) \leq f(s)$ for each $s \in S$. Consider an arbitrary set $s \in S/\{a\}$. Since $s \neq a$, there are x and y where $1 \leq x, y \leq h$ and $s_y - s_x > 1$. Therefore,

$$\sum_{i=1,\ldots,h} \binom{s_i}{2} = \binom{s_x}{2} + \binom{s_y}{2} + \sum_{i=1,\ldots,h \ \&\ i \neq x,y} \binom{s_i}{2}.$$

Now consider the set $s' = s/\{s_x, x_y\} \cup \{s_x + 1, s_y - 1\}$. Obviously $s' \in S$. We have:

$$f(s') = f(s) - \binom{s_x}{2} - \binom{s_y}{2} + \binom{s_x + 1}{2} + \binom{s_y 01}{2}$$

$$= f(s) + \frac{s_x(s_x+1) - s_x(s_x-1) + (s_y-1)(s_y-2) - s_y(s_y-1)}{2}$$
$$= f(s) + s_x - s_y + 1 < f(s).$$

That is $f(s') < f(s)$. □

Lemma 3.7.14. *If* $\sum_{k=1,\dots,h}(s_k) = R$, *then:*

$$\sum_{k=1,\dots,h}\binom{s_k}{2} \geq p\binom{r}{2} + q\binom{r+1}{2},$$

where $r = \lfloor R/h \rfloor$, $q = R - hr$ *and* $p = h - q$.

Proof. Let $a_1, \dots, a_h$ be a sequence where $\sum_{k=1,\dots,h}(a_k) = R$ and for all $1 \leq i, j \leq h$, we have $|a_i - a_j| \leq 1$. According to Lemma 3.7.13, we have:

$$\sum_{k=1,\dots,h}\binom{s_k}{2} \geq \sum_{k=1,\dots,h}\binom{a_k}{2}.$$

On the other hand, it can be easily verified for each $1 \leq i \leq h$, we have $a_i \in \{\lfloor R/h \rfloor, \lfloor R/h \rfloor + 1\}$. In fact, $R - h\lfloor R/h \rfloor$ elements of a has the value of $\lfloor R/h \rfloor + 1$, and $h - (R - h\lfloor R/h \rfloor)$ elements of a has the value of $\lfloor R/h \rfloor$. Therefore:

$$\sum_{k=1,\dots,h}\binom{a_k}{2} = (R - h\lfloor R/h \rfloor)\binom{\lfloor R/h \rfloor + 1}{2} + (h - (R - h\lfloor R/h \rfloor))\binom{\lfloor R/h \rfloor}{2}.$$

Now if $r = \lfloor R/h \rfloor$, $q = R - hr$ and $p = h - q$, then:

$$\sum_{k=1,\dots,h}\binom{s_k}{2} \geq q\binom{r+1}{2} + p\binom{r}{2}.$$

□

Theorem 3.7.13. *Let* G *be a bipartite graph in* $\mathcal{G}_{C_4 \cup \mathcal{B}} \cap S_{(n=n_1+n_2,e,d,m=m_1+m_2,t)}$ *where* $V(G) = V_1 \cup V_2$, $|V_1| = n_1$, $|V_2| = n_2$ *and there are* m_1 *and* m_2 *vertices of degree* d *in* V_1 *and* V_2, *respectively. If* $k = (e - m_1 d)/(n_1 - m_1)$, $q = e - md - k(n_1 - m_1)$ *and* $p = (n_1 - m_1) - q$, *then we have:*

$$m_1\binom{d}{2} + p\binom{k}{2} + q\binom{k+1}{2} \leq \binom{n_2}{2}.$$

Proof. There are m_1 vertices, say $u_1, \dots, u_{m_1}$ in V_1 of degree d, and $n_1 - m_1$ vertices, say $v_1, \dots, v_{n_1-m_1}$ in V_1 of degree larger than d. Suppose d_i be the degree of v_i for each $i = 1, \dots, n_1 - m_1$. We have:

$$\sum_{i=1,\dots,n_1-m_1} d_i = e - m_1 d.$$

Therefore, by Lemma 3.7.14,

$$\sum_{i=1,\dots,n_1-m_1} \binom{d_i}{2} \geq p\binom{r}{2} + q\binom{r+1}{2}$$

where $r = \lfloor (e - m_1 d)/(n_1 - m_1) \rfloor$, $q = (e - m_1 d) - (n_1 - m_1)r$ and $p = (n_1 - m_1) - q$. On the other hand, by Lemma 3.7.12, we have:

$$\binom{n_2}{2} \geq tcov_2(G) = \sum_{z \in V_1} ncov(z) = \sum_{z \in V_1} \binom{deg(z)}{2}$$

$$= \sum_{i=1,\dots,m_1} \binom{d}{2} + \sum_{i=1,\dots,n_1-m_1} \binom{d_i}{2} \geq m_1\binom{d}{2} + p\binom{r}{2} + q\binom{r+1}{2}.$$

Therefore,

$$\binom{n_2}{2} \geq m_1\binom{d}{2} + p\binom{r}{2} + q\binom{r+1}{2}.$$

□

3.7.2.4 Bipartite Subgraphs

In many cases, to decide whether the set of graphs of a given class is empty, we use the theorems provided in this sections on the existence and the structure of bipartite graphs. Given n_1, n_2, e, d, the theorems in this section, help to determine whether a bipartite graph of order $n_1 + n_2$ with e edges exists where at least one vertex in the first part is of degree d. They also give some useful information about the structure of such graphs.

Theorem 3.7.14. *Let G be a (n_1, n_2)-bipartite graph with parts V_1 and V_2, where $|V_1| = n_1$. All vertices in V_1 are in one connected component if the number of covered pairs of vertices in V_1 is at least $\binom{n_1}{2} - n_1 + 2$.*

Proof. Let G be a bipartite graph satisfying the condition of the theorem. We have at least $\binom{n_1}{2} - n_1 + 2$ covered pairs of vertices in V_1. Therefore, we have $n_1 \geq 2$ and every vertex in V_1 is at least in one covered pair, otherwise the number of covered pairs is at most

$$\binom{n_1 - 1}{2} = \frac{(n_1 - 1)(n_1 - 2)}{2} = \frac{(n_1 - 1)n_1 - 2(n_1 - 1)}{2} = \binom{n_1}{2} - n_1 + 1$$

which is a contradiction.

We proceed the proof by induction on n_1, the number of vertices in V_1. It is easy to verify the theorem holds for $n_1 = 2$. Suppose the theorem holds for some k, i.e., for any (k, n_2)-bipartite graph, if at least $\binom{k}{2} - k + 2$ pairs of vertices in the first part of the graph are covered, then all the vertices in that part are in one connected component.

Now consider an arbitrary $(k+1, n_2)$-bipartite graph, H, where at least $\binom{k+1}{2} - (k+1) + 2$ pairs of vertices in V_1^H, the first part of H, are covered. We need to show the vertices in V_1^H are in one connected component.

If all the pairs of vertices in V_1^H are covered, then the statement holds for H. Otherwise, let $\{x, y\}$ be an uncovered pair in V_1^H. By removing x, the number of covered pairs in V_1^H can be reduces by at most $k-1$ and the graph, H', obtained by removing x has at least $\binom{k+1}{2} - (k+1) + 2 - (k-1)$ covered pairs of vertices in $V_1^H/\{x\}$. On the other hand, $\binom{k+1}{2} - (k+1) + 2 - (k-1) = (k(k+1) - 2k)/2 - k + 2 = \binom{k}{2} - k + 2$.

And since H' is a $(k+1, n_2)$-bipartite graph, by our assumption, all the vertices in $V_1^H/\{x\}$ are in one component. We showed any vertex must be in a covered pair. Therefore, there exists $y \in V_1^H/\{x\}$, where there is a path of length two between x and y. Thus, the vertices in V_1^H are in one connected component. □

Corollary 3.7.15. *Let G be a $(n_1 + n_2)$-bipartite graph where $V(G) = V_1 \cup V_2$. If the number of the uncovered pairs in V_1 is at most $n_1 - 2$ then all the vertices in V_1 are in one connected component.*

Proof. If the number of uncovered pairs in V_1 is at most $n_1 - 2$, the number of covered pairs in V_1 is at least $\binom{n_1}{2} - (n_1 - 2)$ and by Theorem 3.7.14, all the vertices in V_1 are in one connected component. □

Theorem 3.7.16. *Let $G \in \mathcal{G}_{\mathcal{B}\cup\mathcal{C}_4}$ be a $(n_1 + n_2)$-bipartite graph where $V(G) = V_1 \cup V_2$. The number of covered pairs in V_2 is at least $q\binom{e/n_1}{2} + (n_1 - q)\binom{e/n_1+1}{2}$ where $q = e - n_1\lfloor e/n_1 \rfloor$ and $p = n_1 - q$.*

Proof. Since $G \in \mathcal{G}_{\mathcal{C}_4}$, if any pair of vertices, say $\{x, y\}$, in V_2 is covered by at least two vertices in V_1, say u and v, then there is a C_4 between x, y, u and v. Hence, every pair of vertices in V_2 is covered by at most one vertex in V_1 and the number of covered pairs in V_2 equals to:

$$tcov_2(G) = \sum_{z \in V_1} ncov(z) = \sum_{z \in V_1} \binom{deg(z)}{2}.$$

Let $d_1, \ldots d_{n_1}$ be the degree sequence of vertices in V_1. We have $\sum_{i=1,\ldots,n_1-m_1} d_i = e$. So by Lemma 3.7.14, we have:

$$covt_2 = \sum_{i=1,\ldots,n_1} \binom{d_i}{2} \geq p\binom{\lfloor e/n_1 \rfloor}{2} + q\binom{\lfloor e/n_1 + 1 \rfloor}{2}$$

where $q = e - n_1\lfloor e/n_1 \rfloor$ and $p = n_1 - q$. □

The proof of the following lemma is trivial.

Lemma 3.7.15. *Let G be a $(n_1 + n_2)$-bipartite graph with e edges where $V(G) = V_1 \cup V_2$. If $n_1 = 1$ the following statements hold:*

- *The vertices in V_2 are all in one connected component only if $e = n_2$.*
- *The number of uncovered pairs of vertices in V_2 is $\binom{n_2}{2} - \binom{e}{2}$.*

3.7.3 Grand Parental Rules

Let $c_1, c_2, c_3 \in U$ be three classes where c_1 is a parent of c_2 and the class c_2 is the parent of c_3. We say c_1, is a *grand parent* of c_3 (via c_2), if there is a graph in S_{c_3} that is generated in a genuine way from a graph, X, in S_{c_2} where X is generated in a genuine way from a graph in S_{c_1}.

Each class can have multiple parents. Therefore, for three given classes, $c_1, c_2, c_3 \in U$, where c_1 is a parent of c_2 and c_2 is the parent of c_3, the class c_1 is not necessarily the grand parent of c_3. That is, given two classes, $c_1, c_2 \in U$, having a class, c_2 where C_1 is a parent of c_2, and c_2 is a parent of c_3, is a necessary condition for c_1 being a grand parent of c_3, but is not a sufficient condition.

Consider two classes, $a, b \in U$ where a is a parent of b. Let $F = \{q_1, \ldots, q_h\}$ be the set of all parents of a and $J \subseteq S_a$ be the set of all graphs in S_a that can be extended in a genuine way to a graph in S_b. For $i = 1, \ldots, h$, let $K_i \subseteq S_a$ be the set of all graphs in S_a that can be generated in a genuine way from a graph in S_{q_i}. In this case, for each $i = 1, \ldots, h$, q_i is a grand parent of b if and only if $J \cap K_i \neq \emptyset$.

In our algorithms, we receive a set, I, of some classes, and we produce all graphs in those class. Let H be the set of all classes whose graphs are required to be generated during the generation process of graphs of input classes. We build a superset, K, of the set H, starting from $K = I$, recursively adding the parents of classes in K. During this recursive procedure, as we explained above, we may add classes to K where each is a parent of a parent of a class in K but is not a grand parent of any class in K. The graphs in such classes are not required to be generated during the generation of graphs of input classes and pruning the generation tree by ruling out these classes from K, increases the efficiency. Furthermore, withing each class in K/I, we only need to generate graphs in those subclasses whose corresponding parent is a grand parent of a class in K.

Unfortunately, it is not always possible to determine the exact set of grand parents of a class. However, in this section, we present a number of necessary conditions for a class to be a grand parent of a given class (via a specific class). For each two classes, a and b, where a is a feasible parent of b, these conditions help to find a smaller superset of grand parents of b (via a). Each class, q, in this superset is called a *feasible grand parent* of a (via b). In this case, the subclass, a_q, is a feasible parent of the subclass b_a. The function *Is_gpc(q, a, b)* in Algorithm 12 uses these theorems to determine if q is a feasible grand parent of b via a.

For example, consider the classes $c_3 = (24, 46, 3, 4, B)$, $c_2 = (23, 43, 2, 1, C)$ and $c_1 = (22, 41, 3, 7, B)$. The class c_1 is a feasible parent of c_2 and c_2 is a feasible parent of c_3. But Theorem 3.7.17 says c_1 is not a grand parent of c_3.

Throughout this section we assume, $G_1 \in S_{(n_1,e_1,d_1,m_1,t_1)}$, $G_2 \in S_{(n_2,e_2,d_2,m_2,t_2)}$ and $G_3 \in S_{(n_3,e_3,d_3,m_3,t_3)}$ are arbitrary graphs where and G_1 is obtained from G_2 by removing

the vertex v, a winning vertex of G_2, and G_2 is obtained from G_3 by removing the vertex u, a winning vertex of G_3. Hence, on the generation tree, G_1 is the parent of G_2 and G_2 is the parent of G_3. For $i = 1,2,3$ and for a vertex $y \in V(G_i)$, we define $deg_i(y) = deg_{G_i}(y)$ and $N_i(y) = N_{G_i}(y)$. Therefore, we have $deg_3(u) = d_3$ and $deg_2(v) = d_2$.

Theorem 3.7.17. *If $d_1 = d_3$ and $d_2 = d_1 - 1$, then $m_3 - 2 \leq m_1 \leq m_3 + 2d_2 - 2$.*

Proof. Form $d_2 = d_1 - 1$, we have $m_2 = 1$ ($t_2 = A$), and from $d_2 = d_3 - 1$, we have $t_3 = B$. Hence, we can infer no vertex in $mins(G_3)/\{u,v\}$ is adjacent to u or to v in G_3. Therefore, the degree of all vertices in $mins(G_3) \setminus \{u,v\}$ remains unchanged in G_1 and we have:

$$mins(G_3) \setminus \{u,v\} \subseteq mins(G_1).$$

Let H be the set of vertices of degree $d_1 + 1$ in G_3 that are adjacent to either u or v. Note that no vertex in of degree $d_1 + 1$ can be adjacent to both u and v in G_3. Since $t_3 = B$, we have $u, v \in mins(G_3)$ and $\{u,v\} \in E(G_3)$. Hence, $|H| \leq 2(d_3 - 1)$ and we have:

$$m_1 = |mins(G_1)| = |mins(G_3) \setminus \{u,v\}| + |H| \leq m_3 - 2 + 2(d_3 - 1) = m_3 + 2d_2 - 2.$$

And:

$$m_1 = |mins(G_1)| \geq |mins(G_3) \setminus \{u,v\}| = m_3 - 2.$$

□

Corollary 3.7.18. *If $t_3 = B$, $t_2 = A$, $m_2 = 1$ and $d_3 = d_1$, then $m_3 - 2 \leq m_1 \leq m_3 + 2d_2 - 2$.*

Proof. From $t_3 = B$, we have $d_2 = d_3 - 1 = d_1 - 1$. Therefore, by Theorem 3.7.17, we have $m_3 - 2 \leq m_1 \leq m_3 + 2d_2 - 2$. □

Theorem 3.7.19. *If $G_1, G_2, G_3 \in \mathcal{G}_{C_4}$ and $d_1 = d_3 - 2$ then $m_1 = 1$.*

Proof. Since $d_1 = d_3 - 2$, any vertex in $mins(G_1)$ is adjacent to both u and v. Suppose $m_1 \geq 2$ and we have at least two distinct vertices, say$x$ and y, in $mins(G_1)$. Then there is a C_4 in between x, y, u and v which is a contradiction. □

Corollary 3.7.20. *If $G_1, G_2, G_3 \in \mathcal{G}_{C_4}$, $t_3 = B$ and $t_2 = B$, then $m_1 = 1$.*

Proof. From $t_3 = B$ and $t_2 = B$, we infer $d_1 = d_3 - 2$ and by Theorem 3.7.19, we have $m_1 = 1$. □

Theorem 3.7.21. *If $G_1, G_2, G_3 \in \mathcal{G}_{C_4}$ and $d_2 = d_3$, $m_3 = 1$ and $t_2 = B$ ($d_1 = d_3 - 1$, $m_2 \geq 2$) then $m_1 = 1$.*

Proof. Since $d_2 = d_3$, $m_3 = 1$, we have $mins(G_2) \subseteq N_3(u)$ and $v \in N_3(u)$. Since $t_2 = B$ we have $mins(G_1) \subseteq N_2(v)$ and $mins(G_1) \subseteq mins(G_2)/\{v\}$. Therefore, $mins(G_1) \subseteq N_3(u) \cap N_2(v)$. If $m_1 > 1$, then there are distinct $x, y \in N_3(u) \cap N_2(v)$ and G_3 contains a C_4 which is a contradiction. □

Theorem 3.7.22. *If $G_1, G_2, G_3 \in \mathcal{G}_{\{C_3,C_4\}}$, $t_3 = B$, $t_2 = A$ and $m_2 \geq 2$ then $t_1 = A$.*

Proof. Since $t_3 = B$, we have $mins(G_2) \subseteq N_3(u)$ and $v \in N_3(u)$, and since $m_2 \geq 2$, we have $mins(G_1) \subseteq (N_3(u) \cup N_3(v)) \setminus \{u, v\}$. We show the subgraph of G_3, induced by $(N_3(u) \cup N_3(v)) \setminus \{u, v \setminus \{u, v\}$ is empty.

By $G_3 \in \mathcal{G}_{C_3}$, we have $N_3(u) \cap N_3(v) = \emptyset$ and no edge can have both ends in $N_3(v)$ or in $N_3(u)$. By $G_3 \in \mathcal{G}_{C_4}$, for every $x \in N_3(u) \setminus \{v\}$ and $y \in N_3(v) \setminus \{u\}$, we have $\{x, y\} \notin E(G_3)$. Thus, there is no edge in the subgraph of G_3, induced by $(N_3(u) \cup N_3(v)) \setminus \{u, v \setminus \{u, v\}$. Therefore, the subgraph of G_3, induced by $mins(G_1)$ is also empty. This means $t_1 = A$. □

Theorem 3.7.23. *If $t_3 = B$, $t_2 = A$ and $m_2 \geq 2$, then $m_1 \leq m_3 - m_2 + d_3 - 2$.*

Proof. From $t_2 = A$ and $m_2 \geq 2$, we have $d_1 = d_2$ and from$t_3 = B$, we have $d_2 = d_3 - 1$. Therefore, we have $d_1 = d_3 - 1$. Also, from $t_3 = B$ and $t_2 = A$, we have:

$$\{x \mid deg_3(x) = d_3 \ \ \& \ \ x \in N_3(u) \cap N_3(v)\} = \emptyset.$$

Therefore, we have:

$$mins(G_1) = K_1 \cup K_2 \cup K_3$$

where:

- $K_1 = \{x \mid deg_3(x) = d_3 + 1 \ \ \& \ \ x \in N_3(u) \cap N_3(v)\}$,
- $K_2 = \{x \mid deg_3(x) = d_3 \ \ \& \ \ x \in N_3(u)/\{v\} \ \ \& \ \ x \notin N_3(v)\}$, and
- $K_3 = \{x \mid deg_3(x) = d_3 \ \ \& \ \ x \in N_3(v)/\{u\} \ \ \& \ \ x \notin N_3(u)\}$.

On the other hand, we have:

$$K_1 \cup K_2 \subseteq N_3(u)/\{v\}$$

and

$$K_3 \subseteq mins(G_3)/\ (mins(G_2) \cup \{u\}).$$

Hence:

$$m_1 \leq |K_1 \cup K_2| \ + \ |K_3| \leq (d_3 - 1) \ + (m_3 - m_2 - 1) = m_3 - m_2 + d_3 - 2.$$

□

Theorem 3.7.24. *If* $G_1, G_2, G_3 \in \mathcal{G}_{C_3}$, $t_3 = B$, $t_2 = A$ *and* $m_2 \geq 2$ *then* $m_1 \leq m_3 - 2$.

Proof. Let K_1, K_2 and K_3 be the sets defined in the proof of Theorem 3.7.23. Since the conditions of Theorem 3.7.23 are also hold here, we still have:

$$mins(G_1) = K_1 \cup K_2 \cup K_3 \ .$$

By $t_3 = B$, we have $\{u, v\} \in E(G_3)$, and since $G_1, G_2, G_3 \in \mathcal{G}_{C_3}$, we have $K_1 = \emptyset$ and:

$$mins(G_1) = K_2 \cup K_3 \ .$$

It is easy to see

$$K_2 \cup K_3 \subseteq mins(G_3)/\{v, u\}$$

Therefore, $m_1 \leq m_3 - 2$. □

Theorem 3.7.25. *If* $G_1, G_2, G_3 \in \mathcal{G}_{C_4}$, $t_3 = B$, $t_2 = A$ *and* $m_2 \geq 2$ *then* $m_1 \leq m_3 - 1$.

Proof. Let K_1, K_2 and K_3 be the sets defined in the proof of Theorem 3.7.23. Since the conditions of Theorem 3.7.23 are also hold here, we still have:

$$mins(G_1) = K_1 \cup K_2 \cup K_3 \ .$$

Every vertex in K_1 is adjacent to both u and v. Therefore, if $|K_1| > 1$, graph G_3 contains a C_4 and since $G_1, G_2, G_3 \in \mathcal{G}_{C_4}$, we have $|K_1| \leq 1$. Also, as shown in the proof of Theorem 3.7.24, we have $K_2 \cup K_3 \subseteq mins(G_3)/\{v, u\}$. Hence, $m_1 \leq m_3 - 2_1 = m_3 - 1$. □

Theorem 3.7.26. *If* $G_1, G_2, G_3 \in \mathcal{G}_{C_3}$, $t_3 = B$, $t_2 = A$ *and* $m_2 \geq 2$ *then* $m_1 \leq 2m_2 - 2$.

Proof. Let K_1, K_2 and K_3 be the sets defined in the proof of Theorem 3.7.23. Since the conditions of Theorem 3.7.23 are also hold here, we have $mins(G_1) = K_1 \cup K_2 \cup K_3$. On the other hand, since $t_3 = B$, we have $\{u, v\} \in E(G_3)$, and since $G_3 \in \mathcal{G}_{C_3}$, we have $N(u) \cap N(v) = \emptyset$. Therefore:

- $K_1 = \emptyset$,
- $K_2 = \{x \mid deg_3(x) = d_3 \ \ \& \ \ x \in N_3(u)/\{v\}\}$, and
- $K_3 = \{x \mid deg_3(x) = d_3 \ \ \& \ \ x \in N_3(v)/\{u\}\}$.

The vertex u is winning in G_3 and by definition of a winning vertex we gave in Section 3.4.2, we have:

$$|K_3| + 1 \leq |K_2| + 1.$$

Also,

$$mins(G_2) = \{x \mid deg_3(x) = d_3 \ \ \& \ \ x \in N_3(u)\} = K_2 \cup \{v\}.$$

Hence, $|K_2| = m_2 - 1$. Altogether, we have:

$$m_1 = |mins(G_1)| = 0 + |K_2| + |K_3| \leq 2|K_2| = 2(m_2 - 1).$$

Thus $m_1 \leq 2m_2 - 2$. □

Theorem 3.7.27. *If $G_1, G_2, G_3 \in \mathcal{G}_{C_4}$, $t_3 = B$, $t_2 = A$ and $m_2 \geq 2$ then $m_1 \leq 2m_2 - 1$.*

Proof. The proof is similar to the proof of Theorem 3.7.19. The only difference is, here we have $|K_1| \leq 1$ as we argued in the proof of Theorem 3.7.25. Hence:

$$m_1 = |mins(G_1)| = 1 + |K_2| + |K_3| \leq 1 + 2|K_2| = 1 + 2(m_2 - 1),$$

and $m_1 \leq 2m_2 - 1$. □

3.8 Efficiency and Group Calculations

In *GCCP* method, when a graph is generated, it is accepted only if it has been generated in a genuine way. By our definitions, a graph, G, is generated in a genuine way if the vertex inserted last is a winning vertex of G. As defined in Section 3.4.2, winning vertices in a graph, are those with lexicographically largest 4-tuple $x(v) = (x_0(v), x_1(v), x_2(v), x_3(v))$ where the values of x_0, x_1 and x_2 are combinatorial invariants. When there are more than one vertex with the largest value of (x_0, x_1, x_2) and our newly added vertex is one of them, then we need to calculate $x_3()$. By definition, for each vertex v, the value of $x_3(v)$ is the largest canonical label of a vertex in the orbit containing the vertex v. Hence, can call *nauty* to canonically label the vertices and calculate $x_3()$ for each vertex. But calling *nauty* is expensive and we want to avoid it when possible. The trick is, we don't really need to calculate the value of $x_3()$ for all vertices. Let K be the set of all vertices with lexicographically largest 3-tuple $(x_0(v), x_1(v), x_2(v))$. Based on the definition of x_3, to decided whether the last inserted vertex, n (which is already in K), is winning, all we need to know is that if n is in the same orbit as the vertex in K with the largest canonical label. When the vertices of a graph are distinguished from each other, we can use vertex coloured graphs. We benefit from how *nauty* manages vertex coloured graphs. It considers the colours to come in some order; i.e., there is a 1st colour, a 2nd colour, etc.. Then the canonical labels of vertices are calculated in order of colour. That is, the vertices of the first colour have the labelled first, of the second colour next, and so on.

When the three invariants x_1, x_2 and x_3 fail to determine if vertex n is winning, we use the values of these invariants that we have calculated, to colour the graph. This colouring increases the efficiency in two ways:

- By assigning the last colour to vertices in K, we enforce the largest canonical label to be assigned to a vertex in K and we avoid the search for the vertex with

largest canonical label that is in K to check if that vertex is in the same orbit as the last inserted vertex n.

- The definition of automorphism respects colours: each vertex can only be mapped onto a vertex of the same colour. Our colouring is based on vertex invariants. Therefore, it does not change the automorphism group but it helps the efficiency of *nauty* by providing some pre-processed information and a partition where the vertices of each orbit of automorphism group are all in one part.

After colouring the vertices based on x_1, x_2 and x_3 where the vertices in K get the last colour, we try to avoid calling *nauty* using a function that starts from the partition of vertices based on the colouring and recursively creates a new partition based on the number of neighbours of vertices in each partition where the order of parts are not changed and the order of new parts are determined the same as in *nauty* . At the end, we have a partition of vertices $P = P_1, \ldots, P_h$ where for any two vertices, u and v, in a same part, say P_i, and for any $1 \leq j \leq h$, the vertices u and v have same number of neighbours in P_j. And of course, any two vertices in the same orbit of automorphism group are in the same part of P. Therefore, the last part contains all vertices in the last orbit calculated by *nauty* but it can possibly contains more vertices. When it can be shown that the last part only consists of the vertices in the last orbit, then we can decide easily about n being winning. For example when the last part, P_h is of size one, then n is the winning only if $P_h = \{n\}$. In this section, we present a number of theorems and lemmas providing sufficient conditions to have the last part equals to an orbit and thus, a call to *nauty* can be avoided. More general theorems can be found in [144]. The following definition of an *equitable* partition is used in this section.

Definition 5. *Let $P = P_1, \ldots, P_h$ be a partition of vertices of a graph, X. We say P is equitable if for any two vertices, u and v, in a same part, say P_i, and for any $1 \leq j \leq h$, the vertices u and v have same number of neighbours in P_j.*

Theorem 3.8.1. *For an undirected graph X of n vertices, any cell of size two, say $\{u, v\}$ is an orbit if $N(u) \setminus \{v\} = N(v) \setminus \{u\}$.*

Proof. It is easy to see (u, v) is an automorphism of X. □

Lemma 3.8.1. *For an undirected graph, an equitable partition of the vertices where all non-trivial cells are of size two, is the orbits partition.*

Proof. Consider an equitable partition, P, of the vertex set of an undirected graph, X, where all non-trivial cells are of size two, say $\{u_1, v_1\}, \{u_2, v_2\}, \ldots, \{u_k, v_k\}$. We now prove the permutation $\pi = (u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)$ is an automorphism of the X. Consider two vertices x and y. We must show x^π is adjacent to y^π in X^π if and only if x and y are adjacent in X.

- If one of these vertices, say x, is in a trivial cell, then for each i, the vertex x is either adjacent to both u_i and v_i, or is adjacent to neither of them. Therefore:
  - If y is also in a trivial cell, then clearly π does not change the adjacency status of x and y.
  - If y is in a non-trivial cell, , $\{y, w\}$, then since P is equitable, we have x and y are adjacent iff x and w are adjacent. On the other hand, we have $x = x^\pi$, and $w = y^\pi$. Hence, x and y are adjacent iff $x = x^\pi$ and y^π are adjacent.
- If x and y are in non-trivial cells, then:
  - If x and y are in a same cell, then obviously π has no effect on their adjacency status.
  - If x and y are in different cells, $\{x, z\}$ and $\{y, w\}$. Then we can show x and y are adjacent iff z and w are adjacent. Suppose in contrary $\{x, y\} \in E(X)$ while $\{w, z\} \notin E(X)$. Then since P is equitable, x and z are in one cell, and x has at least one neighbour in cell $\{y, w\}$, then z must be adjacent to y. Then y is adjacent to two vertices in cell $\{x, z\}$ while w has at most one neighbour in that cell which is a contradiction with P being equitable. Similarly, it is not possible to have $\{w, z\} \in E(X)$ while $\{x, y\} \notin E(X)$. Thus,x and y are adjacent iff z and w are adjacent. On the other hand we have $z = x^\pi$ $w = y^\pi$. Hence, x and y are adjacent iff x^π and y^π are adjacent.

□

Lemma 3.8.2. *For an undirected graph, an equitable partition of the vertices where all non-trivial cells are of size two except one of size three, is the orbits partition.*

Proof. Consider an equitable partition, P, of the vertex set of an undirected graph, X, where there is one cell of size three, $\{a, b, c\}$, and all other non-trivial cells are of size two, say $\{u_1, v_1\}, \{u_2, v_2\}, \ldots, \{u_k, v_k\}$. We now prove the four permutations $\pi_1 = (u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)$, $\pi_2 = (a, b)$, $\pi_3 = (a, c)$ and $\pi_4 = (b, c)$ are automorphisms of X. Consider two vertices x and y. We must show for each $i = 1, 2, 3, 4$ we have x^{π_i} and y^{π_i} are adjacent in X^{π_i} if and only if x and y are adjacent in X.

- If one of these vertices, say x, is in a trivial cell, then $x^{\pi_i} = x$ for $i = 1, 2, 3, 4$. If y is not in the cell of size three, then similar to the first part of the proof of Lemma 3.8.1, we can see π_1 has no effect of on the adjacency of x and y. On the other hand, for $i = 2, 3, 4$, we have $y^{\pi_i} = y$ and hence, π_i has no effect on the adjacency of x and y.

  If $y \in \{a, b, c\}$, thenπ_1 clearly has no effect on adjacency of x and y. So are π_i for $i = 2, 3, 4$ because x is adjacent to none or all vertices in $\{a, b, c\}$.

- If x and y are both in cells of size two, then for each $i = 2, 3, 4$, we have $x = x^{\pi_i}$ and $y = y^{\pi_i}$ and hence, x^{π_i} and y^{π_i} are adjacent in iff x and y are adjacent. The proof for π_1 is similar to the second part of the proof of Lemma 3.8.1.

- If one vertex, say $x = a$, is in the cell of size three and the other vertex, y, is in a cell of size two, say $\{y, z\}$, then one can easily verify none of the permutations above can change the adjacency status between x and y by noting either there is no edge from $\{y, z\}$ to $\{x = a, b, c\}$ or y and z are both adjacent to every vertex in $\{x = a, b, c\}$.

- If x and y are both in the cell of size three, the proof is straightforward, noting that the subgraph induced by $\{a, b, c\}$ is either a K_3 or a $\bar{K}_3$.

□

Lemma 3.8.3. *For an undirected graph, an equitable partition of the vertices where all cells are trivial except two cells of size three, is the orbits partition.*

Proof. Consider an equitable partition, P, of the vertex set of an undirected graph, X, where there are only two non-trivial cells of size three, $D_1 = \{a, b, c\}$ and $D_2 = \{u, v, w\}$. Since P is equitable, there can only be 0, 3, 6 or 9 edges between D_1 and D_2. When there are zero or 9 edges between the two cells, the proof is easy. Also, by symmetry, having 6 edges is similar to having 3 edges. Hence, we only consider the case where there are exactly three edges between the cells. Suppose these edges are $\{a, u\}$, $\{b, v\}$ and $\{c, w\}$. We claim the permutations $\pi_1 = (a, b)(u, v)$, $\pi_2 = (a, c)(u, w)$ and $\pi_3 = (b, c)(v, w)$ are automorphisms of X. For each $i = 1, 2, 3, 4$, we need to show for any two arbitrary vertices x and y, we have x^{π_i} and y^{π_i} are adjacent in X^{π_i} if and only if x and y are adjacent in X.

- If x and y, are in trivial cells, then for $i = 1, 2, 3, 4$, we have $x^{\pi_i} = x$ and $y^{\pi_i} = y$.

- If one of these vertices, say x, is in a trivial cell and the other is in a non-trivial cell, say $y = a$, then $x^{\pi_i} = x$ for $i = 1, 2, 3, 4$ and since P is equitable x is either adjacent to all or none of the vertices in D_1 and no permutations mentioned above can change the adjacency between x and y.

- If both x and y are in one cell of size three, say D_1. Since the subgraph induced by D_1 is either a K_3 or a $\bar{K}_3$, it is easy to see the claim is true.

- If both x and y are in two different cells of size three. If there is zero or nine edges between D_1 and D_2, it is trivial that our claim is true. Suppose there are exactly three edges, $\{a, u\}$, $\{b, v\}$ and $\{c, w\}$, between these two cells. Whether there is an edge between x and y, say $x = a$ and $y = u$, or there is no edge between x and

y, say $x = a$ and $y = v$, we can easily verify none of the permutations mentioned above can change the adjacency status between x and y. By symmetry, So is when there are exactly six edges between D_1 and D_2.

Therefore, $\pi_1 = (a,b)(u,v)$, $\pi_2 = (a,c)(u,w)$ and $\pi_3 = (b,c)(v,w)$ are automorphisms of X, D_1 and D_2 are orbits and P is the orbits partition. □

Lemma 3.8.4. *For an undirected graph X, let P be an equitable partition of the vertices of X where all cells are trivial except two cells of size three and one cell of size two. The partition P is the orbits partition.*

Proof. Let $D_1 = \{a,b,c\}$, $D_2 = \{u,v,w\}$ and $D_3 = \{x,y\}$. Similar to argument used in proof of, we can assume $\{a,u\},\{b,v\},\{c,w\} \in E(X)$ are the only edges between D_1 and D_2. Since each vertex in D_3 is adjacent to zero or three vertices in D_1 and zero or three vertices in D_2, we can verify the permutations $\pi_1 = (a,b)(u,v)$, $\pi_2 = (a,c)(u,w)$, $\pi_3 = (b,c)(v,w)$ and $\pi_4 = (x,y)$are automorphisms of X. □

Lemma 3.8.5. *For an undirected graph X, let P be an equitable partition of the vertices of X where all cells are trivial except one cell of size two and one cell of size four. The partition P is the orbits partition.*

Proof. Let $D_1 = \{u,v\}$ and $D_2 = \{x,y,w,z\}$ be the two non-trivial cells:
Each vertex in $\{u,v\}$ can be adjacent to none, two or four vertices in D_2. If they are each adjacent to exactly two vertices, then we suppose u is adjacent to x and y, and v is adjacent to w and z. It is now easy to verify $\pi_1 = (x,y)(w,z)$, $\pi_2 = (x,w)(y,z)(u,v)$ and $\pi_3 = (x,z)(y,w)(u,v)$ are automorphisms of X.

□

Lemma 3.8.6. *For an undirected graph X, let P be an equitable partition of the vertices of X where all cells are trivial except one cell of size four and two or more cells of size two. Then any cell of size two is an orbit.*

Proof. Let $D_1 = \{u_1,v_1\}$ and $D_2 = \{u_2,v_2\}$ be the cells of size two and $D_3 = \{a,b,c,d\}$ be the cell of size four. Each vertex in a cell of size two can be adjacent to zero, two or four vertices in D_3. If the vertices in D_1 or D_2 are adjacent to zero or four vertices in D_3, then by Lemma 3.8.5, P is an orbits partition. So we can assume each vertex in $D_1 \cup D_2$ has exactly two neighbours in D_3. Suppose u_1 is adjacent to a and b while v_1 is adjacent to c and d. If u_2 and v_2 have same neighbours of u_1 and v_1 in D_1, respectively, then it is easy to verify $\pi_1 = (a,b)(c,d)$, $\pi_2 = (a,c)(b,d)(u_1,v_1)(u_2,v_2)$ and $\pi_3 = (a,d)(b,d)(u_1,v_1)(u_2,v_2)$ are automorphisms of X. Otherwise, without loss of generality, we can assume u_2 is adjacent to x and w while v_2 is adjacent to y and z. We can verify $\pi_1' = (u_1,v_1)(u_2,v_2)(a,d)(b,c)$ is an automorphism of X. Therefore, u_1 is in the same orbit as v_1 and u_2 is in the same orbit as v_2. This means $D_1 = \{u_1,v_1\}$ and D_2 are orbits. □

Note that although the cells of size two in a partition, P, that satisfies condition in Lemma 3.8.6, are orbits, the cell of size four may not be an orbit and hence, P is not necessarily an orbits partition.

Lemma 3.8.7. *For an undirected graph X, let P be an equitable partition of the vertices of X where all cells are trivial except one cell of size five and one cell of size two. The partition P is the orbits partition.*

Proof. Since P is equitable, there is either no edges between the two non-trivial cells, or both vertices in the cell of size two are adjacent to all vertices in the cell of size five. Also, the induced subgraph by these five vertices is a $\bar{K}_5$, a C_5 or a K_5 where with each of them, it is easy to see the partition is the orbits partition. □

Lemma 3.8.8. *For an undirected graph X, let P be an equitable partition of the vertices of X where all cells are trivial except one cell of size four and one cell of size three. The partition P is the orbits partition.*

Proof. Let D_1 and D_2 be the cells of size 3 and 4, respectively. Since P is equitable, either there is no edge between D_1 and D_2, or every vertex in D_1 is adjacent to all vertices in D_2. Also, the subgraph induced by D_1 is either $\bar{K}_3$, K_3 or C_3. While he subgraph induced by D_2 is either $\bar{K}_4$, K_4, C_4 or a consists of two disjoint edges. One can verify any permutation on D_1 or D_2 is an automorphism and hence, P is the orbits partition. □

Theorem 3.8.2. *For an undirected graph of n vertices, an equitable partition of the vertices with at least $n-4$ cells is the orbits partition of the vertices of the graph.*

Proof. Consider an equitable partition, P, of $k \geq n-4$ cells, on the vertex set of a graph with n vertices.

- If $k = n$, then obviously the partition is the orbits partition.
- If $k = n-1$, then we have exactly one cell of size two, say $\{x, y\}$, and any other cell is trivial and by Lemma 3.8.1 P is the orbits partition.
- If $k = n-2$ then the only non-trivial cells can be either exactly one cell of size three or exactly two cells of size two. Hence, by Lemmas 3.8.1 and 3.8.2, P is the orbits partition.
- If $k = n-3$ then we have three cases for non-trivial cells:
  Case 1: Exactly one cell of size four, say $D = \{x, y, u, v\}$. Whether there is none, two, four or six edges between the vertices in D, it is easy to verify the three permutations $\pi_1 = (x, y)(u, v)$ $\pi_2 = (x, u)(y, v)$ and $\pi_3 = (x, v)(u, y)$ are automorphisms of the graph.
  Case 2: Exactly one cell of size three and exactly one cell of size two. Lemma 3.8.2

gives a proof for this case.
Case 3: Exactly three cells of size two. Lemma 3.8.1 gives a proof for this case.

- If $k = n - 4$ then we have five cases based on the non-trivial cells:
  Case 1: Exactly one cell of size five, say $\{x, y, u, v, w\}$:
  Since the partition is equitable, the induced subgraph by these five vertices is a $\bar{K}_5$, a C_5 or a K_5 where with each of them, it is easy to see the partition is the orbits partition.
  Case 2: Exactly one cell of size four and one cell of size two. Lemma 3.8.5 gives a proof for this case.
  Case 3: Exactly two cells of size three. Lemma 3.8.3 gives a proof for this case.
  Case 4: We have exactly one cell of size three and two cells of size two. Lemma 3.8.2 gives a proof for this case.
  Case 5: We have exactly four cells of size two. Lemma 3.8.1 gives a proof for this case.

□

Theorem 3.8.3. *For an undirected graph of n vertices, let P be an equitable partition of the vertices with $n - 5$ cells. Then any cell of size at most three is an orbit.*

Proof. The possible cases are as follows:

- All non-trivial cells are of size 2 except possibly one cell of size three. This case can be proven using Lemmas 3.8.1 and 3.8.2.
- The only non-trivial cells are two cells of size three and possibly one cell of size two: This case can be proven using Lemmas 3.8.3 and 3.8.4.
- The only non-trivial cells are one cell of size two and one cell of size five: This case can be proven using Lemmas 3.8.7.
- The only non-trivial cells are two cells of size two and one cell of size four: This case can be proven using Lemmas 3.8.6.
- The only non-trivial cells are one cell of size three and one cell of size four: This case can be proven using Lemmas 3.8.8.

□

3.9 Conclusion and Future Work

Let $\mathcal{K}$ be an arbitrary subset of $\{C_3, C_4, C_5, \ldots, C_{32}\}$. Based on the theorems and algorithms in this chapter, we developed a set programs that receive a set of forbidden

cycles, $\mathcal{C}$, and a set of classes, $\mathcal{I}$, where $\mathcal{C} = \mathcal{K}$ or $\mathcal{C} = \{C_3, C_5, C_7, \ldots\} \cup \mathcal{K}$, and $\mathcal{I} = \{c_1, c_2, \ldots, c_h\}$. For each i, the programs produce all graphs in $S_{c_i} \cap \mathcal{G}_{\mathcal{C}}$ by the generation method of *GCCP* where the generation tree is efficiently pruned using the theorems presented in this chapter.

These programs can be used to generate extremal graphs, $EX(n, \mathcal{C})$ and calculate Turán numbers, $ex(n, \mathcal{C})$, where $n \leq 64$ and $\mathcal{C} = \mathcal{K}$ or $\mathcal{C} = \{C_3, C_5, C_7, \ldots\} \cup \mathcal{K}$. Using this program, we calculated for the first time, some unknown values of $ex(n, \mathcal{C})$ and discovered the corresponding extremal graphs, $EX(n, \mathcal{C})$. Our new results are listed in tables below. More details are provided in Appendix A. A catalogue of extremal graphs we generated can be found at *http://users.cecs.anu.edu.au/ bdm/data/extremal.html*.

By investigating the extremal graphs in $EX(n, \{C_4\} \cup \mathcal{B})$, we obtained new results on the exact values of Zarankiewicz numbers. These results are presented in Table 3.9. The previously known values of Zarankiewicz numbers can be found in [94, 58, 67, 101].

a	b	$Z_{2,2}(a,b)$
21	23	108
22	22	108
21	24	110
22	23	110
23	23	115
23	24	118
24	24	122
24	25	126
25	25	130
25	26	134
25	27	138
26	26	138
26	27	142

a	b	$Z_{2,2}(a,b)$
27	27	147
27	28	151
28	28	156
27	30	160
28	29	160
28	30	165
29	29	165
29	30	170
30	30	175
30	31	180
31	31	186
31	32	187

Table 3.9: New results for the exact values of Zarankiewicz numbers, $Z_{2,2}(a,b)$

The set of extremal graphs we discovered are yet to be studied in search for interesting properties and structures. For example, according to our results, we have $ex(n, \{C_4\} \cup B) = ex(n, \{C_3, C_4, C_5\})$ for $n \leq 63$ that is the largest n for which the value of $ex(n, \{C_3, C_4, C_5\})$ is known. Also, we have $ex(n, \{C_4, C_6\} \cup B) = ex(n, \{C_3, C_4, \ldots, C_7\})$ for $n \leq 60$ that is the largest n for which the value of $ex(n, \{C_3, C_4, \ldots, C_7\})$ is known. This information inspires the natural conjecture that $ex(n, \{C_4, \ldots, C_{2k}\} \cup B) = ex(n, \{C_3, C_4, \ldots, C_{2k+1}\})$. That is, $EX(n, \{C_3, C_4, \ldots, C_{2k+1}\})$ always contains at least one bipartite graph.

	0	1	2	3	4	5	6	7	8	9
30				96	102	106	110	113	117	122

Table 3.10: New results for the exact values of $ex(n, \{C_4\})$

	0	1	2	3	4	5	6	7	8	9
30				87	90	95	99	104	109	114
40	120	124	129	134	139	145	150	156	162	168
50		176	178							

Table 3.11: New results for the exact values of $ex(n, \{C_3, C_4\})$

	0	1	2	3	4	5	6	7	8	9
20			43	45	48	50	53	55	58	62
30	65	67	70	73	77	79	82	86	89	93
40	96	100	105	107						

Table 3.12: New results for the exact values of $ex(n, \{C_4, C_5\})$

	0	1	2	3	4	5	6	7	8	9
40				106	108	110	115	118	122	126
50	130	134	138	142	147	151	156	160	165	170
60	175	180		187						

Table 3.13: New results for the exact values of $ex(n, \{C_3, C_4, C_5\})$

	0	1	2	3	4	5	6	7	8	9
20										45
30	47	49	51	53	55	58	59	61	63	66
40	68	70	72	74	77	79	81	83	86	88
50	91	94	96	98						

Table 3.14: New results for the exact values of $ex(n, \{C_3, C_4, C_5, C_6\})$

	0	1	2	3	4	5	6	7	8	9
30								56	58	60
40	63	64	66	68	70	72	75	77	80	81
50	83	85	87	90	92	94	96	98	100	103
60	105									

Table 3.15: New results for the exact values of $ex(n, \{C_3, C_4, C_5, C_6, C_7\})$

	0	1	2	3	4	5	6	7	8	9
40					108	110	115	118	122	126
50	130	134	138	142	147	151	156	160	165	170
60	175	180	186	187						

Table 3.16: New results for the exact values of $ex(n, \{C_4\} \cup B)$

	0	1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	4	5	6	8	9
10	10	12	13	14	16	18	19	20	22	24
20	25	27	29	30	32	34	36	38	40	42
30	45	46	47	49	51	53	55	56	58	60
40	63	64	66	68	70	72	75	77	80	81
50	83	85	87	90	92	94	96	98	100	103
60	105	108								

Table 3.17: New results for the exact values of $ex(n, \{C_4, C_6\} \cup \mathcal{B})$

	0	1	2	3	4	5	6	7	8	9
0	0	0	1	2	3	4	5	6	7	8
10	10	11	12	13	15	16	17	18	20	21
20	23	24	25	27	28	30	31	32	34	36
30	37	38	40	42	43	44	46	48	49	51
40	52	54	55	57	59	60	62	63	65	67
50	68	70	71	73	75	76	78	80	82	84
60	85	87	89							

Table 3.18: New results for the exact values of $ex(n, \{C_4, C_6, C_8\} \cup \mathcal{B})$

Chapter 4

A Hierarchical Canonical labelling and its Application in Generation of Graphs

Abstract

We present a new class of canonical labellings, *hierarchical canonical labelling*, for graphs in which if the vertices of a graph G are canonically labelled by $\{1, \ldots, n\}$, then $G \setminus \{n\}$ is also canonically labelled. We provide examples of such canonical labellings. As an application of these labellings, we introduce a new method of generation based on this labelling that combines the benefits of two well-known methods of generation: Orderly Generation and Generation by Canonical Construction Path.

4.1 Introduction

An unlabelled graph is an abstraction of a relation or a network between a number of objects. When we talk about unlabelled graphs, we are solely talking about the structure of graphs; purely abstract concepts. We usually add extra properties to unlabelled graphs to make it easier to present, store, manipulate or to refer to them. For example, when we draw a graph on a piece paper, we are basically adding dimensional properties to the vertices of the graph. When we represent graphs in computer memory by storing their adjacency matrices, we are actually assigning labels to their vertices. Although it is possible, at least in theory, to access and manipulate unlabelled graphs without adding extra properties, but it is not practically an easy task. For example, one may describe a series of structural properties or invariants of a graph to the point that only one unlabelled graph with those properties exists and the graph can be uniquely distinguished. Some examples of less complicated invariants of graphs are the number of vertices, different cycles in graphs, the way the cycles intersect, or the distance between the vertices of a certain degree. Another way to refer to an unlabelled graph is to assign a number or a name to each graph. The problem with these two methods is that they are practically too hard and nearly impossible for most graphs while they do not help much in accessing the subgraphs either.

In many applications, having more than one labelled graph representing each unlabelled graph does not create any problem and one can choose any arbitrary representative to work with. But in many others, it is important to have a unique representative for each unlabelled graph, specially when it comes to comparing two graphs to see if they are equivalent or isomorphic. That's where canonical labellings come into the scene. We explain this concept in the following:

In a labelled graph with n vertices, the vertices can be labelled by $\{1, 2, \ldots, n\}$. Recall from Chapter 1, an adjacency matrix of a labelled graph, G, is an $n \times n$-binary-matrix, A_G, where $A_G[i, j] = 1$ if and only if vertices i and j are adjacent in G. We say two labelled graphs are *identical* if their adjacency matrices are the same. A bijection from the vertices of a graph to itself that takes the graph to an identical graph is called an *automorphism* of that graph. The set of all automorphisms of a graph is called the *automorphism group* of the graph. The orbits of this group form a partition of the vertex set of the graph where for any two vertices i and j in one part, there is an automorphism that takes i to j. Roughly speaking, all the vertices in one part are equivalent; they have the same role in the graph. In Figure 4.1, the graphs G_1, G_2 and G_3 are identical with the adjacency matrix shown in Figure 4.2 and the permutations $(1\ 3)$ and $(0\ 1)(2\ 3)$ are two automorphisms of them.

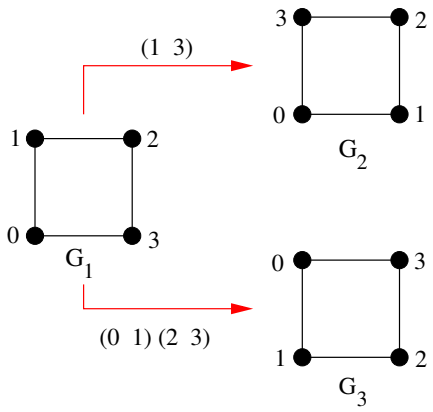

Figure 4.1: Graphs G_1, G_2 and G_3 are identical.

$$\begin{matrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{matrix}$$

Figure 4.2: The adjacency matrix of the identical graphs shown in Figure 4.1 is presented. One can verify the permutations $(1\ 3)$ and $(0\ 1)(2\ 3)$ take this matrix to itself.

Two labelled graphs are *isomorphic* if their adjacency matrices can be obtained from each other by some permutations; swapping some columns and the respective rows, i.e., there is a bijection from the vertices of one graph to the vertices of the other one that preserves the adjacency and non-adjacency of the vertices. Such bijection is called an *isomorphism* between the two graphs. All isomorphic graphs represent the same unlabelled graph. The graph G_1 is called an *isomorph* of G_2 if G_1 is isomorphic to G_2. The *canonical labelling* is defined as a way of assigning a unique labelled representative graph to each isomorphic class of labelled graphs. It is actually simply assigning a unique adjacency matrix to each unlabelled graph and fixing the set of the neighbours of each vertex. Noting this helps to avoid the confusion when we consider other ways of presenting graphs such as drawing or mapping them on a two-dimensional space.

In that sense, canonical labellings serve to unlabelled graphs as Biometric IDs do to humans, that is, every unlabelled graph has a unique corresponding canonically labelled graph. A canonical labelling can be considered as a way of assigning a unique number or a name to each unlabelled graph. For example, consider the number obtained by concatenating the rows of the adjacency matrix of a canonical isomorph. Based on these numbers, a total ordering on graphs can be defined. Unfortunately,

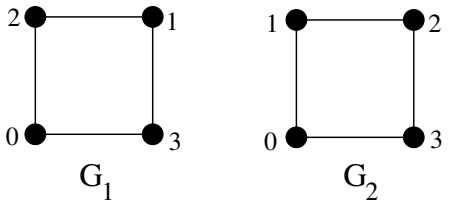

Figure 4.3: Isomorphic graphs G_0 and G_1.

0	0	1	1
0	0	1	1
1	1	0	0
1	1	0	0

0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

Figure 4.4: Adjacency matrices of isomorphic graphs G_0 and G_1 shown in Figure 4.3.

the set of Natural numbers is not a subset of the set of the numbers obtained this way from all graphs that are canonical isomorphs. In general, no polynomial algorithm is known to rank all unlabelled graphs.

The output of canonical labelling of a graph is called the *canonical isomorph*. Thus, every two isomorphic graphs have the same canonical isomorph. It is, of course, possible to have different isomorphisms between two isomorphic graphs. Therefore, an isomorphism from a graph to its canonical isomorph is not necessarily unique. For example, consider graphs G_1 and G_2 in Figure 4.3 with the adjacency matrices that are shown in Figure 4.4. The two bijections depicted in Figure 4.5 represent two different isomorphisms from G_1 to G_2. We define these concepts more precisely in the following section.

One main application of canonical labellings is in the graph isomorphism problem where we want to determine if two labelled graphs are isomorphic. Having calculated the canonical isomorphs of the two given graphs, we can decide if the two graphs are isomorphic by simply checking if their canonical isomorphs are identical. The graph isomorphism problem has many practical applications such as in enumerating different combinatorial objects. This problem is also of special interest in the complexity-theory. Isomorphism problem is clearly in the complexity class of NP but there is no known polynomial time algorithm determining if there is an isomorphism between two graphs. Hence, this problem is one of the few problems that are in NP but not known to be in P, or to be NP-complete [103]. A solution to this problem can involve testing all possible permutations in the hope of finding an isomorphism between the two graphs. The program *nauty* [144], calculates a canonical labelling

$0 \rightarrow 0$	$0 \rightarrow 0$
$1 \rightarrow 2$	$1 \rightarrow 2$
$2 \rightarrow 1$	$2 \rightarrow 3$
$3 \rightarrow 3$	$3 \rightarrow 1$
bijection1	bijection2

Figure 4.5: Two isomorphisms from G_0 to G_1 that are shown in Figure 4.3.

of graphs based on a type of backtrack search where the search tree is pruned using discovered automorphisms. Although the algorithm used to develop *nauty* has exponential running time on some inputs but it performs very efficiently for most graphs. Examples of classes of graphs where the time complexity behaviour of this algorithm is polynomial or exponential can be found in [167, 90].

In this chapter, we introduce the concept of *hierarchical canonical relabellings* for graphs where by removing the vertices with the largest labels from a canonical isomorph, the remaining subgraph is also canonical. We also give an example of applications of this labelling that introduces a new method of generation of graphs.

4.2 Hierarchical Canonical Labellings for Graphs

A canonical labelling chooses one representative for each isomorphism class of labelled graphs. More precisely a *canonical labelling* is a function, C, that takes a labelled graph, G, to a labelled graph, $C(G)$, called *canonical isomorph* of G, where:

- $C(G)$ is isomorphic to G, and
- For any graph, H, isomorphic to G we have $C(H) = C(G)$

We say a graph is a canonical isomorph, canonical, or *canonically labelled*, by a given canonical labelling if the graph is identical to its image under that canonical labelling. We define a canonical labelling function, C, to be *hierarchical* if for any n-vertex graph, G, canonical by C, and for any $k < n$, the graph obtained by removing the k largest vertices is also canonical by C.

There can be more than one isomorphism from the adjacency matrix of a graph to its canonical form. A *canonical relabelling* is a function that fixes an isomorphism or relabelling for each graph to its canonical form. In the rest of this chapter, for any labelled graph, G, and permutation, P, we define G^P to be the labelled graph obtained by applying the permutation P on the vertices (and thus edges) of G. Hence, a canonical relabelling is a function, D, that takes a labelled graph, G, to the permutation, $D(G)$, also called the *canonical isomorphism* of G, where:

- $G^{D(G)}$ is a graph isomorphic to G, and
- For any graph, H, isomorphic to G we have $H^{D(H)} = G^{D(G)}$.

Note that any canonical relabelling, D, deduces canonical labelling, C, where $C(G) = G^{D(G)}$. A canonical relabelling function is called *hierarchical canonical relabelling* if it deduces a hierarchical canonical labelling.

One simple way to define a canonical labelling is to consider the vectors obtained by concatenating the rows of the upper triangle of each adjacency matrix in an isomorphism class and define the canonical isomorph to be the matrix with the lexicographically smallest vector. It might not seem trivial but this canonical labelling is not hierarchical, i.e., if an adjacency matrix of a graph of order n is canonical, then the result of removing the last row and column, is a matrix of order $n-1$ that is not necessarily canonical. For example, the adjacency matrix A_1 shown in Figure 4.6 is in the canonical form but the adjacency matrix A_2 obtained from A_1 by removing the last row and column, is not canonical. One can verify this by considering the canonical matrix A_3 in Figure 4.7. This adjacency matrix can be obtained from A_2 by applying the permutation $[0,2,3,5,1,4]$. This means A_3 and A_2 are isomorphic. On the other hand, it is easy to see the vector corresponding to A_3 is lexicographically smaller than the vector corresponding to A_2. Hence A_2 is not in canonical form. However, if we

consider the vectors obtained by concatenating the rows of the lower triangles of each adjacency matrix in an isomorphism class and define the canonical isomorph to be the matrix with the lexicographically largest vector [79, 182], then it is easy to verify this canonical labelling is hierarchical. But this canonical labelling can not be calculated efficiently.

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \end{matrix}$$

Figure 4.6: The canonical adjacency matrix A_1.

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{matrix}$$

Figure 4.7: The canonical adjacency matrix A_3. This can be obtained by applying the permutation $[0, 2, 3, 5, 1, 4]$ on the matrix obtained by removing the last row and column of A_1.

We introduce a more efficient hierarchical canonical relabelling in Section 4.2.1. This relabelling uses a canonical relabelling function, *basic relabelling*, as its base. We present a recursive and an iterative algorithm to calculate this relabelling. This relabelling is shown to be canonical and hierarchical in Section 4.2.2.

4.2.1 The Recursive Hierarchical Canonical Labellings

Let B be a canonical relabelling. To calculate the recursive hierarchical canonical relabelling of a graph with n vertices, the labelling B is applied n times. Each time, we relabel the graph to its canonical form by B and then we fix the label of the vertex with the largest label before removing this vertex and its incident edges to obtain a graph to be considered in the next step. We repeat this process until only one vertex is remained. Relabeling the graphs at each iteration to their canonical forms is a way to remember, after deleting the last vertex, the role of each vertex in the original graph. For example, consider two vertices that are in the same orbit in the new graph but in two different orbits in the original graph.

The following Algorithms illustrate, more precisely, how we calculate this relabelling that is an isomorphism from a graph to its hierarchical canonical form.

Algorithm 17 The Getcan_Rec algorithm

```
1:  procedure GETCAN_REC(graph: G = (V, E), int: n, func: Can_Relabel )
2:      if n = 1 then
3:          can[n] = n
4:      else
5:          lab_1[] ← Can_Relabel (G, n)
6:          can[n] = lab_1[n]
7:          E' = ∅
8:          for i ← 1, ..., n − 1 do                    ▷ G' = (G \ {lab_1[n]})^{lab_1}
9:              k = lab_1[i]
10:             for each j where {j, k} ∈ E do
11:                 if j ≠ lab_1[n] then
12:                     add {lab_1^{-1}[j], i} to E'
13:                 end if
14:             end for
15:         end for
16:         lab_2[] ← Getcan_Rec( G' = (V \ {n}, E') , n − 1, Can_Relabel )
17:         for i ← 1, ..., n − 1 do
18:             can[i] ← lab_1[lab_2[i]]
19:         end for
20:     end if
21:     return can[]
22: end procedure
```

Algorithms 17 and 18 perform recursively and iteratively, respectively. They accept as input a graph, G with vertex set, V, and the edge set, E, an integer, n, that is the number of vertices of G, and a canonical relabelling function, *Can_Relabel* that serves as

the basic relabelling. The input procedure, *Can_Relabel*, takes a graph and returns its canonical isomorphism. The output, $can[]$, is an isomorphism from G to its canonical form. Note that when we present an isomorphism from a graph, G_1, to a graph, G_2, by an array, X, then $X[b] = a$ shows the bijection from vertex a in G_1 to vertex b in G_2. In Algorithms 18, at each step, the array *sofar* stores the bijection from the input graph to the isomorphic graph that is obtained up to that step.

Algorithm 18 The Getcan_It algorithm

```
1:  procedure GETCAN_IT( graph: G = (V, E), int: n, func: Can_Relabel )
2:      for i ← n, ..., 1 do
3:          sofar[i] ← i
4:      end for
5:      for i ← n, ..., 1 do
6:          lab[] ← Can_Relabel (G, i)
7:          can[i] ← sofar[lab[i]]
8:          E' = ∅
9:          for j ← 1, ..., i − 1 do
10:             sofar[j] ← sofar[lab[j]]
11:             k = lab[j]
12:             for each h where {h, k} ∈ E do
13:                 if j < lab[i] then
14:                     add {lab^{-1}[h], j} to E'
15:                 end if
16:             end for
17:         end for
18:         if i > 1 then
19:             V ← V \ {i}
20:             E ← E'
21:             G = (V, E)
22:         end if
23:     end for
24:     return can[]
25: end procedure
```

4.2.2 Formal Proof

In this section, we prove the relabelling calculated by Algorithms 17 and 18 is a hierarchical canonical relabelling. Let C be the relabelling calculated by Algorithms 17 and 18, and B be the canonical relabelling they use as the basic canonical relabelling that is calculated by the input procedure *Can_Relabel*. Theorem 4.2.1 indicates the relation C is a function. This means, C assigns a unique relabelling, $C(G)$, to each graph G. Theorem 4.2.2 and 4.2.3 show C is a canonical relabelling and Theorem 4.2.4 proves C is hierarchical.

We first describe our algorithm by defining some functions. For any graph, G, let $H_1(G) = G^{B(G)}$, $P^*_{G,1} = P_{G,1} = B(G)$ and for each $1 < i < n$, $k_i = n - i + 2$ and $A_i(G)$ be the subgraph of $H_{i-1}(G)$ induced by the set of vertices with labels smaller than k_i. We define $B_i(G)$, $P_{G,i}$, $P^*_{G,i}$ and $H_i(G)$ as follows:

$$P_{G,i} = B(A_i(G))$$
$$B_i(G) = A_i(G)^{B(A_i(G))} = A_i(G)^{P_{G,i}}$$
$$P^*_{G,i}(v) = \begin{cases} P_{G,i}(v) & v < k_i \\ v & v \geq k_i \end{cases}$$

$H_i(G)$ is the graph obtained from $B_i(G)$ by inserting the vertices $n, n-1, \ldots$ and k_i and adding their incident edges so that:

$$\begin{aligned} E(H_i(G)) &= E(B_i(G)) \\ &\bigcup \left\{ \{u,v\} |\, u,v \geq k_i \ \&\ \{u,v\} \in E(H_{i-1}(G)) \right\} \\ &\bigcup \left\{ \{u,v\} |\, u \geq k_i \ \&\ v < k_i \ \&\ \{u, P_{G,i}(v)\} \in E(H_{i-1}(G)) \right\}. \end{aligned} \tag{4.1}$$

Therefore, $H_{n-1}(G) = G^{P^*_{G,1}\, P^*_{G,2} \ldots P^*_{G,n-1}}$, and we have:

$$C(G) = P^*_{G,1}\, P^*_{G,2}\, \ldots P^*_{G,n-1} \quad ,$$

and

$$G^{C(G)} = H_{n-1}(G).$$

Figure 4.8 illustrates the algorithms in terms of A_is, B_is, H_is, $P_{G,i}$s and $P^*_{G,i}$.

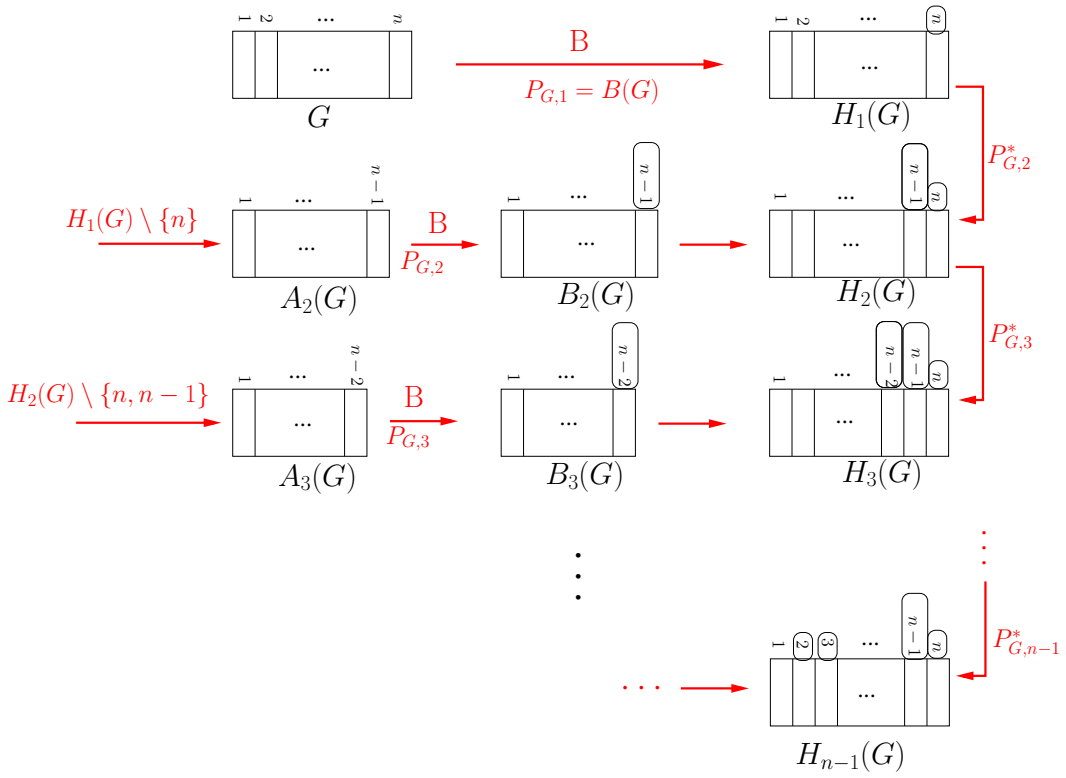

Figure 4.8: The process of calculating the hierarchical canonical relabelling by Algorithms 17 and 18 in terms of A_is, B_is, H_is, $P_{G,i}$s and $P^*_{G,i}$s is presented where each box is a graph whose vertices are sorted vertically in the order of their labels. The vertices with a circle around their labels are those whose labels have been already determined and will not be relabelled till the end of the algorithm.

Lemma 4.2.1. *For any graph, G, and for each $1 < i < n$, we have $H_i(G) = (H_{i-1}(G))^{P^*_{G,i}}$.*

Proof. We have $E(B_i(G)) = \left\{\{u,v\} |\, u,v < k_i \;\&\; \{P_{G,i}(u), P_{G,i}(v)\} \in E(H_i(G))\right\}$. Hence:

$$\begin{aligned}
&E(H_i(G)) = \\
&\left\{\{u,v\} |\, u,v < k_i \;\&\; \{P_{G,i}(u), P_{G,i}(v)\} = \{P^*_{G,i}(u), P^*_{G,i}(v)\} \in E(H_{i-1}(G))\right\} \\
&\bigcup \left\{\{u,v\} |\, u,v \geq k_i \;\&\; \{u,v\} = \{P^*_{G,i}(u), P^*_{G,i}(v)\} \in E(H_{i-1}(G))\right\} \\
&\bigcup \left\{\{u,v\} |\, u \geq k_i \;\&\; v < k_i \;\&\; \{u, P_{G,i}(v)\} = \{P^*_{G,i}(u), P^*_{G,i}(v)\} \in E(H_{i-1}(G))\right\}. \\
&= \left\{\{u,v\} |\, \{P^*_{G,i}(u), P^*_{G,i}(v)\} \in E(H_{i-1}(G))\right\}
\end{aligned} \tag{4.2}$$

Therefore:

$$H_i(G) = (H_{i-1}(G))^{P^*_{G,i}} \quad .$$

□

Lemma 4.2.2. *For any two graphs, G_1 and G_2, and for each $1 < i < n$, if $H_{i-1}(G_1) = H_{i-1}(G_2)$, then $P^*_{G_1,i} = P^*_{G_2,i}$ and $H_i(G_1) = H_i(G_2)$.*

Proof.

$$\begin{aligned}
&H_{i-1}(G_1) = H_{i-1}(G_2) \Rightarrow A_i(G_1) = A_i(G_2) \Rightarrow B(A_i(G_1)) = B(A_i(G_2)) \\
&\Rightarrow P_{G_1,i} = P_{G_2,i} \rightarrow P^*_{G_1,i} = P^*_{G_2,i}.
\end{aligned} \tag{4.3}$$

And since $H_{i-1}(G_1) = H_{i-1}(G_2)$, we have:

$$(H_{i-1}(G_1))^{P^*_{G_1,i}} = (H_{i-1}(G_2))^{P^*_{G_2,i}} \quad .$$

By applying Lemma 4.2.1, we have:

$$H_i(G_1)) = H_i(G_2)) \quad .$$

□

Theorem 4.2.1 indicates the relation C is a function.

Theorem 4.2.1. *For any two identical graphs $G_1 = G_2$ we have $C(G_1) = C(G_2)$.*

Proof.

$$G_1 = G_2 \Rightarrow B(G_1) = B(G_2) \Rightarrow P^*_{G_1,1} = P^*_{G_2,1}.$$

Also,

$$G_1 = G_2 \Rightarrow G_1^{B(G_1)} = G_2^{B(G_2)} \Rightarrow H_1(G_1) = H_1(G_2) \quad .$$

According to Lemma 4.2.2 and by induction, for all $1 \leq i < n$, we have:

$$P^*_{G_1,i} = P^*_{G_2,i} \quad \textit{and} \quad H_i(G_1) = H_i(G_2).$$

Hence:

$$P^*_{G_1,1}\, P^*_{G_1,2} \ldots P^*_{G_1,n-1} = P^*_{G_2,1}\, P^*_{G_2,2} \ldots P^*_{G_2,n-1} \Rightarrow C(G_1) = C(G_2) \quad .$$

□

Theorem 4.2.2. *For any graph, G, the graph $G^{C(G)}$ is isomorphic to G.*

Proof. We have:

$$G^{C(G)} = H_{n-1}(G) = G^{P^*_{G,1}\, P^*_{G,2} \ldots P^*_{G,n-1}} \quad .$$

Therefore, $P^*_{G,1}\, P^*_{G,2} \ldots P^*_{G,n-1}$ is an isomorphism from G to $C(G)$. □

Theorem 4.2.3. *For any two isomorphic graphs, G_1 and G_2, we have $G_1^{C(G_1)} = G_2^{C(G_2)}$.*

Proof. Since G_1 and G_2 are isomorphic, we have $G_1^{B(G_1)} = G_2^{B(G_2)}$ and hence $H_1(G_1) = H_1(G_2)$. According to Lemma 4.2.2, and by induction, for all $1 \leq i < n$ we have $H_i(G_1) = H_i(G_2)$. Therefore:

$$H_{n-1}(G_1) = H_{n-1}(G_2) \Rightarrow G_1^{C(G_1)} = G_2^{C(G_2)} \quad .$$

□

Note that when G_1 and G_2 are isomorphic, $C(G_1)$ is not necessarily the same as $C(G_2)$. The reason is, although for any $1 < i < n$ we have $P^*_{G_1,i} = P^*_{G_2,i}$, but $B(G_1)$ and $B(G_2)$, and hence, $P^*_{G_1,1}$ and $P^*_{G_2,1}$ may not necessarily be the same.

To show the relabelling we introduced is hierarchical, we consider the following definitions. For any graph H and any subset of vertices S, $H[S]$ is the subgraph of H induced by S. For each $n - k < i < n$, we define:

$$M_{k,i}(G) = (H_i(G))[\{1, \ldots, k\}].$$

Lemma 4.2.3. *For any graphs G_1 and G_2 with n and k vertices, respectively, if for some $1 \leq i < k - 1$ we have $M_{k,n-k+i}(G_1) = H_i(G_2)$ then $M_{k,n-k+i+1}(G_1) = H_{i+1}(G_2)$.*

Proof. If for some $1 \leq i < k - 1$ then:

$$M_{k,n-k+i}(G_1) = H_i(G_2) \Rightarrow A_{n-k+i+1}(G_1) = A_{i=1}(G_2) \Rightarrow B(A_{n-k+i+1}(G_1)) = B(A_{i+1}(G_2)).$$

Hence for $v < n+1-i$ we have:

$$v^{B(A_{n-k+i+1}(G_1))} = v^{B(A_{i+1}(G_2))} \Rightarrow v^{P_{G_1,n-k+i+1}} = v^{P_{G_2,i+1}}.$$

Thus, for $v \leq k$, we have:

$$v^{P^*_{G_1,n-k+i+1}} = v^{P^*_{G_2,i+1}} \Rightarrow E((H_{n-k+i+1}(G_1))[\{1,\ldots,k\}]) = E(H_{i+1}(G_2)).$$

Therefore, $M_{k,n-k+i+1}(G_1) = H_{i+1}(G_2)$. □

Theorem 4.2.4. *The canonical relabelling, C, we have defined above, is hierarchical.*

Proof. Let G be a graph with n vertices and $G' = \big(G^{C(G)}\big)[\{1,\ldots,k\}]$ for some $k < n$. It is easy to verify G' is isomorphic to $A_{n-k+1}(G)$. Hence:

$$\begin{aligned} &B(G') = B(A_{n-k+1}(G)) \\ &\Rightarrow H_1(G') = B_{n-k+1}(G) = H_{n-k+1}(G))[\{1,\ldots,k\}] = M_{k,n-k+1}(G). \end{aligned} \tag{4.4}$$

Therefore, $M_{k,n-k+1}(G) = H_1(G')$. According to Lemma 4.2.3, and by induction, for any $1 \leq i \leq k-1$ we have $M_{k,n-k+i}(G) = H_i(G')$. Thus, for $i = k-1$ we have:

$$M_{k,n-1}(G) = H_{k-1}(G').$$

On the other hand, we have:

$$M_{k,n-1}(G) = H_{n-1}(G)[\{1,\ldots,k\}] = \big(G^{C(G)}\big)[\{1,\ldots,k\}].$$

Also,

$$H_{k-1}(G') = G'^{C(G')} = \Big(\big(G^{C(G)}\big)[\{1,\ldots,k\}]\Big)^{C\left(\left(G^{C(G)}\right)[\{1,\ldots,k\}]\right)}.$$

Therefore:

$$\big(G^{C(G)}\big)[\{1,\ldots,k\}] = \Big(\big(G^{C(G)}\big)[\{1,\ldots,k\}]\Big)^{C\left(\left(G^{C(G)}\right)[\{1,\ldots,k\}]\right)}.$$

This means the graph obtained by removing the vertices larger than k from a graph canonical by C, is also canonical by C. Thus C is hierarchical. □

Note that it is important to use a canonical relabelling as the base that fixes an isomorphism from each graph to its canonical form. If B gave a random isomorphism between a graph to its canonical form (instead of a fixed canonical relabelling), then Lemma 4.2.2 can not be applied and C is not necessarily a function. This means for a graph G, there could be more than one permutation or labelled graphs calculated as $C(G)$ which does not satisfy the definition of the canonical labellings. As a simple example, let B be a random relabelling we use as the basic relabelling. Consider

identical labelled graphs, G_1 and G_2, as depicted in Figure 4.9. Let $H_1(G_1) = H_1(G_2) = G_1$ and $P^*_{G_1,1} = P^*_{G_1,1} = (1)$. The graph $A_2(G_1) = A_2(G_2)$ is a K_3 as shown in Figure 4.9. Hence, $G_1^{B(G_1)} = G_2^{B(G_2)}$ is also a K_3 with the same adjacency matrix as $A_2(G_1)$. Let $P_{G_1,2} = (1)$ while $P_{G_2,2} = (1\,3)$. This is possible since B is not a canonical relabelling and more than one isomorphism from each graph to its canonical isomorph can be defined by F. Thus, $P^*_{G_1,2} = (1)$ while $P^*_{G_2,2} = (1\,3)$. If we have $P^*_{G_1,3} = P^*_{G_2,3} = (1)$, then $P = (1)$ and $P' = (1\,3)$ are the calculated isomorphisms of G_1 and G_2, respectively, while $G_1^P = H_3(G_1) \neq H_3(G_1) = G_2^{P'}$ as it is depicted in Figure 4.9.

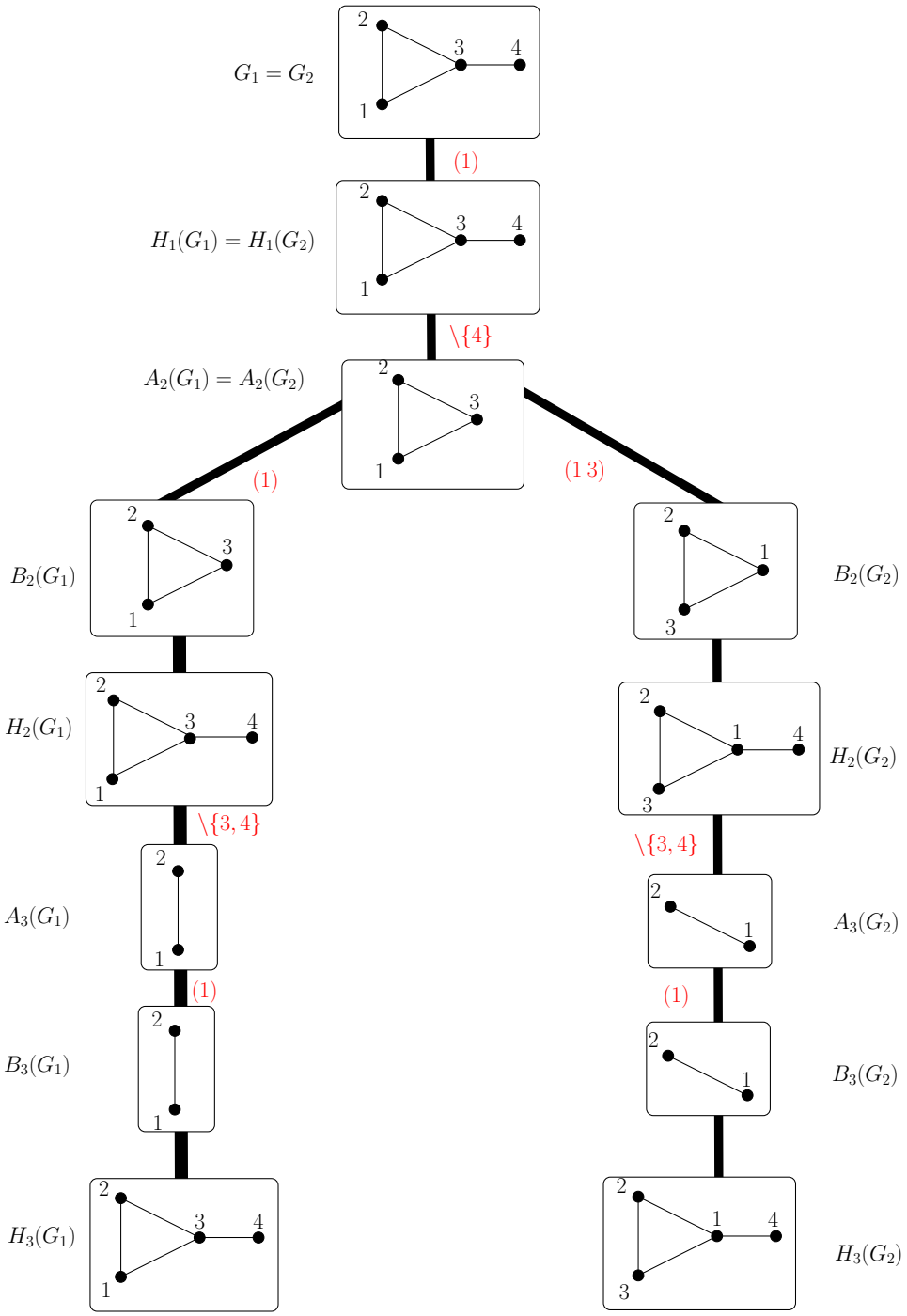

Figure 4.9: The graphs G_1 and G_2 are identical while their canonical forms $H_3(G_1)$ and $H_3(G_1)$ are not. This is an example of failing to uniquely determine the canonical form for each class of isomorphism by a similar algorithm where a canonical labelling is used instead of a canonical relabelling.

4.2.3 Implementations and Testing

We implemented the relabelling we introduced, using the canonical relabelling calculated by *nauty* as the basic relabelling. The procedure *nauty*, as we explained in Chapter 1, returns a canonical isomorph by returning the corresponding isomorphism. Since, there can be more than one isomorphism from one graph to another, *nauty* could randomly choose any isomorphism between G and the canonical isomorph it calculates. In this case, the output of *nauty* is not a canonical relabelling even though the canonical isomorph can be deduced by that.

Luckily, *nauty* has the option of performing deterministically. Having this option activated, the relabelling output by *nauty* is chosen deterministically and serves as a canonical relabelling. To have *nauty* perform deterministically, we only need to call $ran\_init(k)$ before calling *nauty*. This gives the fixed seed of k to the randomizing function and force *nauty* to perform deterministically in practice.

To test our relabelling is actually hierarchical, we took files of random graphs and applied our hierarchical relabelling to these graphs, removed the vertex with the largest canonical label from each graph and checked if the new graphs are all in the canonical form by our hierarchical relabelling. We also successfully produced all graphs up to 11 vertices.

To test our hierarchical relabelling is actually calculating a canonical labelling, we used the function *genrang* in package *nauty*, to produce a file, F, containing a number of random graphs. Each time, we used the function *ranlabg* to randomly relabel the graphs in F and produced a new file, F', whose graphs are isomorphic (but probably different) to the graphs in F, correspondingly. Next, our hierarchical relabelling was applied to the graphs in F' to obtain a new file of graphs ,F''. We, then, checked the file F'' is the same each time, that is, irrelevant to the initial labelling of a graph, when the graphs are isomorphic, applying the relabelling our method outputs for each graph, to the same graph, always gives the same labelled graph. The content of F'', can be compared with the previous content line by line. When the files are bigger we compared the results of hashing the content of the files using the function *shasum*.

4.3 Natural Orderly Generation

As mentioned in Chapter 1, generation of graphs has many applications in different fields of science as well as in industry. The main challenge in generating graph classes is to find an efficient way to avoid isomorphic copies. Generation methods such as the orderly generation [79, 182] and GCCP [146], each suggests a non-comparison based approach to avoid isomorphic copies. Each of these methods may appear more efficient than the other for generating certain classes of graphs. In Chapter 1, these two methods are described. In the orderly generation, only canonical graphs are generated. Hence canonical graphs are generated by extending smaller canonical ones. when a graph is generated, it goes through a canonicity test that involves processing the information of all vertices of the graph. This test can be, in many cases, computationally much more expensive than the genuineness test in GCCP where we basically process the information about the newly added sub-object. Although this genuineness test can be as complex as canonicity test, it is much less complex in many cases. On the other hand, after a graph, G, is accepted, in GCCP method, at the cost of calculating the automorphism group of G, we avoid applying equivalent extensions on G and hence we have a much smaller generation tree. While in the orderly generation, all the equivalent extensions are applied on G and the produced isomorphic copies are later discarded via the canonicity test. Also, GCCP allows many lookaheads that increase the efficiency. Therefore, each of these two methods can perform more efficiently than the other method for certain classes of graphs. We propose a new method of generation for graphs, the *natural orderly generation* based on hierarchical canonical labellings that combines the benefits of the orderly generation and GCCP. This method is basically a version of the orderly generation in which:

- A generated graph is accepted iff it is a canonical isomorph (the *orderly generation*).
- The extension is adding a vertex (the *natural extension*).
- The canonical labelling function is a hierarchical canonical labelling.

4.3.1 Why the Natural Extension?

The natural extension is of a particular interest with many applications in generation of graphs. The class of all graphs up to a certain number of vertices can be generated with this method. Adding a number of post-filterings can give certain classes of graphs. Our generation of PGPs is of similar nature where we post-filter any generated PGP that is not in the class of our interest and does not have any descendant in that class. Thus, we prune the generation tree at nodes where the subtrees rooted from them does not contain any node in the class of our interest.

Specific classes of graphs can be also directly generated with the natural extension. In particular, the classes that can be defined by some *hereditary* properties. A property P is hereditary if every induced subgraph of a graph with property P also has property P. Therefore, for classes of graphs with a given *hereditary* property, say P, the extensions can be limited to those where the new vertex is only added in a way that the generated graphs remain in the class. The natural extension we used with the GCCP method for generating graphs avoiding given set of cycles, is of this kind.

Therefore, with this extension, the natural orderly generation method can be used for generation of different classes of graphs.

4.3.2 Why the Hierarchical Canonical Labelling?

While in GCCP, any canonical relabelling can be considered and the choice of extensions is independent of the choice of canonical relabelling, the orderly generation endures some limitations in choosing extensions and canonical labellings. Given a class of graphs and a fixed canonical labelling, in the orderly generation, only those extensions can be considered where each canonical isomorph in the class we want to generate can be extended from another canonical isomorph.

But finding such extensions for efficient canonical labelling functions, such as the one calculated by *nauty*, can be too difficult. The hierarchical canonical labellings solve this problem with which different classes of graphs can be generated for the first time by the orderly method. The reason is if a graph G with n is a canonical isomorph according to a hierarchical canonical labelling, H, then by definition, the graph obtained from G by removing the vertex labelled n, is also a canonical isomorph according to H. This means, regardless of the exact definition of the canonical labelling, each graph with more than one vertex that is a canonical isomorph, can be extended from another canonical isomorph when the canonical labelling is hierarchical.

4.3.3 Using the Recursive Hierarchical Canonical Labelling

The recursive hierarchical canonical labelling introduced earlier in this chapter can be also used for generating graph classes by the method of natural orderly generation. As it is explained, this method of canonical labelling can be n times slower than the most efficient known canonical labelling. However, when a graph is generated, we might be able to decide whether the generated graph is a canonical isomorph (by the recursive hierarchical canonical labelling) while avoiding part of the computation needed to calculate the recursive hierarchical canonical relabelling. The idea is using the hierarchical property of this labelling, as well as the fact that the relabelling of the vertex with the k-th largest canonical label gets fixed after kth recursion. Thus, similar lookaheads as in GCCP method are also possible in the natural orderly generation that uses the recursive hierarchical canonical labelling.

4.4 Conclusion and Future Work

More Efficient Hierarchical Canonical Labellings

In this chapter, we introduced an efficient hierarchical canonical labellings. The labelling we presented is $\mathcal{O}(n)$ times slower than the most efficient known canonical relabelling where n is the number of the vertices of an input graph. The existence of a more efficient hierarchical canonical labellings is open. One natural question is:

Question: Is there any hierarchical canonical labelling with the same time complexity as of the most efficient known canonical labelling?

Increasing the Efficiency of the Natural Orderly Generation

One approach to increase the efficiency of the generation methods that use the recursive canonical relabelling we introduced here is to find a way base on the definition of the basic relabelling to decrease the number of calls to the basic relabelling or to abort early in the computations of the basic relabelling. For example, an efficient solution to the problem of finding the vertex with the largest canonical label for any (or at least one) canonical labelling function without calculating the whole labelling, would help developing more efficient generation algorithms based on the hierarchical canonical labellings.

Question: In the natural orderly generation, each graph is generated by extending a canonically labelled graph. How can this property be exploited along with the properties of the recursive hierarchical canonical relabelling to provide lookahead to decide early whether to accept or reject a generated graph without calculating the whole canonical relabelling?

k-Hierarchical Canonical Labellings

The same idea can be used for defining k-hierarchical canonical labellings where by definition if G is canonical, then removing the k largest vertices (or any number of vertices that is multiple of k) gives a canonical graph.

Note that k-hierarchical canonical labelling gives k disjoint generation trees where the roots of the trees are all graphs with $1, 2, \ldots, k$ vertices.

It is obvious if a canonical labelling is k-hierarchical, then for any natural number m, it is also mk-hierarchical. Although mk-hierarchical is weaker than k-hierarchical but the earlier can be calculated more quickly. Hence in some applications, in might be sufficient and more efficient to consider k-hierarchical labellings for $k > 1$, for example, in generation of a class of graphs where the extension is applied by adding k vertices.

Question: What are other relations between k_1-hierarchical and k_2-hierarchical canonical labellings for $1 \geq k_1 \neq k_2$?

Question: Is there any class of graphs whose generation process with k-hierarchical canonical labellings($k > 1$) is more efficient than with simple hierarchical canonical labellings?

Appendices

Appendix A

Results on Turán Numbers and Extremal Graphs

In this appendix, the known results and new results on extremal graphs with several forbidden sets of cycles are presented in a number of tables. Each table is for one set of forbidden cycles and has four columns that contain the number of vertices, n, the number of edges, e, the minimum degree, d, and the number of graphs, # with those specifications. For each number of vertices, all possible values of the number of edges and minimum degree an extremal graph are listed.

A.1 The Exact Values of $ex(n, \{C_4\})$

n	e	d	#
4	4	1	1
		total	1
5	6	2	1
		total	1
6	7	1	2
		2	2
		total	4
7	9	2	5
		total	5
8	11	2	5
		total	5
9	13	2	10
		total	10
10	16	3	2
		total	2
11	18	2	7
		3	4
		total	11
12	21	3	3
		total	3
13	24	3	2
		total	2
14	27	3	1
		total	1
15	30	4	2
		total	2
16	33	3	2
		total	2
17	36	3	1
		total	1
18	39	3	1
		total	1
19	42	3	2
		4	3
		total	5
20	46	4	1
		total	1

n	e	d	#
21	50	4	1
		total	1
22	52	2	2
		3	4
		4	7
		total	13
23	56	4	1
		total	1
24	59	3	2
		4	18
		total	20
25	63	4	9
		total	9
26	67	4	6
		5	2
		total	8
27	71	4	7
		total	7
28	76	5	1
		total	1
29	80	4	2
		total	2
30	85	5	1
		total	1
31	90	5	1
		total	1
32	92	2	1
		3	2
		4	6
		total	9
33	96	5	18
		total	18
34	102	6	1
		total	1
35	106	4	1
		total	1
36	110	5	5
		total	5

n	e	d	#
37	113	3	4
		4	7
		total	11
38	117	4	5
		5	≥ 14
		6	≥ 1
		total	≥ 20
39	122	5	≥ 1
		6	≥ 0
		total	≥ 1
40	127	5	≥ 2
		6	≥ 0
		total	≥ 2
	128	6	≥ 0
		total	≥ 0
41	132	5	≥ 3
		6	≥ 0
		total	≥ 3
	133	5	≥ 0
		6	≥ 0
		total	≥ 0
42	137	4	≥ 0
		5	≥ 2
		6	≥ 1
		total	≥ 3
	133-139	total	≥ 0
43	142	3-4	≥ 0
		5	≥ 6
		6	≥ 1
		total	≥ 7
	143-145	total	≥ 0
44	148	3-5	≥ 0
		6	≥ 2
		total	≥ 2
	149-151	total	≥ 0

n	e	d	#
45	154	3-5	≥ 0
		6	≥ 1
		total	≥ 1
	155-158	total	≥ 0
46	157	0-2	≥ 0
		3	≥ 3
		4	≥ 3
		5	≥ 0
		6	≥ 3
		total	≥ 9
	158-165	total	≥ 0
47	163	0-5	≥ 0
		6	≥ 1
		total	≥ 1
	164-170	total	≥ 0
48	168	0-4	≥ 0
		5	≥ 1
		6	≥ 8
		7	≥ 1
		total	≥ 10
	169-175	total	≥ 0
49	174	0-5	≥ 0
		6	≥ 6
		7	≥ 0
		total	≥ 6
	175-182	total	≥ 0

A.2 The Exact Values of $ex(n, \{C_3, C_4\})$

n	e	d	#
5	5	2	1
		total	1
6	6	1	1
		2	1
		total	2
7	8	2	1
		total	1
8	10	2	1
		total	1
9	12	2	1
		total	1
10	15	3	1
		total	1
11	16	1	1
		2	2
		total	3
12	18	2	5
		3	2
		total	7
13	21	3	1
		total	1
14	23	2	1
		3	3
		total	4
15	26	3	1
		total	1
16	28	2	5
		3	17
		total	22
17	31	3	14
		total	14
18	34	3	15
		total	15
19	38	4	1
		total	1
20	41	3	1
		total	1

n	e	d	#
21	44	3	1
		4	2
		total	3
22	47	3	2
		4	1
		total	3
23	50	3	5
		4	2
		total	7
24	54	4	1
		total	1
25	57	3	3
		4	3
		total	6
26	61	4	2
		total	2
27	65	4	1
		total	1
28	68	3	1
		4	3
		total	4
29	72	4	1
		total	1
30	76	4	1
		total	1
31	80	4	1
		5	1
		total	2
32	85	5	1
		total	1
33	87	2	3
		3	6
		4	3
		total	12
34	90	3	73
		4	160
		5	4
		total	237

n	e	d	#
35	95	5	5
		total	5
36	99	4	20
		5	16
		total	36
37	104	5	7
		total	7
38	109	5	2
		total	2
39	114	5	1
		total	1
40	120	6	1
		total	1
41	124	4	1
		total	1
42	129	5	1
		total	1
43	134	5	1
		total	1
44	139	5	2
		total	2
45	145	6	1
		total	1
46	150	5	1
		6	1
		total	2
47	156	6	1
		total	1
48	162	6	1
		total	1
49	168	6	1
		total	1
50	175	7	1
		total	1
51	176	1	1
		2	2
		3	2
		4	2
		total	7

n	e	d	#
52	178	2	21
		3	75
		4	49
		5	3
		total	148
53	181	3	521
		4	≥ 2074
		5	≥ 93
		6	≥ 0
		total	≥ 2688
	181–182	total	≥ 0
	183	6	≥ 0
		total	≥ 0

A.3 The Exact Values of $ex(n, \{C_3, C_4, C_5\})$

n	e	d	#
3	2	1	1
		total	1
4	3	1	2
		total	2
5	4	1	3
		total	3
6	6	2	1
		total	1
7	7	1	1
		2	1
		total	2
8	9	2	1
		total	1
9	10	1	2
		2	2
		total	4
10	12	2	3
		total	3
11	14	2	1
		total	1
12	16	2	1
		total	1
13	18	2	1
		total	1
14	21	3	1
		total	1
15	22	1	1
		2	2
		total	3
16	24	2	3
		3	1
		total	4
17	26	2	4
		total	4
18	29	3	1
		total	1
19	31	2	1
		total	1

n	e	d	#
20	34	3	1
		total	1
21	36	2	1
		3	2
		total	3
22	39	3	2
		total	2
23	42	3	1
		total	1
24	45	3	1
		total	1
25	48	3	1
		total	1
26	52	4	1
		total	1
27	53	1	1
		2	2
		3	1
		total	4
28	56	4	1
		total	1
29	58	2	1
		total	1
30	61	3	1
		total	1
31	64	3	1
		total	1
32	67	3	2
		4	3
		total	5
33	70	3	3
		total	3
34	74	4	1
		total	1
35	77	3	1
		total	1
36	81	4	1
		total	1

n	e	d	#
37	84	3	1
		4	2
		total	3
38	88	4	2
		total	2
39	92	4	1
		total	1
40	96	4	1
		total	1
41	100	4	1
		total	1
42	105	5	1
		total	1
43	106	1	1
		2	2
		3	2
		total	5
44	108	2	6
		3	5
		4	1
		total	12
45	110	2	57
		3	119
		4	7
		total	183
46	115	5	1
		total	1
47	118	4	1
		total	1
48	122	4	1
		total	1
49	126	4	1
		total	1
50	130	4	1
		total	1
51	134	4	1
		total	1

n	e	d	#
52	138	4	2
		5	1
		total	3
53	142	4	3
		total	3
54	147	5	1
		total	1
55	151	4	1
		total	1
56	156	5	1
		total	1
57	160	4	1
		5	2
		total	3
58	165	5	2
		total	2
59	170	5	1
		total	1
60	175	5	1
		total	1
61	180	5	1
		total	1
62	186	6	1
		total	1
63	187	1	1
		2	2
		3	2
		4	1
		total	6
64	189	2	10
		3	15
		4	6
		total	31
	190	5	≥ 0
		total	≥ 0

A.4 The Exact Values of $ex(n, \{C_3, C_4, C_5, C_6\})$

n	e	d	#
3	2	1	1
		total	1
4	3	1	2
		total	2
5	4	1	3
		total	3
6	5	1	6
		total	6
7	7	2	1
		total	1
8	8	1	1
		2	1
		total	2
9	9	1	6
		2	1
		total	7
10	11	2	1
		total	1
11	12	1	4
		2	3
		total	7
12	14	2	2
		total	2
13	15	1	5
		2	10
		total	15
14	17	2	3
		total	3
15	18	1	17
		2	38
		total	55
16	20	2	13
		total	13
17	22	2	3
		total	3
18	23	1	26
		2	148
		total	174

n	e	d	#
19	25	2	48
		total	48
20	27	2	18
		total	18
21	29	2	5
		total	5
22	31	2	3
		total	3
23	33	2	2
		total	2
24	36	3	1
		total	1
25	37	1	2
		2	7
		total	9
26	39	2	3
		3	3
		total	6
27	41	2	3
		total	3
28	43	2	2
		total	2
29	45	2	1
		total	1
30	47	3	1
		total	1
31	49	3	1
		total	1
32	51	2	2
		3	3
		total	5
33	53	2	5
		3	3
		total	8
34	55	2	10
		3	10
		total	20
35	58	3	1
		total	1

n	e	d	#
36	59	1	13
		2	147
		3	45
		total	205
37	61	2	254
		3	138
		total	392
38	63	2	963
		3	452
		total	1415
39	66	3	1
		total	1
40	68	2	1
		3	10
		total	11
41	70	2	16
		3	38
		total	54
42	72	2	168
		3	252
		total	420
43	74	2	1200
		3	1952
		total	3152
44	77	3	5
		total	5
45	79	2	3
		3	19
		total	22
46	81	2	94
		3	396
		total	490
47	83	2	2882
		3	≥ 6212
		total	≥ 9094
48	86	3	≥ 54
		total	≥ 54

n	e	d	#
49	88	2	≥ 375
		3	≥ 674
		total	≥ 1049
50	91	3	≥ 29
		total	≥ 29
51	94	3	≥ 3
		total	≥ 3
52	96	2	≥ 13
		3	≥ 0
		total	≥ 13
53	98	2	≥ 36
		3	≥ 16
		total	≥ 52
	99	3	≥ 0
		total	≥ 0

A.5 The Exact Values of $ex(n, \{C_3, C_4, C_5, C_6, C_7\})$

n	e	d	#
3	2	1	1
		total	1
4	3	1	2
		total	2
5	4	1	3
		total	3
6	5	1	6
		total	6
7	6	1	11
		total	11
8	8	2	1
		total	1
9	9	1	1
		2	1
		total	2
10	10	1	7
		2	1
		total	8
11	12	2	1
		total	1
12	13	1	3
		2	2
		total	5
13	14	1	25
		2	4
		total	29
14	16	2	4
		total	4
15	18	2	1
		total	1
16	19	1	2
		2	4
		total	6
17	20	1	32
		2	33
		total	65
18	22	2	6
		total	6

n	e	d	#
19	24	2	2
		total	2
20	25	1	8
		2	24
		total	32
21	27	2	2
		total	2
22	29	2	1
		total	1
23	30	1	4
		2	33
		total	37
24	32	2	6
		total	6
25	34	2	2
		total	2
26	36	2	2
		total	2
27	38	2	1
		total	1
28	40	2	1
		total	1
29	42	2	1
		total	1
30	45	3	1
		total	1
31	46	1	1
		2	2
		total	3
32	47	1	19
		2	25
		total	44
33	49	2	4
		total	4
34	51	2	4
		3	1
		total	5
35	53	2	1
		total	1

n	e	d	#
36	55	2	1
		total	1
37	56	1	5
		2	68
		total	73
38	58	2	27
		total	27
39	60	2	24
		3	2
		total	26
40	63	3	1
		total	1
41	64	1	8
		2	58
		3	9
		total	75
42	66	2	35
		total	35
43	68	2	30
		3	2
		total	32
44	70	2	20
		3	1
		total	21
45	72	2	24
		3	12
		total	36
46	75	3	3
		total	3
47	77	2	1
		total	1
48	80	3	1
		total	1
49	81	1	2
		2	5
		3	2
		total	9
50	83	2	2
		total	2

n	e	d	#
51	85	2	4
		total	4
52	87	2	7
		3	19
		total	26
53	90	3	1
		total	1
54	92	2	1
		3	1
		total	2
55	94	2	3
		3	2
		total	5
56	96	2	6
		3	≥ 3
		total	≥ 9
57	98	2	≥ 20
		3	≥ 17
		total	≥ 37
58	100	2	≥ 106
		3	≥ 29
		total	≥ 135
	101	3	≥ 0
		total	≥ 0
59	103	2	≥ 0
		3	≥ 2
		total	≥ 2
	104	3	≥ 0
		total	≥ 0
60	105	2	≥ 7
		3	≥ 20
		total	≥ 27
	106	2	≥ 0
		3	≥ 0
		total	≥ 0
	107	3	≥ 0
		total	≥ 0

A.6 The Exact Values of $ex(n, \{C_4\} \cup \mathcal{B})$

n	e	d	#
4	3	1	2
		total	2
5	4	1	3
		total	3
6	6	2	1
		total	1
7	7	1	1
		total	1
8	9	2	1
		total	1
9	10	1	2
		2	1
		total	3
10	12	2	3
		total	3
11	14	2	1
		total	1
12	16	2	1
		total	1
13	18	2	1
		total	1
14	21	3	1
		total	1
15	22	1	1
		2	1
		total	2
16	24	2	3
		3	1
		total	4
17	26	2	4
		total	4
18	29	3	1
		total	1
19	31	2	1
		total	1
20	34	3	1
		total	1

n	e	d	#
21	36	2	1
		3	2
		total	3
22	39	3	2
		total	2
23	42	3	1
		total	1
24	45	3	1
		total	1
25	48	3	1
		total	1
26	52	4	1
		total	1
27	53	1	1
		2	1
		total	2
28	56	4	1
		total	1
29	58	2	1
		total	1
30	61	3	1
		total	1
31	64	3	1
		total	1
32	67	3	2
		4	3
		total	5
33	70	3	3
		total	3
34	74	4	1
		total	1
35	77	3	1
		total	1
36	81	4	1
		total	1
37	84	3	1
		4	2
		total	3

n	e	d	#
38	88	4	2
		total	2
39	92	4	1
		total	1
40	96	4	1
		total	1
41	100	4	1
		total	1
42	105	5	1
		total	1
43	106	1	1
		2	1
		3	1
		total	3
44	108	2	6
		3	5
		4	1
		total	12
45	110	2	57
		3	119
		4	7
		total	183
46	115	5	1
		total	1
47	118	4	1
		total	1
48	122	4	1
		total	1
49	126	4	1
		total	1
50	130	4	1
		total	1
51	134	4	1
		total	1
52	138	4	2
		5	1
		total	3

n	e	d	#
53	142	4	3
		total	3
54	147	5	1
		total	1
55	151	4	1
		total	1
56	156	5	1
		total	1
57	160	4	1
		5	2
		total	3
58	165	5	2
		total	2
59	170	5	1
		total	1
60	175	5	1
		total	1
61	180	5	1
		total	1
62	186	6	1
		total	1
63	187	1	1
		2	1
		3	1
		total	3
64	189	2	10
		3	≥ 15
		4	≥ 6
		5	≥ 0
		total	≥ 31
	190	4	≥ 0
		5	≥ 0
		total	≥ 0

A.7 The Exact Values of $ex(n, \{C_4, C_6\} \cup \mathcal{B})$

n	e	d	#
4	3	1	2
		total	2
5	4	1	3
		total	3
6	5	1	6
		total	6
7	6	1	11
		total	11
8	8	2	1
		total	1
9	9	1	1
		total	1
10	10	1	6
		2	1
		total	7
11	12	2	1
		total	1
12	13	1	3
		2	1
		total	4
13	14	1	20
		2	2
		total	22
14	16	2	4
		total	4
15	18	2	1
		total	1
16	19	1	2
		2	3
		total	5
17	20	1	27
		2	20
		total	47
18	22	2	6
		total	6
19	24	2	2
		total	2

n	e	d	#
20	25	1	8
		2	19
		total	27
21	27	2	2
		total	2
22	29	2	1
		total	1
23	30	1	4
		2	29
		total	33
24	32	2	6
		total	6
25	34	2	2
		total	2
26	36	2	2
		total	2
27	38	2	1
		total	1
28	40	2	1
		total	1
29	42	2	1
		total	1
30	45	3	1
		total	1
31	46	1	1
		2	1
		total	2
32	47	1	14
		2	13
		total	27
33	49	2	4
		total	4
34	51	2	4
		3	1
		total	5
35	53	2	1
		total	1

n	e	d	#
36	55	2	1
		total	1
37	56	1	5
		2	63
		total	68
38	58	2	27
		total	27
39	60	2	24
		3	2
		total	26
40	63	3	1
		total	1
41	64	1	8
		2	50
		3	7
		total	65
42	66	2	35
		total	35
43	68	2	30
		3	2
		total	32
44	70	2	20
		3	1
		total	21
45	72	2	24
		3	12
		total	36
46	75	3	3
		total	3
47	77	2	1
		total	1
48	80	3	1
		total	1
49	81	1	2
		2	3
		3	1
		total	6

n	e	d	#
50	83	2	2
		total	2
51	85	2	4
		total	4
52	87	2	7
		3	19
		total	26
53	90	3	1
		total	1
54	92	2	1
		3	1
		total	2
55	94	2	3
		3	2
		total	5
56	96	2	6
		3	3
		total	9
57	98	2	20
		3	≥ 17
		total	≥ 37
58	100	2	≥ 106
		3	≥ 29
		total	≥ 135
59	103	3	≥ 2
		total	≥ 2
60	105	2	≥ 7
		3	≥ 20
		total	≥ 27
	106	3	≥ 0
		total	≥ 0
61	108	2	≥ 0
		3	≥ 3
		total	≥ 3
	109	3	≥ 0
		total	≥ 0

A.8 The Exact Values of $ex(n, \{C_4, C_6, C_8\} \cup \mathcal{B})$

n	e	d	#
5	4	1	3
		total	3
6	5	1	6
		total	6
7	6	1	11
		total	11
8	7	1	23
		total	23
9	8	1	47
		total	47
10	10	2	1
		total	1
11	11	1	1
		total	1
12	12	1	7
		2	1
		total	8
13	13	1	23
		total	23
14	15	2	1
		total	1
15	16	1	3
		2	1
		total	4
16	17	1	27
		2	2
		total	29
17	18	1	157
		2	3
		total	160
18	20	2	5
		total	5
19	21	1	21
		2	6
		total	27
20	23	2	1
		total	1

n	e	d	#
21	24	1	5
		2	7
		total	12
22	25	1	116
		2	42
		total	158
23	27	2	1
		total	1
24	28	1	14
		2	35
		total	49
25	30	2	2
		total	2
26	31	1	10
		2	23
		total	33
27	32	1	459
		2	382
		total	841
28	34	2	10
		total	10
29	36	2	2
		total	2
30	37	1	12
		2	19
		total	31
31	38	1	398
		2	535
		total	933
32	40	2	10
		total	10
33	42	2	1
		total	1
34	43	1	8
		2	20
		total	28
35	44	1	545
		2	1545
		total	2090

n	e	d	#
36	46	2	26
		total	26
37	48	2	1
		total	1
38	49	1	9
		2	60
		total	69
39	51	2	1
		total	1
40	52	1	24
		2	268
		total	292
41	54	2	8
		total	8
42	55	1	138
		2	1122
		total	1260
43	57	2	8
		total	8
44	59	2	1
		total	1
45	60	1	12
		2	255
		total	267
46	62	2	3
		total	3
47	63	1	36
		2	1539
		total	1575
48	65	2	30
		total	30
49	67	2	1
		total	1
50	68	1	16
		2	1063
		total	1079
51	70	2	25
		total	25

n	e	d	#
52	71	1	692
		2	23495
		total	24187
53	73	2	612
		total	612
54	75	2	31
		total	31
55	76	1	1284
		2	$\geq$ 43991
		total	$\geq$ 45275
56	78	2	$\geq$ 1930
		total	$\geq$ 1930
57	80	2	$\geq$ 75
		total	$\geq$ 75
58	82	2	$\geq$ 4
		total	$\geq$ 4
59	84	2	$\geq$ 1
		total	$\geq$ 1
60	85	1	$\geq$ 16
		2	$\geq$ 912
		total	$\geq$ 928
	86	2	$\geq$ 0
		total	$\geq$ 0
61	87	1	$\geq$ 0
		2	$\geq$ 33
		total	$\geq$ 33
	88	2	$\geq$ 0
		total	$\geq$ 0
62	89	1	$\geq$ 0
		2	$\geq$ 3
		total	$\geq$ 3
	90	2	$\geq$ 0
		total	$\geq$ 0

References

1. E. Abajo, C. Balbuena, and A. Diánez. New families of graphs without short cycles and large size. *Discrete Appl. Math.*, 158(11):1127–1135, 2010. (cited on pages 74 and 75)

2. E. Abajo, C. Balbuena, and A. Diánez. Girth of $\{c_3, \ldots, c_s\}$-free extremal graphs. *Discrete Appl. Math.*, 2012. (cited on pages xvii, 75, 76, and 77)

3. E. Abajo and A. Diánez. Size of graphs with high girth. *Electron. Notes Discrete Math.*, 29:179–183, 2007. (cited on pages xvii, 75, 76, and 77)

4. E. Abajo and A. Diánez. Exact values of $ex(v; \{c_3, c_4, \ldots, c_n\})$. *Discrete Appl. Math.*, 158(17):1869–1878, 2010. (cited on pages xvii, 75, and 76)

5. E. Abajo and A. Diánez. Graphs with maximum size and lower bounded girth. *Appl. Math. Lett.*, 25(3):575–579, 2012. (cited on page 77)

6. E. Abajo and A. Diánez. More about graphs without three-cycles or four-cycles. 2012. (cited on page 74)

7. M. Abreu, G. Araujo-Pardo, C. Balbuena, D. Labbate, and J. Salas. Small regular graphs of girth 7. *Electron. J. Combin.*, 22(3):P3–5, 2015. (cited on page 77)

8. M. Abreu, C. Balbuena, and D. Labbate. Adjacency matrices of polarity graphs and of other C_4-free graphs of large size. *Designs, Codes and Cryptography*, 55(2-3):221–233, 2010. (cited on page 73)

9. N. Afzaly, S. Morrison, and D. Penneys. The classification of subfactors with index at most $5\frac{1}{4}$. *arXiv:1509.00038 [math.OA]*, 2015. (cited on pages 23, 24, 25, 57, 58, and 67)

10. P. Allen, P. Keevash, B. Sudakov, and J. Verstraëte. Turán numbers of bipartite graphs plus an odd cycle. *Journal of Combinatorial Theory, Series B*, 106:134–162, 2014. (cited on pages 71 and 78)

11. F. Angiulli. Enumerating consistent metaquery instantiations. *AI COMMUNICATIONS*, 18(2):117, 2005. (cited on page 3)

12. M. Asaeda and U. Haagerup. Exotic subfactors of finite depth with Jones indices. *Communications in mathematical physics*, 202(1):1–63, 1999. (cited on pages xv and 27)

13. M. Asaeda and S. Yasuda. On Haagerup's list of potential principal graphs of subfactors. *Communications in mathematical physics*, 286(3):1141–1157, 2009. (cited on page 24)

14. K. Atasu, G. Dündar, and C. Özturan. An integer linear programming approach for identifying instruction-set extensions. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 172–177. ACM, 2005. (cited on page 3)

15. D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1):21–46, 1996. (cited on page 5)

16. L. Babai. Moderately exponential bound for graph isomorphism. In *Fundamentals of Computation Theory*, pages 34–50. Springer, 1981. (cited on page 19)

17. L. Babai. Graph isomorphism in quasipolynomial time. *arXiv preprint arXiv:1512.03547*, 2015. (cited on page 19)

18. L. Babai, D. Y. Grigoryev, and D. M. Mount. Isomorphism of graphs with bounded eigenvalue multiplicity. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 310–324. ACM, 1982. (cited on page 19)

19. A. T. Balaban. Valence-isomerism of cyclopolyenes. *Revue Roumaine de chimie*, 11(9):1097, 1966. (cited on pages 2 and 9)

20. C. Balbuena, G. Araujo, P. García-Vázquez, X. Marcote, and J. C. Valenzuela. On the order of bi-regular cages of even girth. *Electron. Notes Discrete Math.*, 26:1–7, 2006. (cited on page 79)

21. C. Balbuena and P. García-Vázquez. On the minimum order of extremal graphs to have a prescribed girth. *SIAM J. Discrete Math.*, 21:251, 2007. (cited on page 77)

22. C. Balbuena, P. García-Vázquez, X. Marcote, and J. C. Valenzuela. Extremal bipartite graphs with high girth. *Ars Combin.*, 83:3–14, 2007. (cited on page 79)

23. C. Balbuena, K. Marshal, and M. Miller. Contribution to the $ex(n;\{c_4\})$ problem. 2014. (cited on pages xvii and 74)

24. P. Balister, B. Bollobás, O. Riordan, and R. Schelp. Graphs with large maximum degree containing no odd cycles of a given length. *J. Combin. Theory Ser. B*, 87(2):366–373, 2003. (cited on page 77)

25. R. A. Beezer and J. Riegsecker. Catalogs of regular graphs. *International journal of computer mathematics*, 51(1-2):1–5, 1994. (cited on page 9)

26. C. T. Benson. Minimal regular graphs of girth eight and twelve. *Canad. J. Math.*, 18:1091–1094, 1966. (cited on pages 76 and 78)

27. A. Bialostocki and J. Schonheim. On some Turán and Ramsey numbers for C_4. *Graph Theory and Combinatorics. Academic Press, London*, 1984. (cited on pages xvii, 71, and 73)

28. S. Bigelow, E. Peters, S. Morrison, and N. Snyder. Constructing the extended Haagerup planar algebra. *Acta mathematica*, 209(1):29–82, 2012. (cited on page 24)

29. J. Bion-Nadal. An example of a subfactor of the hyperfinite II_1 factor whose principal graph invariant is the Coxeter graph $E6$. *Current topics in operator algebras (Nara, 1990)*, pages 104–113, 1991. (cited on page 24)

30. D. Bisch. Principal graphs of subfactors with small Jones index. *Mathematische Annalen*, 311(2):223–231, 1998. (cited on page 24)

31. H. L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *Journal of Algorithms*, 11(4):631–643, 1990. (cited on page 19)

32. B. Bollobás. Extremal graph theory. *Academic Press,London*, 1978. (cited on pages 72 and 77)

33. J. Bondy. Large cycles in graphs. *Discrete Math.*, 1(2):121–132, 1971. (cited on pages 76 and 77)

34. J. A Bondy and M. Simonovits. Cycles of even length in graphs. *J. Combin. Theory Ser. B*, 16(2):97–105, 1974. (cited on page 76)

35. G. Brinkmann. Fast generation of cubic graphs. *Journal of Graph Theory*, 23(2):139–149, 1996. (cited on pages 2, 6, and 9)

36. G. Brinkmann and A. Dress. A constructive enumeration of fullerenes. *Journal of Algorithms*, 23(2):345–358, 1997. (cited on page 5)

37. G. Brinkmann, J. Goedgebeur, and B. D. McKay. Homepage of snarkhunter:. *http://caagt.ugent.be/cubic/*. (cited on page 9)

38. G. Brinkmann and B. D. McKay. Fast generation of planar graphs. *MATCH Commun. Math. Comput. Chem*, 58(2):323–357, 2007. (cited on page 9)

39. G. Brinkmann, B. D. McKay, and C. Saager. The smallest cubic graphs of girth nine. *Combinatorics, Probability and Computing*, 4(04):317–329, 1995. (cited on page 7)

40. G. Brinkmann, B. D. McKay, and U. von Nathusius. Backtrack search and look-ahead for the construction of planar cubic graphs with restricted face sizes. *MATCH Commun. Math. Comput. Chem*, 48:163–177, 2003. (cited on page 9)

41. G. Brinkmann and E. Steffen. Chromatic-index-critical graphs of orders 11 and 12. *European Journal of Combinatorics*, 19(8):889–900, 1998. (cited on page 7)

42. Gunnar Brinkmann, Jan Goedgebeur, and Brendan D McKay. Generation of cubic graphs. *Discrete Mathematics and Theoretical Computer Science*, 13(2):69–79, 2011. (cited on pages 2, 7, and 9)

43. A. E. Brouwer, J. H. Koolen, and R. J. Riebeek. A new distance-regular graph associated to the Mathieu group $M_{\{10\}}$. *Journal of Algebraic Combinatorics*, 8(2):153–156, 1998. (cited on page 3)

44. W. Brown. On graphs that do not contain a Thomsen graph. *Canad. Math. Bull.*, 9:281–289, 1966. (cited on pages 72, 73, and 76)

45. J. Brujic. Experimental study of stress transmission through particulate matter. *PhD Thesis, Gonville and Caius College, Cambridge: University of Cambridge*, 2004. (cited on page 3)

46. J. Brujic, G. Marty, C. Song, C. Briscoe, and H. A. Makse. Fluorescent contacts measure the coordination number and entropy of a 3d jammed emulsion packing. *arXiv preprint cond-mat/0605752*, 2006. (cited on page 3)

47. F. C. Bussemake, S. Čobeljić, D. M. Cvetković, and J. J. Seidel. Cubic graphs on ≤ 14 vertices. *Journal of Combinatorial Theory, Series B*, 23(2):234–235, 1977. (cited on pages 2 and 9)

48. F. C. Bussemaker, S. Cobeljic, D. M. Cvetkovic, and J. J. Seidel. Computer investigation of cubic graphs. *Technical Report T.H. Report 76-WSK-01, Department of Math. Technological University Eindhoven, The Netherlands*, 1976. (cited on pages 2 and 9)

49. F. C. Bussemaker and J. J. Seidel. Cubical graphs of order $2n \leq 10$. *Technische Hogeschool Eindhoven, Notitie*, (10), 1968. (cited on pages 2 and 9)

50. F. Calegari, S. Morrison, and N. Snyder. Cyclotomic integers, fusion categories, and subfactors. *Communications in mathematical physics*, 303(3):845–896, 2011. (cited on pages 24 and 25)

51. H. G. Carstens and E. Steffen. Construction of regular graphs preprintreihe. *Diskrete Strukturen in der Mathematik*, pages 93–114, 1993. (cited on page 9)

52. A. Ceselli, E. Damiani, Sabrina De C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Modeling and assessing inference exposure in encrypted databases. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):119–152, 2005. (cited on page 3)

53. C. Clapham, A. Flockhart, and J. Sheehan. Graphs without four-cycles. *J. Graph Theory*, 13(1):29–47, 1989. (cited on pages xvii, 71, and 73)

54. C. J. Colbourn and R. C. Read. Orderly algorithms for generating restricted classes of graphs. *Journal of Graph Theory*, 3(2):187–195, 1979. (cited on page 6)

55. C. J. Colbourn and R. C. Read. Orderly algorithms for graph generation. *International Journal of Computer Mathematics*, 7(3):167–172, 1979. (cited on page 6)

56. J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 183–189. ACM, 2004. (cited on page 3)

57. D. Crnković and V. Mikulić. Block designs and strongly regular graphs constructed from the group $U(3,4)$. *Glasnik matematički*, 41(2):189–194, 2006. (cited on page 3)

58. G. Damásdi, T. Héger, and T. Szőnyi. The Zarankiewicz problem, cages, and geometries. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Mathematica*, 56(1):3–37, 2013. (cited on pages xvii, 79, and 127)

59. J. de Vries. Over vlakke configuraties waarin elk punt met twee lijnen incident is. *Verslagen en Mededeelingen der Koninklijke Akademie voor Wetenschappen, Afdeeling Natuurkunde (3)*, 6:382–407, 1889. (cited on pages 2 and 9)

60. J. de Vries and V. Martinetti. Sur les configurations planes dont chaque point supporte deux droites. *Rendiconti del Circolo Matematico di Palermo (1884-1940)*, 5(1):221–226, 1891. (cited on pages 2 and 9)

61. O. Delgado-Friedrichs and M. O'Keeffe. Isohedral simple tilings: binodal and by tiles with $\leq$ 16 faces. *Acta Crystallographica Section A: Foundations of Crystallography*, 61(3):358–362, 2005. (cited on page 2)

62. M. Deza, M. Dutour, and P. W. Fowler. Zigzags, railroads, and knots in fullerenes. *Journal of chemical information and computer sciences*, 44(4):1282–1293, 2004. (cited on page 2)

63. G. Ding and P. Chen. Generating r-regular graphs. *Discrete applied mathematics*, 129(2):329–343, 2003. (cited on page 9)

64. J. H. Dinitz, D. K. Garnick, and B. D. McKay. There are 526,915,620 nonisomorphic one-factorizations of K_{12}. *Journal of Combinatorial Designs*, 2(4):273–285, 1994. (cited on page 6)

65. F. M. Dong and K. M. Koh. The sizes of graphs with small girth. *Bulletin of the ICA*, 18:33–44, 1996. (cited on pages 74 and 75)

66. R. D. Dutton and R. C. Brigham. Edges in graphs with large girth. *Graphs and Combinatorics*, 7(4):315–321, 1991. (cited on pages 74 and 77)

67. J. Dybizbański, T. Dzido, and S. P. Radziszowski. On some Zarankiewicz numbers and bipartite Ramsey numbers for quadrilateral. *arXiv preprint arXiv:1303.5475*, 2013. (cited on pages xvii, 79, and 127)

68. T. Dzido. A note on Turán numbers for even wheels. *Graphs Combin.*, 29(5):1305–1309, 2013. (cited on page 77)

69. P. Erdős. Extremal problems in graph theory. In *IN "THEORY OF GRAPHS AND ITS APPLICATIONS," PROC. SYMPOS. SMOLENICE*. Citeseer, 1964. (cited on page 76)

70. P. Erdős. Some recent results on extremal problems in graph theory. *Results, Theory of Graphs (Internat. Sympos., Rome, 1966), Gordon and Breach, New York*, pages 117–123, 1967. (cited on page 72)

71. P. Erdős. Some recent progress on extremal problems in graph theory. *Congr. Numer.*, 14:3–4, 1975. (cited on page 74)

72. P. Erdős. Problems and results in combinatorial analysis. *Creation in Mathematics*, 9:25, 1976. (cited on page 72)

73. P. Erdős, A. Rényi, and V. Sós. On a problem in the theory of graphs. *Publ. Math. Inst. Hungar. Acad. Sci*, 7:215–235, 1962. (cited on page 72)

74. P. Erdős, A. Rényi, and V. Sós. On a problem of graph theory. *Studia Sci. Math. Hungar*, 1:215–235, 1966. (cited on pages 72 and 73)

75. P. Erdős and M. Simonovits. On a valence problem in extremal graph theory. *Discrete Math.*, 5(4):323–334, 1973. (cited on page 78)

76. P. Erdős and M. Simonovits. Compactness results in extremal graph theory. *Combinatorica*, 2(3):275–288, 1982. (cited on pages 76 and 78)

77. P. Erdős and M. Simonovits. Cube-saturated graphs and related problems. *Academic Press, Toronto, Progress in graph theory*, pages 203–218, 1984. (cited on page 72)

78. I. A. Faradzev. Constructive enumeration of combinatorial objects. *Problemes Combinatoires et Theorie des Graphes Colloque Internat. CNRS 260. CNRS Paris*, 1978. (cited on page 6)

79. I. A. Faradzhev. Generation of non-isomorphic graphs with a given distribution of the degree of vertices. *Algorithmic Studies in Combinatorics*, 562:11–19, 1976. (cited on pages 6, 20, 137, and 148)

80. R. J. Faudree and B. D. McKay. A conjecture of Erdős and the Ramsey number $r(W6)$. *J. Combinatorial Math. and Combinatorial Comput.*, 13:23–31, 1993. (cited on page 7)

81. F. Fiedler, M. Klin, and M. H. Muzychuk. Small vertex-transitive directed strongly regular graphs. *Discrete mathematics*, 255(1):87–115, 2002. (cited on page 3)

82. I. S. Filotti and J. N. Mayer. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 236–243. ACM, 1980. (cited on page 19)

83. F. Firke, P. Kosek, E. Nash, and J. Williford. Extremal graphs without 4-cycles. *J. Combin. Theory Ser. B*, 103(3):327–336, 2013. (cited on page 73)

84. Z. Füredi. Graphs without quadrilaterals. *J. Combin. Theory Ser. B*, 34(2):187–190, 1983. (cited on pages 72 and 73)

85. Z. Füredi. Quadrilateral-free graphs with maximum number of edges. *submitted for publication*, 1988. (cited on pages 72 and 73)

86. Z. Füredi. On the number of edges of quadrilateral-free graphs. *J. Combin. Theory Ser. B*, 68(1):1–6, 1996. (cited on pages 72 and 73)

87. Z. Füredi and D. Gunderson. Extremal numbers for odd cycles. *Combin. Probab. Comput.*, 24(04):641–645, 2015. (cited on page 77)

88. Z. Füredi, A. Naor, and J. Verstraëte. On the Turán number for the hexagon. *Advances in Mathematics*, 203(2):476–496, 2006. (cited on page 77)

89. Z. Füredi and M. Simonovits. The history of degenerate (bipartite) extremal graph problems. *arXiv:1306.5167 / Erd?s Centennial, 2013 - Springer*, 2013. (cited on pages 70, 71, and 72)

90. M. Furer. *A counterexample in graph isomorphism testing*. Pennsylvania State University, Department of Computer Science, 1987. (cited on pages 20 and 135)

91. D. K. Garnick, Y. H. H. Kwong, and F. Lazebnik. Algorithmic search for extremal graphs of girth at least five. *Report, Bowdoin College, Brunswick, ME*, 1992. (cited on pages xvii and 74)

92. D. K. Garnick, Y. H. H. Kwong, and F. Lazebnik. Extremal graphs without three-cycles or four-cycles. *J. Graph Theory*, 17(5):633–645, 1993. (cited on pages xvii and 74)

93. D. K. Garnick and N. A. Nieuwejaar. Non-isomorphic extremal graphs without three-cycles or four-cycles. *J. Combin. Math. Combin. Comput.*, 12:33–56, 1992. (cited on pages xvii and 74)

94. W. Goddard, M. A. Henning, and O. R. Oellermann. Bipartite Ramsey numbers and Zarankiewicz numbers. *Discrete Math.*, 219(1):85–95, 2000. (cited on pages xvii, 78, 79, and 127)

95. L. A. Goldberg. Efficient algorithms for listing unlabeled graphs. *Journal of Algorithms*, 13(1):128–143, 1992. (cited on pages 3 and 6)

96. F. Goodman, P. de la Harpe, and V. F. R. Jones. Coxeter graphs and towers of algebras,. *Mathematical Sciences Research Institute Publications 14*, 1989. (cited on page 24)

97. R. Grund. Konstruktion schlichter graphen mit gegebener gradpartition. *Bayreuther Mathematische Schriften*, 44:73–104, 1993. (cited on page 2)

98. R. Grund, A. Kerber, and R. Laue. Construction of discrete structures, especially of chemical isomers. *Discrete Applied Mathematics*. (cited on page 2)

99. R. Grund, A. Kerber, and R. Laue. Molgen, ein computeralgebra-system für die konstruktion molekularer graphen. *Communications in mathematical chemistry(Match)*, 27:87–131, 1992. (cited on page 2)

100. T. Grüner, R. Laue, and M. Meringer. Algorithms for group actions applied to graph generation. In *Groups and Computation II*, volume 68, page 113. American Mathematical Soc., 1997. (cited on page 2)

101. R. K. Guy. The many-facetted problem of Zarankiewicz. *Lecture Notes in Maths (The Many Facets of Graphs Theory)*, 110:129–148, 1969. (cited on page 127)

102. U. Haagerup. Principal graphs of subfactors in the index range $4 < [m : N] < 3 + \sqrt{2}$. *In: Subfactors (Kyuzeso, 1993). MR1317352. World Sci. Publ., River Edge, NJ*, page 1–38, 1994. (cited on page 24)

103. S. G. Hartke and A. J. Radcliffe. Mckay's canonical graph labeling algorithm. *Communicating mathematics*, 479:99–111, 2009. (cited on pages 20 and 134)

104. J. Heitzig and J. Reinhold. The number of unlabeled orders on fourteen elements. *Order*, 17(4):333–341, 2000. (cited on page 7)

105. J. Hu, A. H. MacDonald, and B. D. McKay. Correlations in two-dimensional vortex liquids. *Physical Review B*, 49(21):15263, 1994. (cited on page 7)

106. Y. Hu, H. Chen, Y. Zhu, A. Chien, and C. Cheng. Physical synthesis of energy-efficient networks-on-chip through topology exploration and wire style optimization. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 111–118. IEEE, 2005. (cited on page 3)

107. C. Igel and P. Stagge. Effects of phenotypic redundancy in structure optimization. *Evolutionary Computation, IEEE Transactions on*, 6(1):74–85, 2002. (cited on page 3)

108. C. Igel and P. Stagge. Graph isomorphisms effect on structure optimization of neural networks. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, volume 1, pages 142–147. IEEE, 2002. (cited on page 3)

109. W. Imrich. Zehnpunktige kubische graphen. *aequationes mathematicae*, 6(1):6–10, 1971. (cited on pages 2 and 9)

110. M. Izumi. Application of fusion rules to classification of subfactors. *Publications of the Research Institute for Mathematical Sciences*, 27(6):953–994, 1991. (cited on page 24)

111. M. Izumi. On flatness of the Coxeter graph E_8. *Pacific J. Math. vol. 166 (2), MR1313457 euclid.pjm/1102621140,*, page 305–327, 1994. (cited on page 24)

112. M. Izumi, V. F. R. Jones, S. Morrison, and N. Snyder. Subfactors of index less than 5, part 3: quadruple points. *Communications in Mathematical Physics*, 316(2):531–554, 2012. (cited on page 24)

113. M. Izumi and Y. Kawahigashi. Classification of subfactors with the principal graph $D_n^{(1)}$. *Journal of functional analysis*, 112(2):257–286, 1993. (cited on page 24)

114. M. Izumi, S. Morrison, D. Penneys, E. Peters, and N. Snyder. Subfactors of index exactly 5. *Bulletin of the London Mathematical Society*, page bdu113, 2015. (cited on page 24)

115. D. Jackson, S. Jha, and C. A. Damon. Isomorph-free model enumeration: a new method for checking relational specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(2):302–343, 1998. (cited on page 7)

116. V. F. R. Jones. Subfactors of type II_1 factors and related topics. In *International Congress of Mathematicians*, page 939, 1986. (cited on page 24)

117. V. F. R. Jones. The annular structure of subfactors. *Enseign. Math,"Essays on geometry and related topics, Vol. 1, 2", Monogr*, 38:401–463, 2001. (cited on page 24)

118. V. F. R. Jones. Quadratic tangles in planar algebras. *Duke Mathematical Journal*, 161(12):2257–2295, 2012. (cited on page 24)

119. V. F. R. Jones, S. Morrison, and N. Snyder. The classification of subfactors of index at most 5. *Bulletin of the American Mathematical Society*, 51(2):277–327, 2014. (cited on page 24)

120. Y. Kawahigashi. On flatness of Ocneanu's connections on the Dynkin diagrams and classification of subfactors. *Journal of Functional Analysis*, 127(1):63–107, 1995. (cited on page 24)

121. P. Keevash, B. Sudakov, and J. Verstraëte. On a conjecture of Erdős and Simonovits: Even cycles. *Combinatorica*, 33(6):699–732, 2013. (cited on page 78)

122. J. Kobler, U. Schöning, and J. Torán. *The graph isomorphism problem: its structural complexity*. Springer Science & Business Media, 2012. (cited on page 19)

123. T. Kovari, V. Sos, and P. Turán. On a problem of K. Zarankiewicz. In *Colloq. Math.*, volume 3, pages 50–57, 1954. (cited on page 72)

124. D. L. Kreher and D. R. Stinson. *Combinatorial algorithms: generation, enumeration, and search*, volume 7. CRC press, 1998. (cited on pages 1, 5, 40, and 43)

125. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2006. (cited on pages 3 and 6)

126. D. Kühn and D. Osthus. Four-cycles in graphs without a given even cycle. *J. Graph Theory*, 48(2):147–156, 2005. (cited on page 77)

127. V. Kvasnicka and J. Pospichal. Canonical indexing and constructive enumeration of molecular graphs. *Journal of chemical information and computer sciences*, 30(2):99–105, 1990. (cited on page 2)

128. T. Lam. Graphs without cycles of even length. *Bull. Aust. Math. Soc.*, 63(3):435–440, 2001. (cited on page 79)

129. T. Lam. A result on 2k-cycle-free bipartite graphs. *Australas. J. Combin.*, 32:163–170, 2005. (cited on page 79)

130. F. Lazebnik and V. A. Ustimenko. New examples of graphs without small cycles and of large size. *European J. Combin.*, 14(5):445–460, 1993. (cited on page 78)

131. F. Lazebnik and V. A. Ustimenko. Explicit construction of graphs with an arbitrary large girth and of large size. *Discrete Appl. Math.*, 60(1):275–284, 1995. (cited on page 78)

132. F. Lazebnik, V. A. Ustimenko, and A. Woldar. Properties of certain families of 2k-cycle-free graphs. *J. Combin. Theory Ser. B*, 60(2):293–298, 1994. (cited on page 78)

133. F. Lazebnik, V. A. Ustimenko, and A. J. Woldar. A new series of dense graphs of high girth. *Bulletin of the AMS*, 1995. (cited on page 78)

134. F. Lazebnik, V. A. Ustimenko, and A. J. Woldar. Polarities and 2k-cycle-free graphs. *Discrete Math.*, 197:503–513, 1999. (cited on pages 76 and 77)

135. F. Lazebnik and P. Wang. On the structure of extremal graphs of high girth. *J. Graph Theory*, 26(3):147–153, 1997. (cited on page 77)

136. J. Lemeire, B. Smets, P. Cara, and E. Dirkx. Exploiting symmetry for partitioning models in parallel discrete event simulation. In *Proceedings of the eighteenth workshop on Parallel and distributed simulation*, pages 189–194. ACM, 2004. (cited on page 3)

137. R. K. Lindsay, B. G. Buchanan, E. A. Feigenbaum, and J. Lederberg. Applications of artificial intelligence for organic chemistry: The dendral project. *McGraw-Hill, New York*, 1980. (cited on page 2)

138. Z. Liu. Composed inclusions of A_3 and A_4 subfactors". *Adv. Math. vol. 279 (2015). arXiv:1308.5691 MR3345186 DOI:10.1016/j.aim.2015.03.017, pp. 307–371.* (cited on page 24)

139. Z. Liu, S. Morrison, and D. Penneys. 1-supertransitive subfactors with index at most $6\frac{1}{5}$. *Communications in Mathematical Physics*, 334(2):889–922, 2013. (cited on page 24)

140. E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 42–49. IEEE, 1980. (cited on page 19)

141. W. Mantel. Problem 28. *Wiskundige Opgaven*, 10(60-61):320, 1907. (cited on page 69)

142. F. Margot. Exploiting orbits in symmetric ILP. *Mathematical Programming*, 98(1-3):3–21, 2003. (cited on page 3)

143. R. Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–136, 1979. (cited on page 19)

144. B. D. McKay. *Practical graph isomorphism*. Department of Computer Science, Vanderbilt University, 1981. (cited on pages 20, 121, and 134)

145. B. D. McKay. Geng. *A program for generating graphs available at http://cs. anu. edu. au/~ bdm/nauty*, 1984. (cited on pages 2 and 7)

146. B. D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26(2):306–324, 1998. (cited on pages 7, 8, 9, 10, 25, 60, 61, 63, 91, 92, 94, and 148)

147. B. D. McKay. Nauty user's guide (version 2.4). *Computer Science Dept., Australian National University*, pages 225–239, 2007. (cited on pages 3, 20, 27, 40, and 102)

148. B. D. McKay, N. D. Megill, and M. Pavičic. Algorithms for Greechie diagrams. *International Journal of Theoretical Physics*, 39(10):2381–2406, 2000. (cited on page 2)

149. B. D. McKay and Z. k. Min. The value of the Ramsey number $R(3,8)$. *Journal of Graph Theory*, 16(1):99–105, 1992. (cited on page 7)

150. B. D. McKay and A. Piperno. *nauty and Traces* software program, version 2.5. *http://cs.anu.edu.au/ bdm/nauty/*. (cited on page 20)

151. B. D. McKay and A. Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014. (cited on page 20)

152. B. D. McKay and S. Radziszowski. 2-(22, 8, 4) designs have no blocks of type 3. *The Charles Babbage Research Centre: The Journal of Combinatorial Mathematics and Combinatorial Computing*, 30, 1999. (cited on page 7)

153. B. D. McKay and S. P. Radziszowski. The first classical Ramsey number for hypergraphs is computed. In *SODA*, pages 304–308, 1991. (cited on page 7)

154. B. D. McKay and S. P. Radziszowski. A new upper bound for the Ramsey number $R(5,5)$. *Centre for Discrete Mathematics and Computing: Australasian Journal of Combinatorics*, 5, 1992. (cited on page 7)

155. B. D. McKay and S. P. Radziszowski. $R(4,5) = 25$. *Journal of Graph Theory*, 19(3):309–322, 1995. (cited on page 7)

156. B. D. McKay and S. P. Radziszowski. The nonexistence of 4-$(12,6,6)$ designs. In *Computational and Constructive Design Theory*, pages 177–188. Springer, 1996. (cited on page 7)

157. B. D. McKay and S. P. Radziszowski. Towards deciding the existance of 2-(22, 8, 4) designs. *The Journal of Combinatorial Mathematics and Combinatorial Computing*, 22:211–222, 1996. (cited on page 7)

158. B. D. McKay and S. P. Radziszowski. Subgraph counting identities and Ramsey numbers. *journal of combinatorial theory, Series B*, 69(2):193–209, 1997. (cited on page 7)

159. B. D. McKay and E. Rogoyski. Latin squares of order 10. *Electron. J. Combin*, 2:N3, 1995. (cited on page 3)

160. B. D. McKay and G. F. Royle. Constructing the cubic graphs on up to 20 vertices. In *Ars Combinatoria*, pages 129–140. 21A, 1986. (cited on pages 2, 7, and 9)

161. C. Menon. isomorphism-free generation of quartic graphs. *Honours Essay, Australian National University, advisor: B.D. McKay*, 2001. (cited on page 9)

162. M. Meringer. *GENREG* software program. *http://www.mathe2.uni-bayreuth.de/markus/reggraphs.html*. (cited on pages 9 and 15)

163. M. Meringer. Erzeugung regulärer graphen. *Universität Bayreuth*, 1996. (cited on pages 6 and 9)

164. M. Meringer. Fast generation of regular graphs and construction of cages. *Journal of Graph Theory*, 30(2):137–146, 1999. (cited on pages 9 and 15)

165. G. Miller. Isomorphism testing for graphs of bounded genus. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 225–235. ACM, 1980. (cited on page 19)

166. R. V. Mises and H. Pollaczek-Geiringer. Praktische verfahren der gleichungsauflösung. *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, 9(1):58–77, 1929. (cited on page 52)

167. T. Miyazaki. The complexity of McKay's canonical labeling algorithm. In *Groups and Computation II*, volume 28, pages 239–256. Aer. Math. Soc.: Providence, RI, 1997. (cited on pages 20 and 135)

168. S. Morrison, D. Penneys, E. Peters, and N. Snyder. Subfactors of index less than 5, part 2: Triple points. *International Journal of Mathematics*, 23(03):1250016, 2012. (cited on page 24)

169. S. Morrison and E. Peters. The little desert? some subfactors with index in the interval $(5, 3+\sqrt{5})$. *International Journal of Mathematics*, 25(08):1450080, 2014. (cited on page 24)

170. S. Morrison and N. Snyder. Subfactors of index less than 5, part 1: the principal graph odometer. *Communications in Mathematical Physics*, 312(1):1–35, 2012. (cited on page 24)

171. A. Ocneanu. Quantized groups, string algebras and Galois theory for algebras. *Operator algebras and applications*, 2:119–172, 1988. (cited on page 24)

172. T. Ohl. Electroweak gauge bosons at future electron-positron colliders. *arXiv preprint hep-ph/9911437*, 1999. (cited on page 3)

173. P. R. J. Östergård, T. Baicheva, and E. Kolev. Optimal binary one-error-correcting codes of length 10 have 72 codewords. *Information Theory, IEEE Transactions on*, 45(4):1229–1231, 1999. (cited on page 3)

174. J. Ostrowski, J. Linderoth, F. Rossi, and S. Smriglio. Orbital branching. *Mathematical Programming*, 126(1):147–178, 2011. (cited on page 3)

175. D. Penneys. Chirality and principal graph obstructions. *Advances in Mathematics*, 273:32–55, 2015. (cited on page 24)

176. D. Penneys and J. E. Tener. Subfactors of index less than 5, part 4: vines. *International Journal of Mathematics*, 23(03):1250017, 2012. (cited on page 24)

177. A. N. Petrenjuk and L. P. Petrenjuk. On constructive enumeration of 12 vertex cubic graphs. *Combinatorial analysis*, (3), 1974. (cited on pages 2 and 9)

178. A. Piperno. Search space contraction in canonical labeling of graphs. *arXiv preprint arXiv:0804.4881*, 2008. (cited on page 20)

179. S. Popa. Classification of amenable subfactors of type II. *Acta Mathematica*, 172(2):163–255, 1994. (cited on page 24)

180. A. Ramani and I. L. Markov. Automatically exploiting symmetries in constraint programming. In *Recent Advances in Constraints*, pages 98–112. Springer, 2005. (cited on page 3)

181. A. Rassat, P. W. Fowler, and B. de La Vaissiere. Cahn–ingold–prelog descriptors of absolute configuration for carbon cages. *Chemistry-A European Journal*, 7(18):3985–3991, 2001. (cited on page 2)

182. R. C. Read. Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations. *Annals of Discrete Mathematics*, 2:107–120, 1978. (cited on pages 6, 20, 137, and 148)

183. I. Reiman. Uber ein problem von K. Zarankiewicz. *Acta Math. Hungar.*, 9(3):269–273, 1958. (cited on pages 72 and 76)

184. G. F. Royle. An orderly algorithm and some applications in finite geometry. *Discrete Mathematics*, 185(1):105–115, 1998. (cited on page 7)

185. F. Ruskey. *Combinatorial generation*, volume 11. 2003. (cited on pages 1 and 5)

186. S. Sanjmyatav. *Algorithms for generation of cubic graphs*. Master's thesis, Australian National University, advisor: B.D. McKay., 2000. (cited on pages 2, 7, 9, and 17)

187. Z. Shao, J. Xu, and X. Xu. A new Turán number for quadrilateral. *Util. Math.*, 79:51–58, 2009. (cited on pages xvii, 72, and 73)

188. M. Simonovits. Extremal graph theory. *Selected topics in graph theory*, 2:161–200, 1983. (cited on pages 76 and 78)

189. N. Snyder. A rotational approach to triple point obstructions. *Analysis & PDE*, 6(8):1923–1928, 2014. (cited on page 24)

190. L. H. Soicher. Is there a Mclaughlin geometry? *Journal of Algebra*, 300(1):248–255, 2006. (cited on page 3)

191. M. Tait and C. Timmons. Sidon sets and graphs without 4-cycles. *arXiv preprint arXiv:1309.6350*, 2013. (cited on page 73)

192. J. Tang, Y. Lin, C. Balbuena, and M. Miller. Calculating the extremal number $ex(v; \{c_3, c_4, \ldots, c_n\})$. *Discrete Appl. Math.*, 157(9):2198–2206, 2009. (cited on page 75)

193. J. Tits. Sur la trialité et certains groupes qui s' en déduisent. *Publications Mathématiques de l'IHÉS*, 2(1):14–60, 1959. (cited on page 78)

194. S. Toida. Construction of quartic graphs. *Journal of Combinatorial Theory, Series B*, 16(2):124–133, 1974. (cited on page 9)

195. P. Turán. Eine extremalaufgabe aus der graphentheorie. *Mat. Fiz. Lapok*, 48(436-452):61, 1941. (cited on page 69)

196. P. Turán. On the theory of graphs. In *Colloq. Math.*, volume 1, pages 19–30, 1954. (cited on page 69)

197. V. A. Ustimenko and A. J. Woldar. An improvement on the Erdős bound for graphs of girth 16. *Contemp. Math.*, 184:419–419, 1995. (cited on page 78)

198. L. G. Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979. (cited on page 19)

199. D. Van Dyck, G. Brinkmann, V. Fack, and B. D. McKay. To be or not to be Yutsis: algorithms for the decision problem. *Computer physics communications*, 173(1):61–70, 2005. (cited on page 2)

200. R. L. C. Vink. Computer simulations of amorphous semiconductors. *PhD Thesis, Universiteit Utrecht*, 2004. (cited on page 3)

201. R. L. C. Vink and G. T. Barkema. Configurational entropy of network-forming materials. *Physical review letters*, 89(7):076405, 2002. (cited on page 3)

202. P. Wang, G. W. Dueck, and S. MacMillan. Using simulated annealing to construct extremal graphs. *Discrete Math.*, 235(1):125–136, 2001. (cited on page 74)

203. R. Wenger. Extremal graphs with no C^4's, C^6's, or C^{10}'s. *J. Combin. Theory Ser. B*, 52(1):113–116, 1991. (cited on pages 76 and 78)

204. A. J. Woldar and V. A. Ustimenko. An application of group theory to extremal graph theory. In *Group theory, Proceedings of the Ohio State-Denison Conference*, pages 293–298, 1993. (cited on page 78)

205. D. Woodall. Sufficient conditions for circuits in graphs. *Proceedings of the London Mathematical Society*, 3(4):739–755, 1972. (cited on page 77)

206. Y. Wu, Y. Sun, and S. Radziszowski. Wheel and star-critical Ramsey numbers for quadrilateral. *Discrete Math. Appl.*, 186:260–271, 2015. (cited on page 72)

207. Y. Wu, Y. Sun, R. Zhang, and S. Radziszowski. Ramsey numbers of C_4 versus wheels and stars. (cited on page 73)

208. Y. Yang, X. Lin, G. Dong, and Y. Zhao. Extremal graphs without three-cycles, four-cycles or five-cycles. *Util. Math.*, 66:249–265, 2004. (cited on pages xvii, 74, and 75)

209. S. Yongqi, L. Xiaohui, Y. Yuansheng, and S. Lei. Extremal graphs without four-cycles or five-cycles. *Util. Math.*, 80:115–130, 2009. (cited on pages xvii and 74)

210. Y. Yuansheng and P. Rowlinson. On extremal graphs without four-cycles. *Util. Math.*, 41:204–210, 1992. (cited on pages xvii, 72, and 73)

211. Y. Yuansheng and P. Rowlinson. On graphs without 6-cycles and related Ramsey numbers. 1992. (cited on page 77)

Index