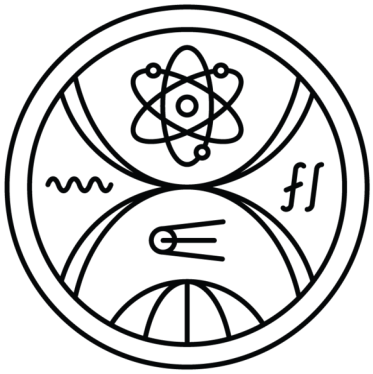
COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

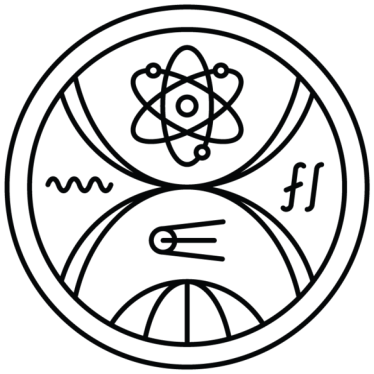


SYMMETRIES OF COMBINATORIAL STRUCTURES

Diploma Thesis

2023 Bc. Matúš Gál

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS



SYMMETRIES OF COMBINATORIAL STRUCTURES

Diploma Thesis

Study Programme:	Applied Informatics
Field of Study:	2508 Informatics
Department:	Department of Applied Informatics
Supervisor:	doc. RNDr. Tatiana Jajcayová, PhD.

Bratislava, 2023

Bc. Matúš Gál

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Matúš Gál
Study programme:	Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Symmetries of combinatorial structures

Annotation: The aim of the theses is to study algebraic tools (groups, inverse semigroups,...)to describe symmetries of several combinatorial structures (like graphs, hypergraphs, designs, ...). The work will involve generating these underlying algebraic structures by computer, resp. computer searching through big families of these structures for certain given properties.

Supervisor:	doc. RNDr. Tatiana Jajcayová, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.

Assigned: 01.10.2020

Approved: 08.10.2020 prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Hereby I declare I worked on this thesis on my own, using the sources listed in the bibliography and under the supervision of my thesis supervisor.

.....

Bratislava, 2023 Bc. Matúš Gál

Acknowledgements

Firstly, I would like to thank my thesis supervisor doc. RNDr. Tatiana Jajcayová, PhD., for her help in making complex topics easy to understand and providing feedback.

I also want to thank everyone who mentally supported me and kept me going throughout the writing of this thesis.

Abstract

The aim of this thesis is to study both the symmetries and partial symmetries of graphs and explore them using algebraic tools from group theory and inverse semigroup theory.

When studying symmetries of graphs using groups, one quickly finds this approach is insufficient, since almost all graphs are asymmetric, meaning they have no non-trivial symmetries. For this reason, we need to use tools from inverse semigroup theory, such as inverse monoids and Green's relations, to examine partial symmetries of graphs, specifically the symmetries between their vertex-induced subgraphs.

While our primary goal was studying the partial symmetries of minimal asymmetric graphs, our solution works with any graph. To find the inverse automorphism monoids of graphs we implemented an application in Python, which provides an interface for working with graphs, their full and partial automorphisms. With our algorithm, we achieved significantly better performance in comparison to other currently existing solutions.

Finally, we introduced the definition of asymmetric depth, used to measure the asymmetricity of graphs.

Keywords: graph theory, group theory, inverse semigroup theory, automorphism, partial automorphism

Abstrakt

Cieľom tejto diplomovej práce bolo štúdium symetrií a čiastočných symetrií grafov a ich skúmanie použitím algebraických nástrojov z teórie grúp a teórie inverzných pologrúp.

Pri štúdiu grafov použitím grúp zistíme, že tento prístup je neadekvátny vo väčšine prípadov, keďže takmer všetky grafy sú asymetrické, teda nemajú netriviálne symetrie. Z toho dôvodu potrebujeme použiť nástroje z teórie inverzných pologrúp, inverzné monoidy a Greenove relácie, na štúdium symetrií ich indukovaných podgrafov.

Našim primárnym cieľom bolo skúmanie čiastočných symetrií minimálnych asymetrických grafov, avšak naše riešenie je vhodné pre ľubovoľné grafy. Na nájdenie inverzných monoidov automorfizmov grafov sme implementovali aplikáciu v Pythone, ktorá ponúka rozhranie na prácu s grafmi, ich úplnými a čiastočnými symetriami. S našim algoritmom sme dosiahli lepšie výsledky ako ostatné aktuálne existujúce riešenia.

Nakoniec sme predstavili definíciu asymetrickej hĺbky, ktorá môže byť použitá na meranie asymetrickosti grafov.

Kľúčové slová: teória grafov, teória grúp, teória inverzných pologrúp, automorfizmus, čiastočný automorfizmus

Contents

List of Figures

List of Tables

Introduction

Graph theory is a field of mathematics aimed at studying the properties of structures representable by graphs. Graphs, in some use cases referred to as networks, are an integral part of contemporary computer science, with usages in artificial intelligence, machine learning, and data science.

It might be advantageous to analyze the symmetries exhibited by a graph to gain more information about it. Symmetries are rotations, reflections, or permutations of vertices preserving the graph's structure and relationships between its vertices. Historically, algebraic tools, specifically groups and permutations, were used to describe the symmetries of graphs. For graphs with rich symmetries, the approach can provide beneficial information. However, most graphs do not have non-trivial symmetries, rendering this approach unsatisfactory [PE63]. Therefore, different algebraic tools, namely inverse semigroups, have been proposed as an alternative [JJSS21]. Using inverse semigroups also allows us to examine the graph's partial symmetries, meaning the symmetries of its induced subgraphs.

Our thesis aims to study the partial symmetries of graphs, specifically focused on finding the partial symmetries of minimal asymmetric graphs [SS17].

In the first chapter, we introduce concepts useful for our thesis. Firstly, focus on graphs and their utilization in real-world applications and list im-

portant definitions necessary for our work. We provide examples to help better illustrate and understand these concepts. We then explore algebraic tools, groups, and permutation groups used to study symmetries of graphs. Additionally, we extend the concept of symmetries from global to partial and examine algebraic tools used to study partial symmetries.

In the second chapter, we talk about our implementation, the selection of the appropriate programming language for our solution, and list some existing libraries that deal with graphs and their symmetries. We examine the classes we implemented for our application, introduce our algorithm and explain how we implemented it. Afterward, we look at the steps we took to optimize the runtime of our algorithm and provide a comparison between different approaches. Finally, we introduce the web application we implemented to provide a user-friendly interface for working with our application.

In the third chapter, we focused on the results we achieved with our solution and compare its performance to other existing solutions. We introduce the concept of *asymmetric depth* and present our initial research into this idea. We also examine how the structure of graphs relates to the number of symmetries these graphs have.

Chapter 1

Preliminaries

This chapter contains definitions and concepts related to the study of symmetries and partial symmetries of graphs. We also provide examples to illustrate and explain these concepts. Examining the symmetries and partial symmetries of graphs requires knowledge from different fields of mathematics, such as graph theory, group theory, and semigroup theory.

1.1 Graph theory

In this section, we introduce the basics of graph theory. We examine different ways to represent graphs, what subgraphs and induced subgraphs are, and define graph isomorphism and automorphism.

Firstly, we mention some examples and usages of graphs in real-world applications. For example, we can represent the road network as a graph with crossroads or dead ends acting as imaginary vertices and the roads connecting them as edges. Indeed, graphs are used in video game development to represent the road network.

While the term graph is commonly used formally in mathematics, the

graphs appearing in real-world applications are often referred to as networks. Some examples of networks include social networks, used to represent individuals and their relationships, or neural networks, most commonly used in artificial intelligence.

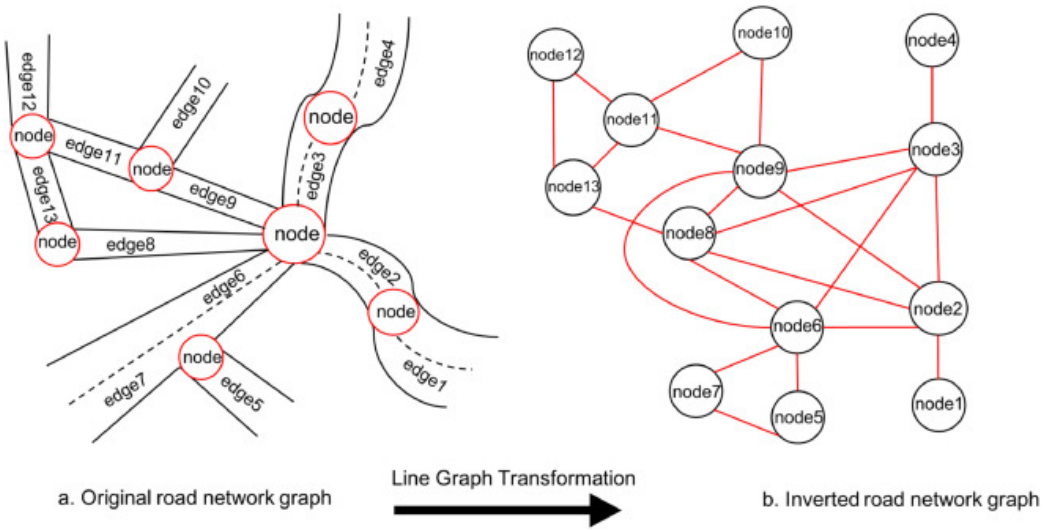


Figure 1.1: Example of a road network represented as a graph [GKSF21].

The definitions used in this section are taken from the book [Wes01].

Definition 1.1 *A graph G is a triple consisting of a vertex set $V(G)$, an edge set $E(G)$, and a relation that associates with each edge two vertices (not necessarily distinct) called its endpoints.*

Let's examine the graph in Fig. 1.2. The vertex set of this graph is $\{a, b, c, d, e\}$ and the edge set is $\{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$. As we can see in the picture, the edge e_1, as well as the edge e_2 have the vertices a and b as its endpoints, the edge e_4 has vertices b and c as its endpoints, and the edge e_7 has the vertex d as both of its endpoints. We also say that an edge is *incident* to its endpoints. We define the following terms to refer to edges whose endpoints are equal and edges that share the same pair of endpoints:

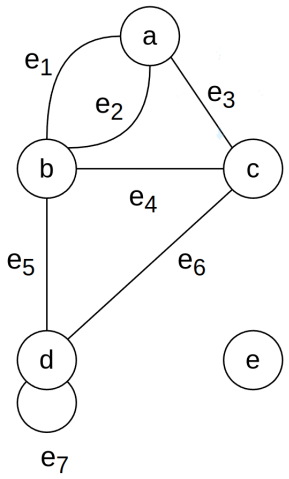


Figure 1.2: An example of a graph with five vertices and seven edges.

Definition 1.2 *A loop is an edge whose endpoints are equal. Multiple edges are edges having the same pair of endpoints.*

By examining the definition of the graph from above, we can see that the relation that associates an edge with two vertices (the adjacency relation) is symmetric, and such graphs are called undirected. When the relation between vertices does not need to be symmetric, we talk about directed graphs, which are defined as follows:

Definition 1.3 *A directed graph or digraph G is a triple consisting of a vertex set V(G), an edge set E(G), and a function assigning each edge an ordered pair of vertices. The first vertex of the ordered pair is the tail of the edge, and the second is the head; together, they are the endpoints. We say that an edge is an edge from its tail to its head.*

When visually representing an edge in a digraph, we draw an arrow near the head vertex. We can see an example of an undirected graph and a digraph in Fig. 1.3.

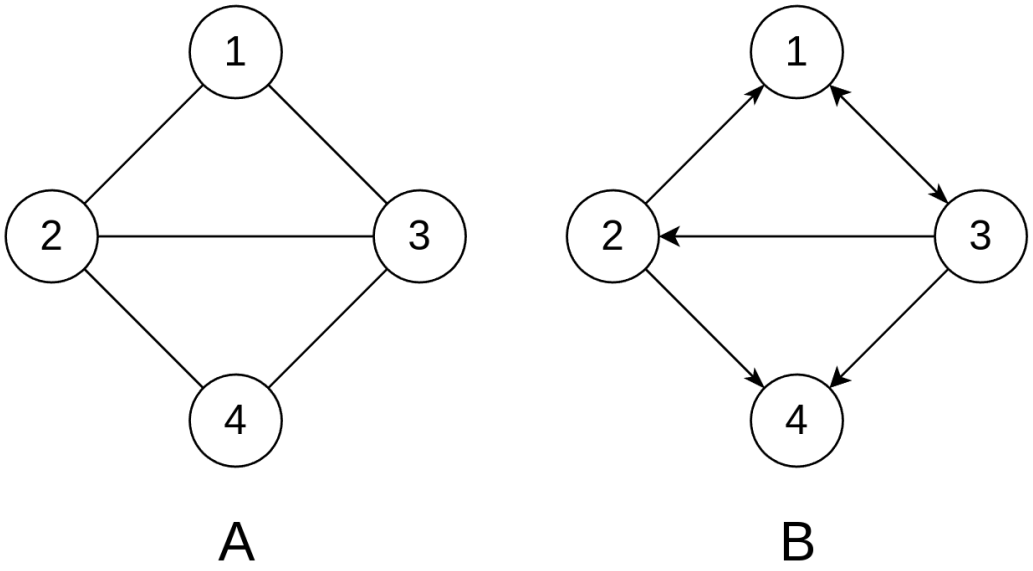


Figure 1.3: An example of an undirected graph (A) and a directed graph (B).

1.1.1 Finite simple graphs

In our work, we focus on a class of graphs called finite simple graphs. A finite simple graph is an undirected graph with a finite vertex set that does not allow loops or multiple edges.

Definition 1.4 *A simple graph is a graph having no loops or multiple edges. We specify a simple graph by its vertex set and edge set, treating the edge set as a set of unordered pairs of vertices and writing $e = uv$ (or $e = vu$) for an edge e with endpoints u and v.*

Definition 1.5 *The complement $\bar{G}$ of a simple graph G is the simple graph with vertex set $V(G)$ defined by $uv \in E(\bar{G})$ if and only if $uv \notin E(G)$. A clique in a graph is a set of pairwise adjacent vertices. An independent set (or stable set) in a graph is a set of pairwise nonadjacent vertices.*

In total, there are $2^{\frac{n(n-1)}{2}}$ possible undirected simple graphs with n vertices. Each vertex can be connected with an edge to $n-1$ different vertices, and since the graph is undirected, we ignore repetitions, hence $\frac{n(n-1)}{2}$. We then have two choices for each pair of vertices: either they are connected by an edge or not.

From this point on, "graph" refers to a finite simple graph unless specified otherwise.

Definition 1.6 *The degree of vertex v in a graph G, written $d_G(v)$ or $d(v)$, is the number of edges incident to v. The maximum degree is $\Delta(G)$, the minimum degree is $\delta(G)$, and G is regular if $\Delta(G) = \delta(G)$.*

The maximum degree of a vertex in graph G on n vertices is $n-1$. The minimum degree of a vertex is 0, and such vertices are called *isolated*. If G is a complete graph, the minimum and maximum degrees are $n-1$. In a k-regular graph, all vertices are of the same degree k.

1.1.2 Subgraphs and induced subgraphs

In this section, we talk about subgraphs and induced subgraphs. We start by listing the definitions of walks, trails, and paths. We then also provide the definitions for subgraphs and components of a graph.

Definition 1.7 *A walk is a list $v_0, e_1, v_1, ..., e_k, v_k$ of vertices and edges such that, for $1 \leq i \leq k$, the edge e_i has endpoints v_{i-1} and v_i. A trail is a walk with no repeated edge. A u, v-walk or u, v-trail has first vertex u and last vertex v; these are its endpoints. A u, v-path is a path whose vertices of degree 1 (its endpoints) are u and v; the others are internal vertices.*

The length of a walk, trail, path, or cycle is its number of edges. A walk or trail is closed if its endpoints are the same.

Definition 1.8 *A subgraph of a graph G is a graph H such that V(H) $\subseteq$ V(G) and E(H) $\subseteq$ E(G) and the assignment of endpoints to edges in H is the same as in G. We then write H $\subseteq$ G and say that "G contains H".*

A graph G is connected if each pair of vertices in G belong to a path; otherwise, G is disconnected.

Definition 1.9 *The components of a graph G are its maximal connected subgraphs.*

Every connected graph has exactly one component. A disconnected n-vertex graph has at least two and at most n components.

Note that in the above definition of subgraph, if $u, v \in V(H)$ and $(u, v) \in E(G)$, then (u, v) does not need to be in $E(H)$. However, we focus on specific types of subgraphs called induced subgraphs.

Definition 1.10 *A cut-edge or cut-vertex of a graph is an edge or vertex whose deletion increases the number of components. We write G - e or G - M for the subgraph of G obtained by deleting an edge e or set of edges M. We write G - v or G - S for the subgraph obtained by deleting a vertex v or set of vertices S. An induced subgraph is a subgraph obtained by deleting a set of vertices, along with their incident edges. We write G[T] for G - $\overline{T}$, where $\overline{T}$ = V(G) - T; this is the subgraph of G induced by T.*

In other words, an induced subgraph H is a subgraph obtained by taking a subset of vertices of graph G and all edges that are incident to these vertices in G.

In Fig. 1.4, we have a graph G. We obtain an induced subgraph $G[T]$, where $T = \{1, 2, 4\}$, by deleting a set of vertices $V(G) - T$ as well as their incident edges. In particular, we delete the vertex 3 and its incident edges, so e_3 and e_4.

A graph with n vertices has at most $\sum_{m=0}^{n} \binom{n}{m} = 2^n$ induced subgraphs.

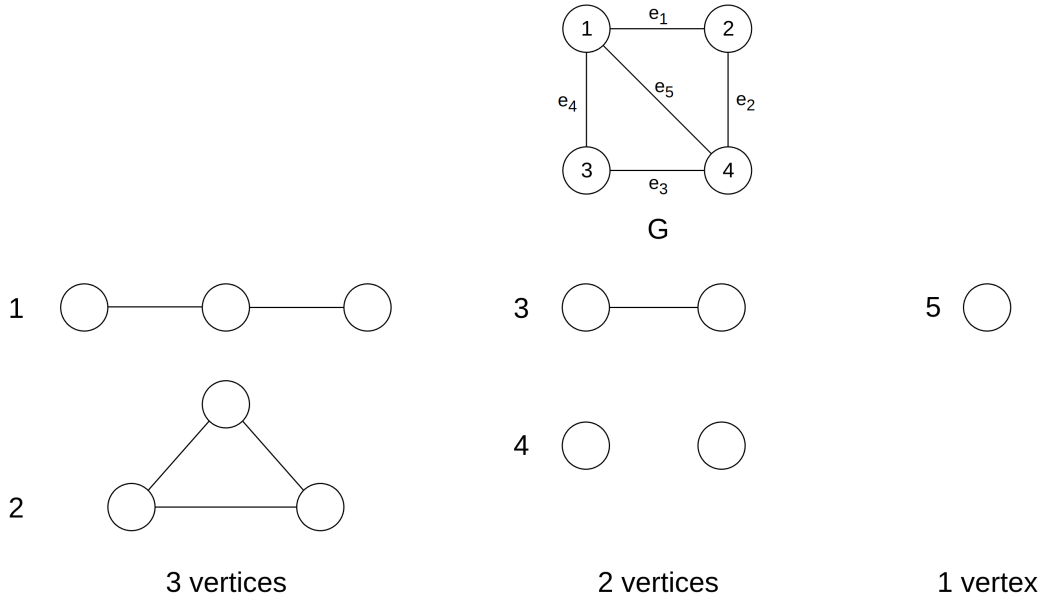


Figure 1.4: A list of all induced subgraphs of a graph G.

For all graphs with $n \geq 2$ vertices, the following statements are always true:

- there is exactly 1 induced subgraph with n vertices, the original graph itself,
- there is an induced subgraph with two vertices and either 0 or 1 edges,
- a single isolated vertex is an induced subgraph,
- the null graph, a graph with an empty vertex set is also an induced subgraph (while this conflicts with our requirement for a graph to have a non-empty set of vertices, this corresponds to an empty partial permutation, the concept of partial permutations is explained in further detail in Section 1.5)

In Fig. 1.4, we can see all induced subgraphs of a graph G with four vertices. The induced subgraph S_1 is a subgraph of G induced by either the set $\{2, 3, 4\}$ or $\{1, 2, 3\}$. The subgraph S_2 is an induced subgraph whose vertex set is either $\{1, 2, 4\}$ or $\{1, 3, 4\}$. Similarly, we can obtain the induced subgraphs S_3, S_4 and S_5.

We will now explore a relationship that exists between induced subgraphs. The subgraph S_3 is obtainable as an induced subgraph from either S_1 or S_2. The subgraph S_4, on the other hand, can be obtained as an induced subgraph only from S_1 and not from S_2.

Generally, for a graph G with n vertices, every k-vertex induced subgraph of $G, k < n$, can be obtained as an induced subgraph from at least one of the induced subgraphs with $k + 1$ vertices.

1.2 Isomorphism and automorphism of graphs

Definition 1.11 *An isomorphism from a simple graph G to a simple graph H is a bijection: V(G) → V(H) such that uv ∈ E(G) if and only if f(u)f(v) ∈ E(H). We say "G is isomorphic to H", written G ≅ H if there is an isomorphism from G to H.*

An example of two isomorphic simple graphs is in Fig. 1.5. The graph G has two vertices with degree 3 (w, z) and two vertices with degree 2 (x, y). Both w and z vertices have three neighbors with degrees 3, 2, and 2, and both vertices x and y have two neighbors with degree 3. In graph H, we have two vertices with degree 3 (c, d) and two vertices with degree 2 (a, b). The vertices with degree 3 have neighbors with degrees 3, 2, and 2, and the vertices with degree 2 have two neighbors with degree 3. We can see that while graphs G and H look different visually, they are indeed isomorphic.

More formally, in our example, there exists a function f that maps the vertices of graph G to the vertices of graph H as follows: $f(x) = b$, $f(y) = a$, $f(z) = d$ and $f(w) = c$. The set of edges of graph H is then $E(H) = \{f(w)f(x), f(w)f(y), f(w)f(z), f(x)f(z), f(y)f(z)\}$.

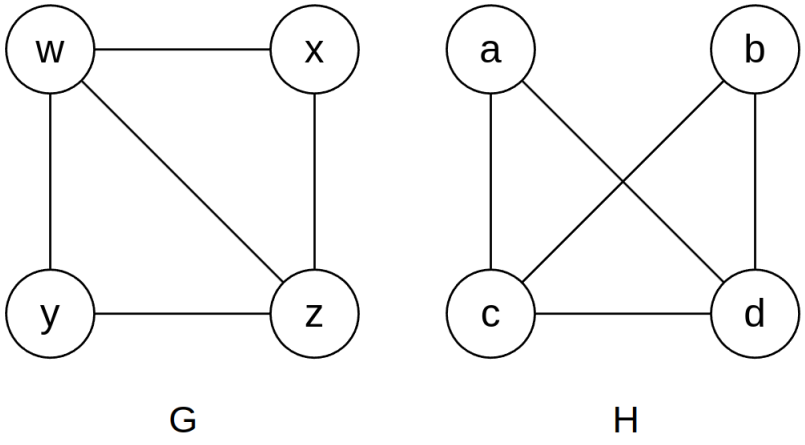


Figure 1.5: An example of two isomorphic graphs.

Even though the graphs G and H have the same degree sequence $(3, 3, 2, 2)$, this information alone is insufficient to determine whether the two graphs are isomorphic. Consider the graphs shown in Fig. 1.6. Both graphs have the same degree sequence (3, 2, 2, 2, 2, 1, 1, 1), yet they are not isomorphic. The only vertex of degree 3 has neighbors with degrees 2, 1, 1 in one graph but 2, 2, 1 in the other.

We will further examine the adjacency matrix of graph G. An adjacency matrix A is a square matrix in which the value of an element $a_{i,j}$ is 1 if there exists an edge between vertex i and vertex j otherwise it is 0. The adjacency matrix for an undirected graph is symmetric, with simple graphs having only zeroes on the main diagonal.

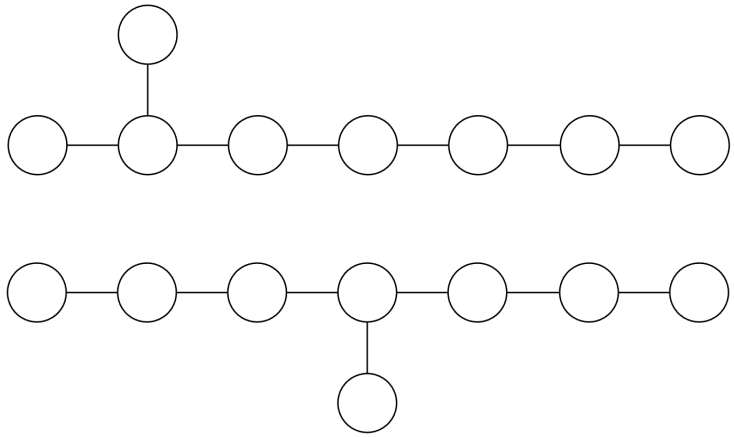


Figure 1.6: An example of two graphs with the same degree sequence that are not isomorphic.

As is stated in Def. 1.11, an isomorphism is a bijection that preserves adjacency and non-adjacency. Two graphs G and H are isomorphic if and only if the following holds for their corresponding adjacency matrices A_G and A_H:

$$A_G = PA_HP^{-1}, \tag{1.1}$$

where P is a permutation matrix. A permutation matrix is a square matrix in which there is exactly one element in each row and one element in each column equal to 1, with other elements equal to 0.

Using the aforementioned graphs G and H, we get the following matrices, where the matrix P was chosen so that 1.1 holds:

$$A_G = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} A_H = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Definition 1.12 *An isomorphism class of graphs is an equivalence class of graphs under the isomorphism relation.*

The isomorphism relation is an equivalence relation because it is reflexive, symmetric, and transitive. An equivalence relation partitions a set into equivalence classes, so the isomorphism relation partitions a set of graphs into isomorphism classes. For example, the elements of a set of all triangle graphs (3-cliques) are pairwise isomorphic, so this set forms an isomorphism class.

Definition 1.13 *An automorphism of G is an isomorphism from G to G.*

1.3 Minimal asymmetric graphs

A graph with only the trivial automorphism group, so its only symmetry is the identity, is called asymmetric. We mention groups in more detail in Section 1.4. Our work aims to examine the 18 finite minimal asymmetric undirected graphs, whose existence was proved by Pascal Schweitzer and Patrick Schweitzer in 2017 [SS17]. Their research confirms the Nešetřil conjecture that there are only finitely many finite minimal asymmetric undirected graphs. The 18 minimal asymmetric graphs are shown in Fig. 1.7. There are 8 graphs with 6 vertices, 6 graphs with 7 vertices, and 4 graphs with 8 vertices. The information underneath each graph has the form (v, e, co), where v is the number of vertices, e is the number of edges, and co is the complement graph.

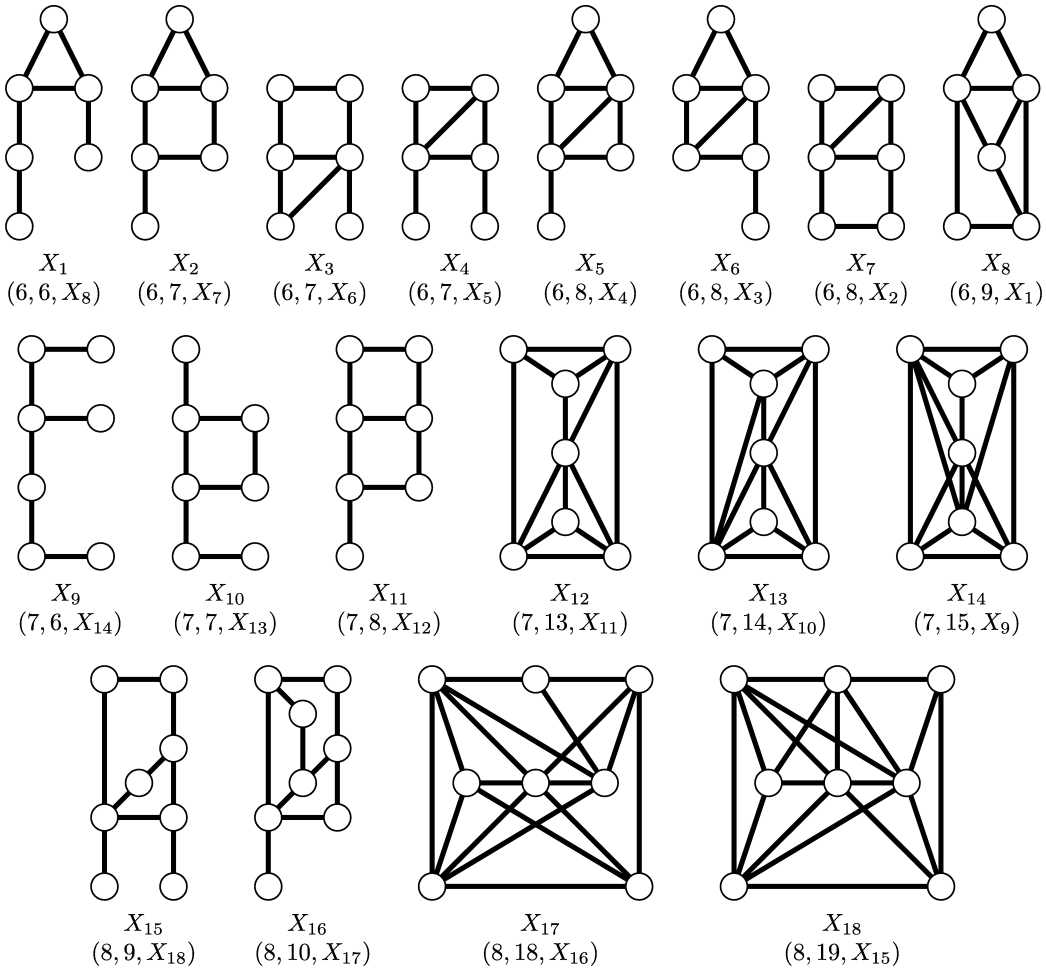


Figure 1.7: A list of all minimal asymmetric graphs [SS17].

1.4 Group theory

In this section, we introduce the necessary terms from group theory and a generalization of groups, the inverse semigroup theory.

Firstly, let's examine the symmetries of a trivial geometric object: a regular pentagon. We can imagine the pentagon as a graph with 5 vertices and 5 edges. We color the vertices using different colors: R (red), G (green), B (blue), Y (yellow), and P (purple). Imagine we fix the pentagon in place,

take a copy of it and rotate or flip it so that it precisely covers the original.

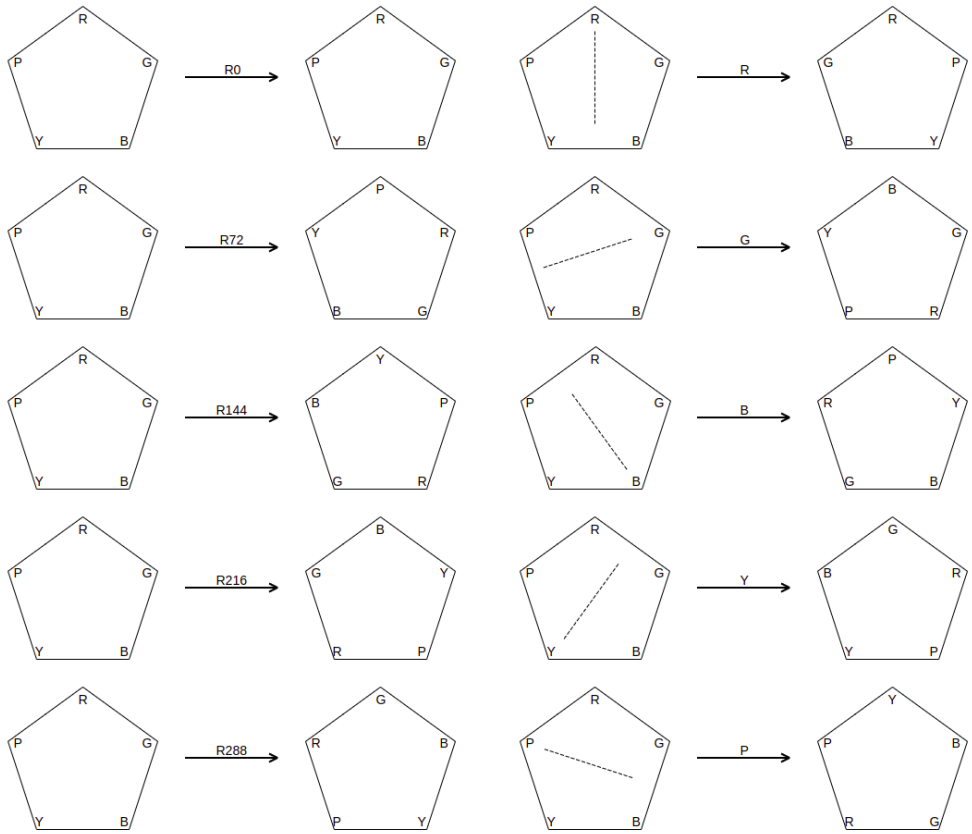


Figure 1.8: All symmetries of a regular pentagon.

There are five ways of rotating the pentagon clockwise, by 72, 144, 216, 288, and 360 (same as 0-degree rotation) degrees. We denote these rotations as $RD, D \in \{72, 144, 216, 288, 360\}$. Note that rotations by multiples of these degrees already correspond to one of the 5 rotations listed, so we ignore them. Counter-clockwise rotations correspond to one of the previously mentioned rotations, so we also ignore them.

Next, we can flip the pentagon along the perpendicular bisector of an edge. Altogether, since there are 5 edges in a pentagon, there are 5 such flips possible. We denote these flips as $fC, C \in \{R, G, B, Y, P\}$, to mean flips

*	R0	R72	R144	R216	R288	fR	fG	fB	fY	fP
R0	R0	R72	R144	R216	R288	fR	fG	fB	fY	fP
R72	R72	R144	R216	R288	R0	fY	fP	fR	fG	fB
R144	R144	R216	R288	R0	R72	fG	fB	fY	fP	fR
R216	R216	R288	R0	R72	R144	fP	fR	fG	fB	fY
R288	R288	R0	R72	R144	R216	fB	fY	fP	fR	fG
fR	fR	fY	fG	fP	fB	R0	R216	R72	R288	R144
fG	fG	fP	fB	fR	fY	R144	R0	R216	R72	R288
fB	fB	fR	fY	fG	fP	R288	R144	R0	R216	R72
fY	fY	fG	fP	fB	fR	R72	R288	R144	R0	R216
fP	fP	fB	fR	fY	fG	R216	R72	R288	R144	R0

Table 1.1: Cayley table for a group of automorphisms of a regular pentagon.

over the line of symmetry that passes through the C-colored vertex.

We can see these 10 symmetries in Fig. 1.8.

Let S be the set of all of these symmetries. We can consider all elements of S to be functions that take the original pentagon and map it to itself. Since functions can be composed, we can construct a table in which the element in i-th row and j-th column is obtained by composing elements i and j. This table is called a *Cayley table*, an example of a Cayley table for our set S can be seen in Table 1.1.

We can observe that the result of the composition of any two elements from S is also in S. This property is called a closure (under function composition). We can also see there exists a unique element $e = R0$ such that $\forall x \in S, xe = ex = x$. This element is called the *identity*. Furthermore, for every element $x \in S$, there exists a single element $y \in S$, such that $xy = yx = e = R0$, and we say that x is the inverse of y (and vice versa). The inverse of an element is unique. This property is also observable in the Cayley table since the identity element $R0$ appears exactly once in each row and each column.

We can conclude that our set S forms a group under function composition

(multiplication).

All definitions relating to group theory are taken from [Gal12].

Definition 1.14 *Let G be a set together with a binary operation (usually called multiplication) that assigns to each ordered pair (a, b) of elements of G an element in G denoted by ab. We say G is a group under this operation if the following properties are satisfied:*

1. ***Associativity*** *- the operation is associative; that is, $(ab)c = a(bc)$ for all a, b, c in G*
2. ***Identity*** *- there is an element e (called the identity) in G such that $ae = ea = a$ for all a in G*
3. ***Inverses*** *- for each element a in G, there is an element b in G (called an inverse of a) such that $ab = ba = e$*

1.4.1 Permutation groups

In our above example, we obtained the symmetries of a regular pentagon by permuting its vertices. We showed that the set of all such symmetries forms a group under function composition. This group also forms a permutation group.

Definition 1.15 *A permutation of a set A is a function from A to A that is both one-to-one and onto. A permutation group of a set A is a set of permutations of A that forms a group under function composition.*

In other words, a permutation is a bijective function from a set to itself. In total, there are $n!$ different permutations of an n-element set. The permutation α can be written in what is known as Cauchy's two-line notation as

$$\alpha = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ \alpha(x_1) & \alpha(x_2) & \cdots & \alpha(x_n) \end{pmatrix} \tag{1.2}$$

1.4.2 Cycle notation

Cycle notation is an alternative way of representing permutations. For example, let's have a permutation α of the set A = {1, 2, 3, 4, 5, 6}, given by $\alpha(1) = 4, \alpha(2) = 3, \alpha(3) = 5, \alpha(4) = 1, \alpha(5) = 2, \alpha(6) = 6$. The permutation α can be written in *cycle notation* as follows: $\alpha = (14)(523)(6)$. The process of obtaining the cycle notation for a permutation α of a set A is the following:

1. select any element x from the set A
2. successively apply the permutation α to trace all images of element x and write down the results
3. keep repeating the previous step until the successive applications of α trace back to x
4. place the results of steps 2-3 inside parentheses so we get a cycle $(x\alpha(x)$ $\alpha(\alpha(x))\alpha(\alpha(\alpha(x)))...)$, the element x we started with appears in the cycle only once
5. select an element y not yet in a cycle and repeat the previous steps until all elements of A appear in one of the cycles

We use the term length of a cycle to mean the number of elements in it. Note that the cycles in cycle notation are disjoint (they do not share elements), and we can omit any cycle that fixes an element (a cycle of length 1), so in our previous example, we get $\alpha = (14)(523)$. We will also rewrite the cycles so that the minimal element of each cycle is listed first: $\alpha = (14)(235)$.

1.4.3 Orbits and stabilizers

Definition 1.16 *Let G be a group of permutations of a set S. For each s in S, let $orb_G(s) = \{\phi(s)|\phi \in G\}$. The set $orb_G(s)$ is a subset of S called the orbit of s under G. We use $|orb_G(s)|$ to denote the number of elements in $orb_G(s)$.*

Definition 1.17 *Let G be a group of permutations of a set S. For each i in S, let $stab_G(i) = \{\phi \in G|\phi(i) = i\}$. We call $stab_G(i)$ the stabilizer of i in G.*

Definition 1.18 *Let G be a finite group of permutations of a set S. Then, for any i from S, $|G| = |orb_G(i)||stab_G(i)|$.*

Let's examine the graph G, $V(G) = \{1, 2, 3, 4, 5, 6\}$ shown in Fig. 1.9. The group of automorphisms of G is $Aut(G) = \{(1)(2)(3)(4)(5)(6), (1)(23)(45)(6), (16)(24)(35), (16)(25)(34)\}$. The orbit of an element x are all vertices to which x gets mapped by some automorphism in the automorphism group $Aut(G)$. For example, the orbits of vertex 1 are $\{1, 6\}$ and the orbits of vertex 2 are $\{2, 3, 4, 5\}$.

The stabilizers of an element x are those elements of $Aut(G)$, that map x to itself (stabilize it in its place). In other words, we can easily find the stabilizers of x by finding all elements of $Aut(G)$, in which the length of a cycle containing x is 1. The stabilizers of 1 are $\{(1)(2)(3)(4)(5)(6), (1)(23)(45)(6)\}$ and the stailizer of 2 is $\{(1)(2)(3)(4)(5)(6)\}$.

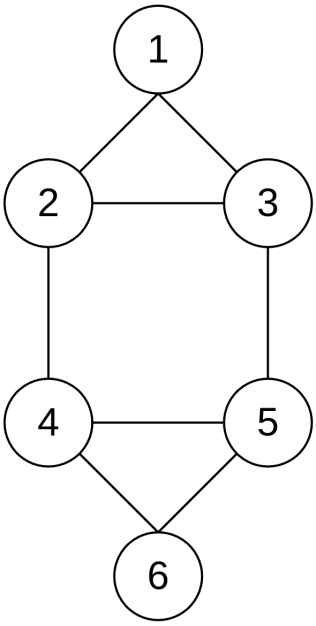


Figure 1.9: Graph G with 4 automorphisms.

The Definition 1.18, known as the $orbit - stabilizer\ theorem$, also holds as $|G| = 4 = |orb_G(1)||stab_G(1)| = |orb_G(2)||stab_G(2)|$.

1.5 Semigroups

We now explore the concept of inverse semigroups, a generalization of groups. A semigroup is a set with an associative binary operation that does not need to have all of the properties groups have. For this reason, every group is also a semigroup, but not every semigroup satisfies the properties necessary to be a group. Definitions in this section are taken from the book [Law98].

Definition 1.19 *A non-empty set together with an associative multiplication is called a semigroup, and a semigroup admitting an identity (neutral) element is called a monoid. A monoid S is said to be an inverse monoid if,*

for every $s \in S$, there exists a unique element $s^{-1} \in S$, called the inverse of s, such that $ss^{-1}s = s$ and $s^{-1}ss^{-1} = s^{-1}$ hold. Note that the operation of taking inverses has the properties that $(s^{-1})^{-1} = s$ and $(st)^{-1} = t^{-1}s^{-1}$ for any $s, t \in S$.

1.5.1 Partial permutations

We introduce the concept of partial permutations and describe a modification of the cycle notation we use to represent them.

Definition 1.20 *The set of all bijections between subsets of a set X, including the empty set is called the set of partial permutations of X and is denoted PSym(X). If $\psi : Y \to Z \in PSym(X), Y, Z \subseteq X$, then Y and Z are the domain (denoted dom) and range (ran) of ψ, respectively.*

To store partial permutations in an easy-to-understand format we modify the cycle notation described in Section 1.4.2. Cycle notation is insufficient for partial permutations, since there might be elements that appear in the domain but not in the range, in which case we cannot cyclically permute them. In cases when the process of obtaining a cycle as described in Section 1.4.2 ends outside the domain, we use a *path* notation instead [JJSS21]. A path is denoted as $[x_k, \cdots, x_2, x_1)$, where all elements are distict and $k \geq 2$. We read a path notation from right to left, so for example a path [5312) denotes a partial permutation that maps 2 to 1, 1 to 3, 3 to 5, and 5 does not get mapped to anything, it is undefined.

1.5.2 Green's relations

Definitions and concepts described in this section are from [JJSS21, Section 3.3] and [Jaj22, Section 3].

Definition 1.21 *Let S be an arbitrary monoid and $a, b \in S$ be arbitrary elements. We define two equivalence relations $\mathcal{L}$ and $\mathcal{R}$ the following way:*

1. *$a \mathcal{L} b$ if and only if there exist $x, y \in S$ such that $xa = b$ and $yb = s$.*
2. *$a \mathcal{R} b$ if and only if there exist $x, y \in S$ such that $ax = b$ and $by = a$.*

Applying the general definition of Green's relations on partial permutations, we get that the $\mathcal{R}$ relations coincide with partial permutations sharing the same range, and the $\mathcal{L}$ relations coincide with partial permutations sharing the same domain.

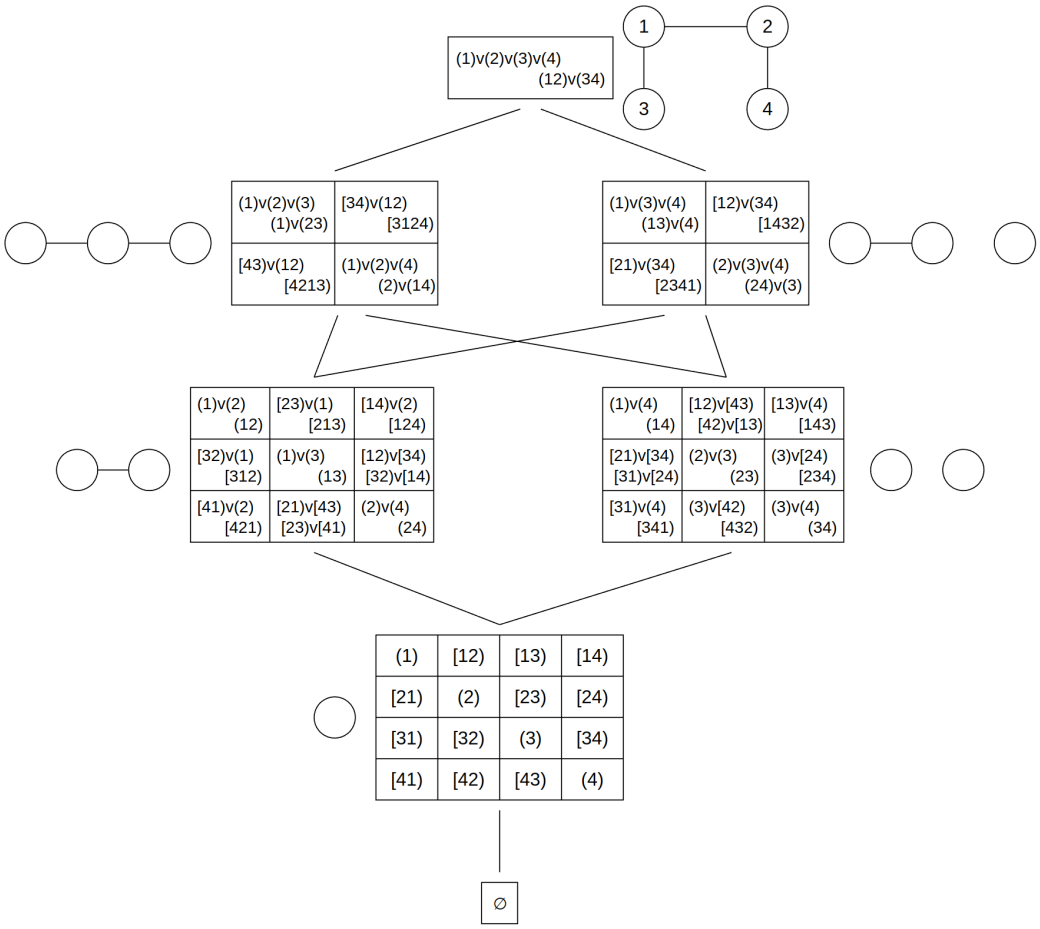


Figure 1.10: An example of a partial automorphism monoid of a graph with 4 vertices

We define a third Green's equivalence relation as $\mathcal{H} = \mathcal{R} \cap \mathcal{L}$. In an inverse monoid, every $\mathcal{R}$-class and $\mathcal{L}$-class contain exactly one idempotent, and $\mathcal{H}$-classes that contain them are the maximal subgroups of the inverse monoid.

The last Green's equivalence relation we need to define contains both $\mathcal{R}$ and $\mathcal{L}$ relations as is defined as $\mathcal{D} = \mathcal{R} \circ \mathcal{L} = \mathcal{L} \circ \mathcal{R}$, where $\circ$ is the composition of relations. In the partial automorphism monoid of graph G, the $\mathcal{D}$-classes correspond to the isomorphism classes of vertex-induced subgraphs of G.

Fig. 1.10 shows the partial automorphism monoid of a graph G with 4 vertices. In the visual representation of a monoid, we have "levels" where each level contains $\mathcal{D}$-classes of induced subgraphs with the same number of vertices. At the top is an induced subgraph with 4 vertices, the graph G itself. Below it, in the second level, we have all isomorphism classes corresponding to subgraphs with 3 vertices. Generally, for a graph with n vertices, the partial automorphism monoid consists of $n + 1$ levels. At the bottom of every partial automorphism monoid, we have a $\mathcal{D}$-class containing $\emptyset$, which corresponds to a map with an empty domain.

Let's further examine the $\mathcal{D}$-class of an induced subgraph with 2 vertices and 1 edge. As mentioned earlier, the $\mathcal{R}$ relation coincides with partial permutations that share the same range. In the $\mathcal{D}$-class, visually represented as what is known as an "eggbox" diagram, partial permutations belonging to the same $\mathcal{R}$-class are in the same row. The first row in our example contains the following partial permutations: $\{(1)v(2), (12), [23)v(1), [213), [14)v(2), [124)\}$, that share the same range $\{1, 2\}$. The elements in the second row share the range $\{1, 3\}$, and in the last row we have the range $\{2, 4\}$.

Similarly, partial permutations that share the same domain belong to the same $\mathcal{L}$-class and are in the same column. All partial permutations in the

$\mathcal{L}$-class $\{(1)v(2), (12), [32)v(1), [312), [41)v(2), [421)\}$ share the same domain $\{1, 2\}$ and all these elements are in the first column. The second column contains elements with domain $\{1, 3\}$ and elements with domain $\{2, 4\}$ are in the last column.

When constructing the $\mathcal{D}$-class, we arrange $\mathcal{R}$-classes and $\mathcal{L}$-classes so that $\mathcal{H}$-classes containing idempotents are on the main diagonal. These $\mathcal{H}$-classes containing idempotents form a group.

Chapter 2

Implementation

In this chapter, we explain our implementation, from choosing the correct programming language, designing and implementing the algorithm, optimizing our solution, and creating a GUI application to make our solution easier accessible.

While graphs are one of the most well-known and commonly used data structures across many fields of computer science, there are several computationally difficult-to-solve graph-related problems. One such problem is the graph isomorphism problem, the problem of determining whether two finite graphs are isomorphic to each other or not. The graph isomorphism problem is one of the yet not effectively solved problems in computer science. A polynomial time algorithm is not known to exist. The problem also does not belong to the family of NP-complete problems. Instead, it belongs to the family of NP-intermediate.

There are, however, certain families of graphs for which the graph isomorphism problem can be solved in polynomial time, most notably *trees*.

The current theoretical state-of-the-art algorithm got introduced by Babai in 2015 and solves the problem in quasipolynomial time [Bab15].

2.1 Programming language

This section explains our search for the correct programming language to use to solve our problem and lists its advantages and disadvantages.

The computational complexity of the GI problem or the rapid growth of the number of partial automorphisms of graphs with an increasing number of vertices would suggest the usage of programming languages known for their speed, such as compiled languages from the C-family (C, C++, Java).

Interpreted languages, such as Python, are considerably slower than compiled languages. While interpreted languages are inherently slower than compiled ones, we list some specific areas in which Python is slower. For example, the cost of calling a function is high, making programs using recursion especially slow.

However, Python is one of the most popular programming languages nowadays, mainly due to its readability and simple syntax. It commonly gets used in data analysis, machine learning, image processing, and web development. Due to its popularity, there are many third-party libraries, including libraries for working with networks, graph isomorphism, or finding the automorphism groups of graphs.

We decided to use Python in our work due to its popularity in areas that use graphs or networks and utilize existing libraries with current state-of-the-art algorithms to make our solution as efficient as possible. We describe the steps we took to optimize the runtime of our algorithm in further detail in Section 2.5.

2.2 Existing libraries

In this section, we provide details about some existing libraries used in graph and group theory. One is a Python library (*networkX*), and the other is outside the Python ecosystem (*GAP*).

GAP - (Groups, Algorithms, Programming) is a library for computational discrete algebra, mainly focused on group theory [gap]. *Nauty* is a GAP package used for finding graph automorphism groups and graph isomorphism testing, able to work with directed and undirected graphs as well [nau]. However, since nauty is a program that offers only a command line interface, we decided against its use in our implementation, since the initial startup is slow, while we focused on performance. We only use GAP to get information about the structure of a group.

NetworkX is a network analysis library written in Python [net]. It includes the implementation of the $vf2++$ graph isomorphism testing algorithm, an improved version of the original $vf2$ algorithm, introduced in 2004 [CFSV04][JM18]. The vf2++ algorithm features a significantly improved performance compared to the original algorithm, capable of determining whether two graphs are isomorphic in milliseconds for any graphs with less than 10000 vertices.

The networkX implementation of the vf2++ algorithm allows us to find all isomorphisms between given graphs. We used the function $vf2pp\_is\_isomorphic$ to check whether two graphs are isomorphic. We also used the function $vf2pp\_all\_isomorphisms$, which takes two graphs $G1$ and $G2$ as arguments and returns all possible isomorphic mappings between the vertices of $G1$ and $G2$. Using this function, we can find all automorphisms of a graph $G1$ by setting $G2 = G1$.

2.3 Classes

Below we explain the most important classes we implemented for our solution.

2.3.1 Class Graph

The class *Graph* is the main class used in our application to represent simple graphs.

Different methods of representing graphs as a data structure include an adjacency matrix, incidence matrix, or adjacency list. For a graph $G = (V, E), V = \{v_1, v_2, ..., v_n\}, E = \{e_1, e_2, ..., e_n\}$, an adjacency matrix A is a 2-dimensional matrix, in which the value of A_{ij} is set to 1 if there is an edge from vertex v_i to vertex v_j otherwise it is set to 0. Similarly, in an incidence matrix I, the value of I_{ij} is set to 1 if the vertex v_i is incident with edge e_j, else it is set to 0. The disadvantage of these two approaches is the space complexity of $O(V^2)$ because we keep track of adjacency and non-adjacency.

In our work with graph automorphisms and partial automorphisms, we are only interested in keeping track of adjacency, so we decided to use a variation of the adjacency list representation, in which each vertex gets associated with a list containing its neighbors. Instead of a list, we use a dictionary, in which the keys are vertices of a graph, and the value associated with each key is a set of neighbors of a given vertex.

To use any networkX function, we first need to convert our graph to a representation compatible with networkX, so we implemented the *get_nx_rep* method. The method *get_symmetries*, which calls the $vf2pp\_all\_isomorphisms$ function, is used to find the automorphism group of a graph, by finding all isomorphic mappings between the graph's vertices. We store the symmetries as tuples, where the vertex at the i-th index in the tuple gets mapped to the

i-th vertex in the list of vertices, assuming this list of vertices is in ascending order.

We use the function $find\_isomorphism\_classes$ with an argument k to find isomorphism classes for k-vertex induced subgraphs. We explain this function further in Sections 2.4 and 2.5.

Other useful functions we implemented include adding or removing a vertex or an edge from the graph, creating a new graph with certain edges deleted, or removing a vertex from the set of neighbors.

2.3.2 Representing partial permutations

We implemented a class called $PartialPermutation$ to represent and provide an interface for working with partial permutations, as is described in Section 1.20. The class gets initialized with two arguments, dom and ran, representing the domain and range of the partial permutation, respectively.

Both arguments are of type $tuple$, and the i-th element in dom gets mapped to the i-th element in ran. We use the function $to\_cycle\_path\_notation$ to represent the partial permutation using the cycle-path notation we described in Section 1.5.1. The method is_cycle returns True if the cycle-path notation contains only cycles and no paths, so when $dom = ran$.

We use the classes $Cycle$ and $Path$ to represent the cycle notation and path notation, respectively. For displaying both notations in an easily readable format, we implemented a method that converts instances of $Cycle$ and $Path$ to a string.

2.3.3 Class IsomorphismClass

We use the class $IsomorphismClass$ to represent isomorphism classes and store all members of the isomorphism class. Each instance has an attribute

rep, which stores a graph that serves as a representative of the given isomorphism class. In the list *graphs*, we keep all isomorphism class members, so graphs isomorphic to *rep*. We implemented methods *is_isomorphic* and *add_isomorphic* that check if some graph is isomorphic to *rep*, and if it is, add it to *graphs*. We use the method *size* to calculate the number of partial automorphisms of the isomorphism class using the following formula: $len(graphs)^2 * |Aut(rep)|$, where $len(graphs)$ is the number of members of the isomorphism class and $|Aut(rep)|$ is the number of symmetries of *rep*. Finally, we use the method *create_d_class* to create the $\mathcal{D}$-class as we explain in Section 2.4.

2.3.4 Class PartialSymmetries

The class *PartialSymmetries* is the main class used for finding partial symmetries and the partial automorphism monoid of a graph. It takes an instance of the *Graph* class and has three optional arguments, $only\_full\_symmetries$, use_timer, and $timeout\_after$. If *only_full_symmetries* is true, the algorithm will only find the automorphism group of the graph and not the entire partial automorphism monoid. If *use_timer* is true, the execution of the algorithm will time out after $timeout\_after$ seconds, with the default timeout set to 60 seconds. The method *get_number_of_partial_symmetries* counts the number of partial symmetries of a graph by finding all isomorphism classes and calculating the size of each corresponding $\mathcal{D}$-class.

2.4 Algorithm

In this section, we introduce and describe the algorithm for finding the partial automorphism monoid of a graph and talk in detail about its implementation

in Python.

Algorithm 1 An algorithm for constructing the partial automorphism monoid of a graph

```
1: Input: graph G with n vertices
2: Output: 𝒟-classes corresponding to all isomorphism classes of vertex-
       induced subgraphs of G
3: d_classes ← empty set
4: for k = 0 to n do
5:     if k == 0 then
6:         add a list containing an empty partial permutation to d_classes
7:     else
8:         isomorphism_classes ← find all isomorphism classes and their
               members for vertex-induced subgraphs with k vertices
9:         for iso_class in isomorphism_classes do
10:            d_class ← empty list
11:            for g1 in members of iso_class do
12:                row ← empty list
13:                for g2 in members of iso_class do
14:                    h_class ← list of all isomorphic mappings between the
                           vertices of g2 and g1
15:                    append h_class to row
16:                end for
17:                append row to d_class
18:            end for
19:            add d_class to d_classes
20:        end for
21:    end if
22: end for
23: return d_classes
```

The pseudocode of our algorithm can be seen in Algorithm 1. Finding the partial automorphism monoid of a graph G with n vertices involves finding isomorphism classes corresponding to k-vertex subgraphs ($0 \leq k \leq n$) along with the members of each isomorphism class.

For $k = 0$, we only have one partial automorphism, corresponding to an empty partial permutation ϕ, $dom(\phi) \cap ran(\phi) = \emptyset$.

For each $k, k \geq 1$, we first find isomorphism classes of k-vertex induced subgraphs. We generate all possible k-element permutations of the set $V(G)$. We used the *combinations* function from Python's *itertools* library, which takes an iterable and generates all possible k-element combinations [ite]. We generate all k-element combinations of a sorted list of vertices, ensuring the output will also be sorted in increasing order.

We have a class $IsomorphismClass$, that takes a graph *rep* as an argument. The graph *rep* is a representative of this isomorphism class and is also stored in the list *graphs*, which contains members of this isomorphism class. When we create an induced subgraph H, $|V(H)| = k$ we first check if this induced subgraph is isomorphic to any of our previously found k-vertex isomorphism classes, so check if H is isomorphic to a representative of any existing $IsomorphismClass$ instance. If it is, we add it as a new member of the corresponding isomorphism class. Otherwise, it does not belong to any previously found isomorphism class, so we create a new instance of $IsomorphismClass$ and make H the representative of this instance.

Since we generated the k-vertex combinations of vertices in sorted order, the members of each isomorphism class stored in the list *graphs* are also sorted. As a result, while creating the corresponding $\mathcal{D}$-class for each isomorphism class, we save the time otherwise needed to sort the graphs.

The creation of the $\mathcal{D}$-class can be seen in lines 10–19. When we represent the $\mathcal{D}$-class using the "eggbox" diagram, each row corresponds to partial permutations having the same range ($\mathcal{R}$-classes), and each column contains partial permutations with the same domain ($\mathcal{L}$-class). Each cell in the eggbox diagram ($\mathcal{H}$-class) then represents those partial permutations, that share the same domain and range. Because the graphs are sorted, only the cells on the main diagonal contain idempotents. From a programming point of view,

the eggbox diagram is a simple 2D array. The range of all elements in i-th row corresponds to the vertex set of i-th member of the isomorphism class, and the domain of elements in j-th column corresponds to the vertex set of j-th member of the isomorphism class. The number of elements in each $\mathcal{H}$-class is equal to the size of the automorphism group of members of the isomorphism class. We get all elements of $\mathcal{H}$-class with indexes ij by generating all possible isomorphic mappings between the vertices of j-th and i-th member of the isomorphism class.

2.5 Runtime and optimization

In this section, we look at different approaches we explored to make our solution more efficient and compare their runtimes. We ran all tests on *Amazon EC2 C5 instances* commonly used for computationally difficult tasks [ec2]. They use Intel Xeon Platinum 8000 series processors, clocked up to 3.6 GHz. We used the $c5.2xlarge$ instance, which has 8 vCPUs and 16 GiBs of memory and runs on an Amazon Linux software image.

2.5.1 Python interpreter

First, we compare the speed of the base Python interpreter, CPython, and an alternative implementation of Python called PyPy [pyp]. PyPy features a Just-in-Time compiler, which improves performance, especially for longer-running and computationally difficult programs. Some situations, in which PyPy might be slower than CPython are short-running programs, which are slower due to PyPy having to start the Just-in-Time compiler. Also, it is not as well-optimized as CPython for interacting with programs that utilize C extensions.

To measure the performance of CPython and PyPy, we ran a test on a simple recursive function calculating the n-th number in the Fibonacci sequence, with $\mathcal{O}(2^n)$ time complexity. We chose this test because function calls in CPython are generally expensive, resulting in poor performance of recursive functions, so PyPy should perform better.

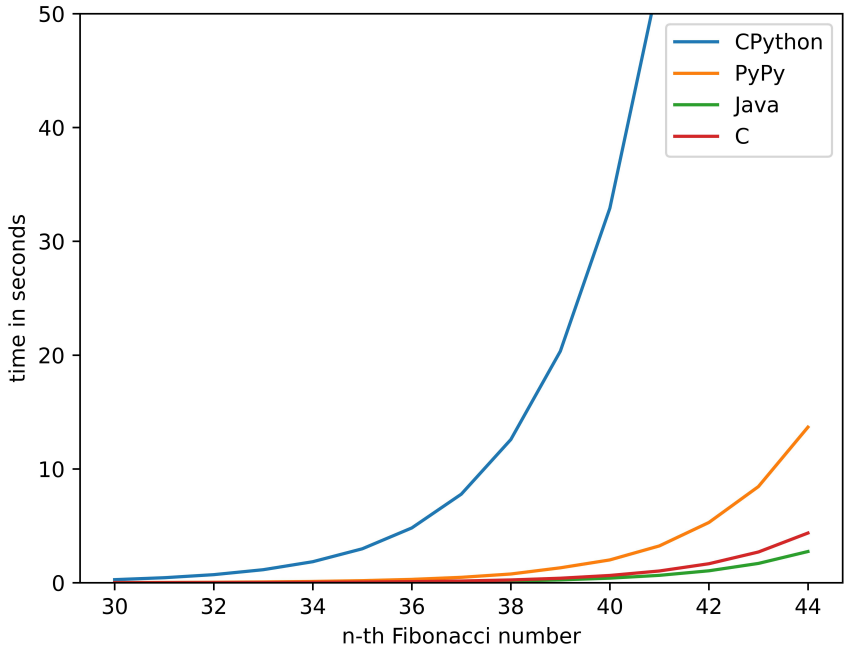


Figure 2.1: Runtime comparison for recursive Fibonacci function.

We ran the test once using Python 3.9.16 with the packaged CPython interpreter and then with PyPy3.9, the equivalent PyPy release. Figure 2.1 shows the results of this test. For every Fibonacci number, we ran the function 5 times, with the times shown being the averages of these 5 runs. Even from this basic example, the performance advantage of using PyPy over the CPython interpreter is apparent. For comparison, we also included the results for compiled languages like Java and C. Despite the significant time

savings gained by using PyPy, it is still not as fast as compiled languages. However, we believe the existence of extensive graph libraries in the Python ecosystem outweighs this disadvantage.

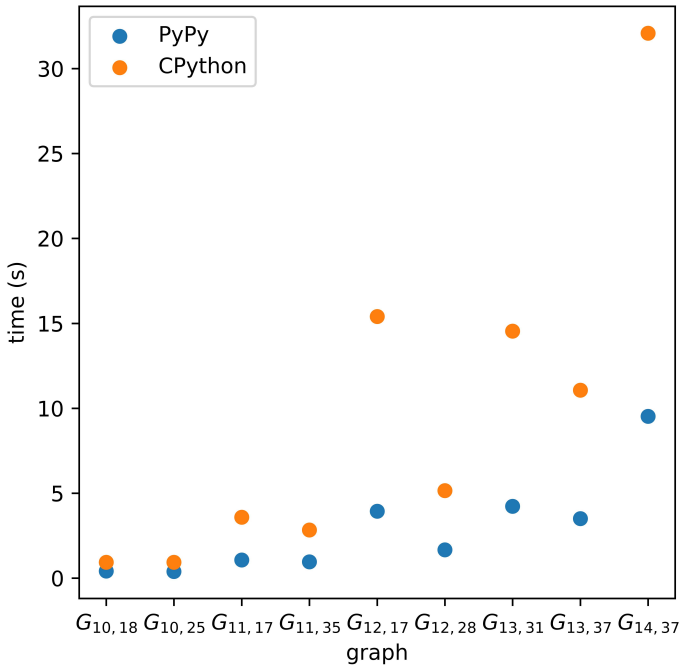


Figure 2.2: Runtime comparison of our algorithm for CPython and PyPy for graphs $G_{v,e}$, where v is the number of vertices, and e is the number of edges.

To verify PyPy is indeed faster than CPython for our specific use case, we ran tests with our algorithm for finding the partial automorphism monoid. Results of this test are in Fig. 2.2. PyPy achieved better performance in all test cases, and as expected, the difference got more significant for graphs with longer runtimes.

2.5.2 Algorithm optimization

While we achieved significantly better performance for our solution just by using a different interpreter, we took additional steps to improve the performance further.

We specifically focused on the process of finding isomorphism classes and their member for vertex-induced subgraphs. Our first version put the isomorphism classes into a list, so for every new subgraph, we had to traverse this list to check whether the graph is a member of any previously found isomorphism class, which resulted in many calls of the *is_isomorphic* function (isomorphism checks). In order to decrease the number of isomorphism checks, we used an isomorphism class filtering approach. We filter isomorphism classes by the degree sequences of their representatives. We create a dictionary in which the keys are degree sequences, and the associated values are isomorphism classes with that degree sequence.

When using the degree sequence filtering approach, we use the following process for every induced subgraph with degree sequence *ds*. We first check if a key *ds* already exists in the dictionary. If not, we know this subgraph does not belong to any isomorphism class we found previously. If the key exists, we check whether the graph is isomorphic to the representative of any isomorphism class with the same degree sequence. In Python, searching for a key in a dictionary and adding a key gets done in $\mathcal{O}(1)$ time in an average case, making the approach especially effective for graphs that contain many unique degree sequences and graphs with many induced subgraphs [dic].

Our final improvement involved filtering the subgraphs using not only their degree sequence but also their *triangle sequence*. We calculate the triangle sequence by calculating the number of triangles each vertex is a part of, where a triangle is a 3-clique, and sorting these numbers in ascending

order. To calculate the number of triangles containing a given vertex v, we take the set of neighbors $N(v)$ and count how many vertices from unique 2-element subsets of $N(v)$ are connected with an edge.

In Table 2.1, we can see the difference in the number of isomorphism checks between the version that does not filter the induced subgraphs, the version in which we filter them by their degree sequence, and finally, the version that filters not just by the degree sequence, but also by the triangle sequence.

A significant reduction in the number of isomorphism checks greatly decreased the time required to find isomorphism classes.

We included two graphs with 11 vertices and only 1 and 2 edges to observe cases in which the time saved using dtf and df is minimal compared to nf. Because these graphs are sparse, a vast majority of induced subgraphs of these graphs will have the same degree and triangle sequences. However, even in the case of graphs with 0 edges, the number of isomorphism checks will always be less for dtf and df approaches compared to nf. All graphs with less than five vertices having the same degree sequence (and consequently degree triangle sequence) are always isomorphic, so we need no isomorphism checks for induced subgraphs with less than five vertices. While we decrease the number of isomorphism checks, the cost of calculating the degree sequence or triangle sequence is comparable to the cost of an isomorphism check, so the runtime is very similar.

In some of the remaining cases, we see a significant decrease in the number of isomorphism checks when using graph filtering as well as a substantial decrease in the runtime. In the case of the 16 vertex graph with 84 vertices ($\delta = 0.7$), dtf decreased the number of isomorphism checks by more than 99%, while the runtime decreased by 83%. On the other hand, in the case

of a graph with 14 vertices and a density of 0.82, the difference was less noticeable than in the earlier case. While we still decreased the number of isomorphism checks by more than 98%, the time saved was only 9.3%.

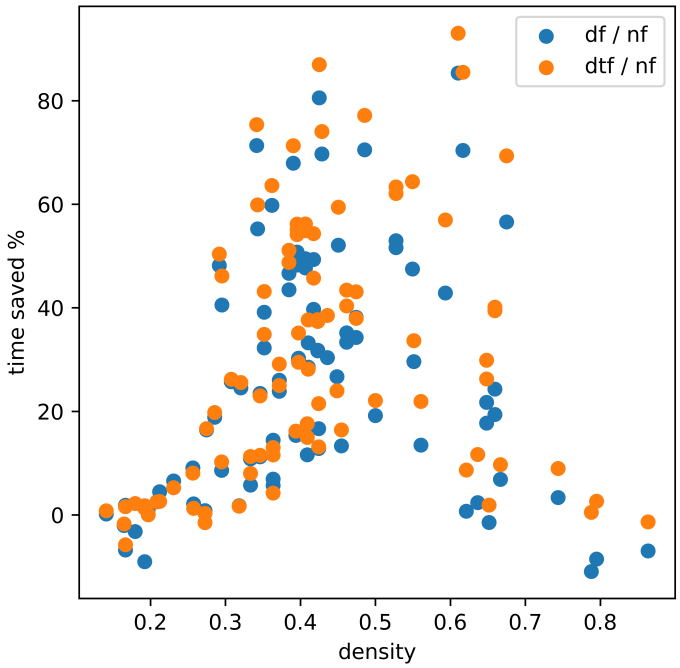


Figure 2.3: Comparison of the time saved when using dtf and df graph filtering approaches compared to no filtering (nf).

In conclusion, the time we can save by filtering the graphs by their degree sequences or degree triangle sequences depends on the graph's structure. The time saved will be less for sparse graphs or their complements, dense graphs. The time saved will be the greatest for graphs with many unique degree and triangle sequences, which causes a significant drop in the number of isomorphism checks compared to no filtering approach. Generally, the df and dtf should perform significantly better for graphs with density $\delta = 0.5 \pm 0.3$.

Graph info		Filter type					
|V(G)|	|E(G)|	nf	t(s)	df	t(s)	df + dtf	t(s)
11	1	2514	74.024	1473	72.961	1473	71.714
11	2	3028	39.448	1467	38.732	1467	39.053
14	75	822268	70.703	18619	67.952	13763	64.088
15	64	24511511	84.855	176682	24.324	20380	9.337
15	76	11915998	61.482	94851	31.661	23577	17.083
15	84	3959678	72.918	68546	63.032	27091	42.625
16	84	56858121	247.703	295276	99.501	47644	41.821

Table 2.1: Comparison between the number of isomorphism checks when using different induced subgraph filtering approaches (nf - no filter, df - degree sequence filter, dtf - degree triangle sequence filter).

To verify these claims, we ran tests on randomly generated graphs with 12+ vertices, with the results shown in Fig. 2.3. For some sparse graphs with a density of less than 0.2 and dense graphs with a density of at least 0.8, the sequence filtering approaches might be slower when compared to the no-filtering approach. Meanwhile, graph filtering performs better in the remaining cases, with a few exceptions.

2.6 Web Application

Some other theses that dealt with graphs and their automorphisms used a simple command line interface. We decided against this approach since command-line applications are generally difficult to work with. For example, in the case of manually entering a graph, if the user makes a mistake, they are required to re-enter the entire graph again. Also, displaying larger automorphisms or partial automorphism monoids in the command line with basic text graphics is confusing.

Therefore, in order to make the application more user-friendly, we created a dynamic single-page web application. On the frontend, we worked

in the Vue.js ecosystem, using Vue Router for routing, Pinia for page-wide storage, and Vuetify for design [vuea, pin, vueb]. On the backend, we used Django, a Python web framework, and Django Rest framework, a framework for creating REST APIs [dja, drf]. The backend-frontend communication gets handled using Axios, and the data exchanged is in JSON format [axi].

2.6.1 Minimal assymetric graphs

The home page of the web page is dedicated to the primary goal of this thesis, minimal asymmetric graphs and their partial symmetries. Users can select which minimal asymmetric graph's symmetries they want to view in a simple drop-down list. The partial symmetries are then loaded and displayed to users. The structure of the partial automorphism monoid is displayed at the top of the page, as shown in Fig. 2.4. It illustrates the structure of the partial automorphism monoid but also serves as a navigation, where users can scroll to a $\mathcal{D}$-class of a given induced subgraph by clicking the appropriate button. How $\mathcal{D}$-classes are displayed on the website is shown in Fig. 2.5. Above the $\mathcal{D}$-class is a visualization of a representative of the isomorphism class, generated using the *pyvis* library. Vertices in the visualization can be dragged, and users can zoom in or out on parts of the graph. Users can get additional information about the elements on the main diagonal, that contain idempotents and form a group, by clicking the information icon.

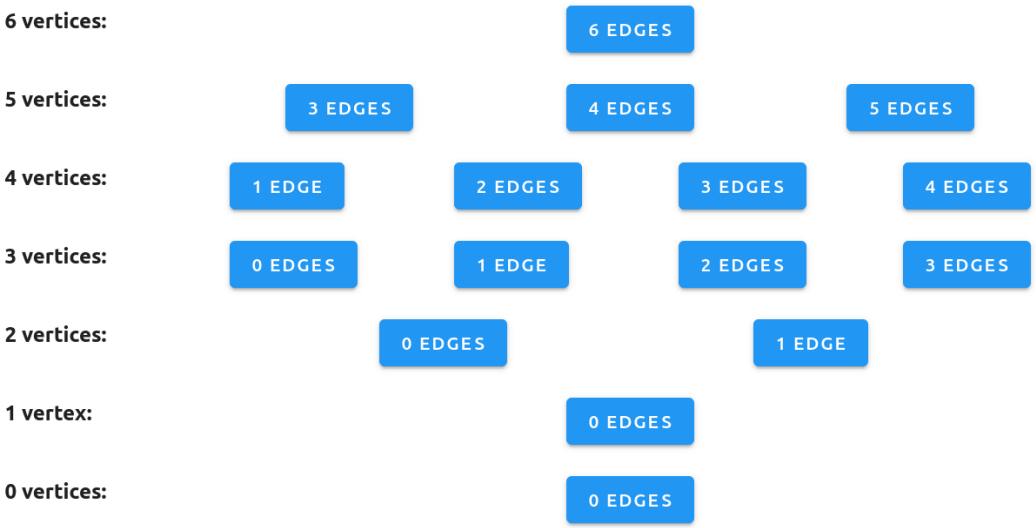
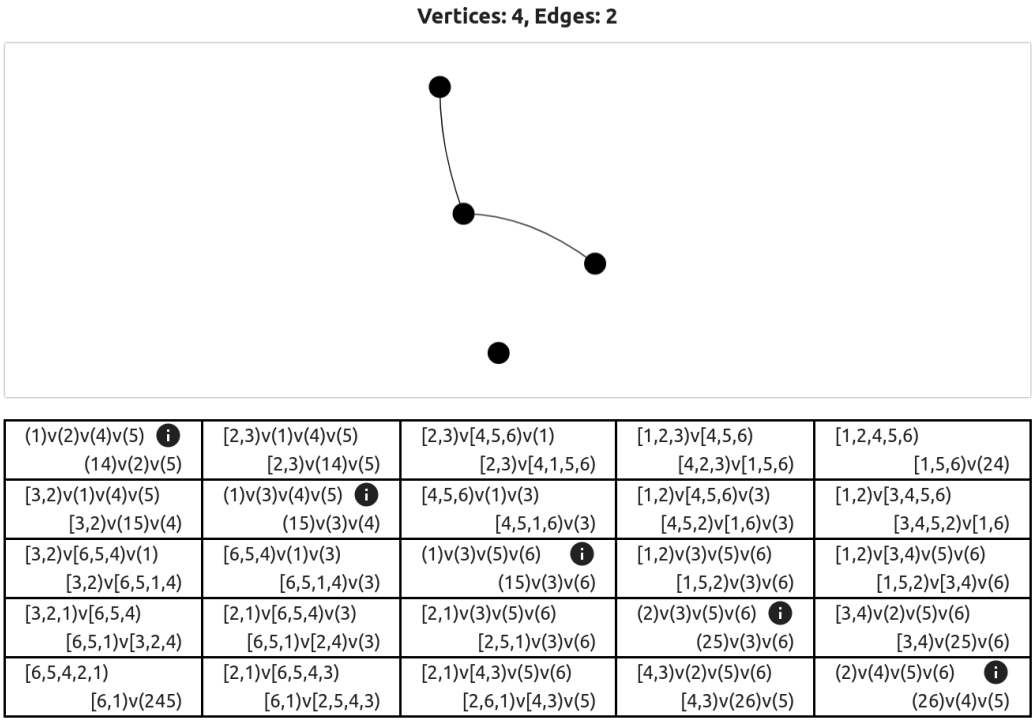


Figure 2.4: An example of how the structure of the partial automorphism monoid gets displayed on the webpage.



(1)v(2)v(4)v(5) (14)v(2)v(5)	[2,3)v(1)v(4)v(5) [2,3)v(14)v(5)	[2,3)v[4,5,6)v(1) [2,3)v[4,1,5,6)	[1,2,3)v[4,5,6) [4,2,3)v[1,5,6)	[1,2,4,5,6) [1,5,6)v(24)
[3,2)v(1)v(4)v(5) [3,2)v(15)v(4)	(1)v(3)v(4)v(5) (15)v(3)v(4)	[4,5,6)v(1)v(3) [4,5,1,6)v(3)	[1,2)v[4,5,6)v(3) [4,5,2)v[1,6)v(3)	[1,2)v[3,4,5,6) [3,4,5,2)v[1,6)
[3,2)v[6,5,4)v(1) [3,2)v[6,5,1,4)	[6,5,4)v(1)v(3) [6,5,1,4)v(3)	(1)v(3)v(5)v(6) (15)v(3)v(6)	[1,2)v(3)v(5)v(6) [1,5,2)v(3)v(6)	[1,2)v[3,4)v(5)v(6) [1,5,2)v[3,4)v(6)
[3,2,1)v[6,5,4) [6,5,1)v[3,2,4)	[2,1)v[6,5,4)v(3) [6,5,1)v[2,4)v(3)	[2,1)v(3)v(5)v(6) [2,5,1)v(3)v(6)	(2)v(3)v(5)v(6) (25)v(3)v(6)	[3,4)v(2)v(5)v(6) [3,4)v(25)v(6)
[6,5,4,2,1) [6,1)v(245)	[2,1)v[6,5,4,3) [6,1)v[2,5,4,3)	[2,1)v[4,3)v(5)v(6) [2,6,1)v[4,3)v(5)	[4,3)v(2)v(5)v(6) [4,3)v(26)v(5)	(2)v(4)v(5)v(6) (26)v(4)v(5)

Figure 2.5: An example of the $\mathcal{D}$-class for an isomorphism class.

2.6.2 Custom graphs

We also created a page for users to enter custom graphs in an easy-to-use interface. Users can select the number of vertices of a graph in a slider, with the maximum amount of vertices set to 10. The reason for setting the limit at 10 is the exponential increase in the number of partial symmetries. For example, 10-vertex graphs might have more than a million partial symmetries. Limiting the web page to smaller graphs also decreases the server load and the amount of data exchanged between the server and the client.

For every vertex, users can enter the set of neighbors of this vertex by selecting vertices in a multi-select element. Since the graphs used by our application are undirected, once the user selects vertex u to be the neighbor of vertex v, the vertex v gets set automatically as the neighbor of vertex i. After the user updates the set of neighbors for any vertex, a visualization of a graph based on the current values is displayed.

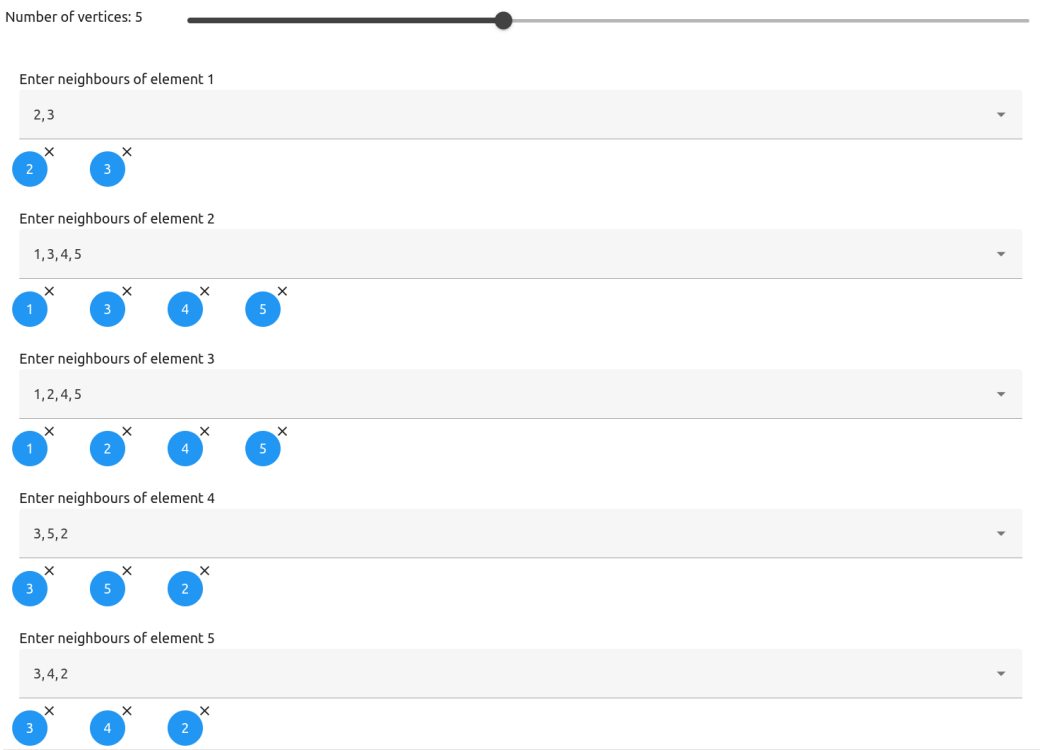


Figure 2.6: Custom graph creation component.

Chapter 3

Results

In this chapter, we present the results we achieved with our application and algorithm for finding the partial symmetries of combinatorial structures. We also provide some comparisons to existing solutions, explore asymmetric graphs in further detail, and explore how the number of partial automorphisms depends on the graph's structure.

3.1 Minimal asymmetric graphs

The main aim of this thesis was to study minimal asymmetric graphs and their partial symmetries. We know that almost all graphs are asymmetric, so using inverse monoids and finding partial symmetries of graphs provides us with useful information about the structure we want to study.

Table 3.1 shows the minimal asymmetric graphs, their numbers of vertices, edges, pairwise non-isomorphic induced subgraphs (number of isomorphism classes), and partial symmetries. We know from [JJSS21] that the structures of partial automorphism monoids for graph G and its complement $\tilde{G}$ are equal. We only list 9 minimal asymmetric graphs, since the number

Graph code	Vertices	Edges	# of non-isomorphic induced subgraphs	# of partial symmetries
X1	6	6	20	768
X2	6	7	22	704
X3	6	7	20	714
X4	6	7	21	680
X9	7	6	28	3373
X10	7	7	30	2793
X11	7	8	29	2553
X15	8	9	45	9728
X16	8	10	45	8560

Table 3.1: The number of non-isomorphic induced subgraphs and partial symmetries of minimal asymmetric graphs (graph codes taken from Fig. 1.7).

of induced subgraphs and the number of partial symmetries is the same for the complement of each of these graphs.

We found the partial automorphism monoid for each of the minimal asymmetric graphs. In Appendix, we present the partial automorphism monoid for a graph with 6 (X1) and 7 (X9) vertices as an example.

3.2 Performance comparison

We programmed our application to work not just with minimal asymmetric graphs, but with any graph. We compared the performance of our application to a bachelor's thesis *Partial symmetries of graphs* written by Michal Slávik [Slá21]. Note that the goal of his thesis was only finding all partial symmetries of a given graph, without constructing the partial automorphism monoids. On the other hand, the times for our results include not only the time required to find all partial symmetries but also constructing the partial automorphism monoid, meaning creating the $\mathcal{D}$-classes for all isomorphism classes.

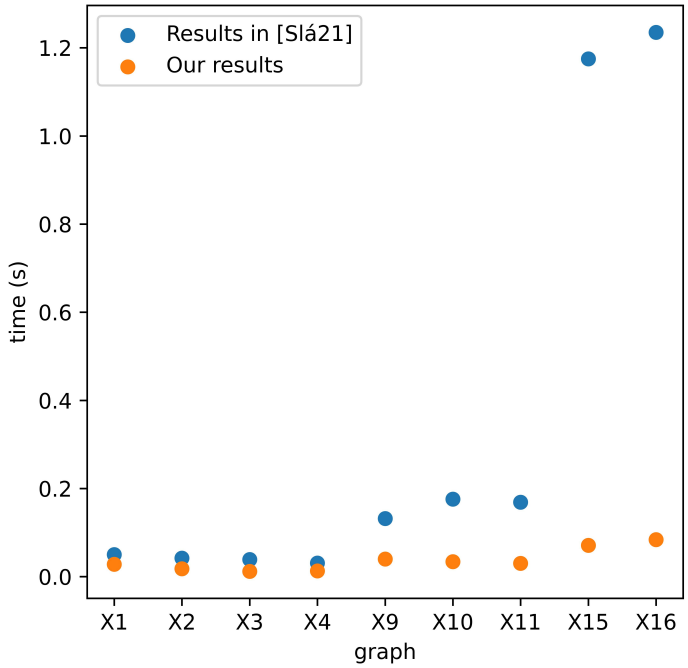


Figure 3.1: A comparison of the runtime for minimal asymmetric graphs.

A comparison of the runtime for minimal asymmetric graphs is in Fig. 3.1. We ran the program three times for each graph, with the times shown being averages of these runs. Our solution was marginally quicker for minimal asymmetric graphs with 6 vertices. We achieved slightly better performance for 7-vertex graphs, and the difference became more significant for 8-vertex graphs. Our solution found the partial automorphism monoid for any minimal asymmetric graph in under 0.1 seconds.

We also compared the runtimes for randomly generated graphs with 7-10 vertices with the results of these tests shown in Fig.3.2. Our solution achieved significantly better performance for larger graphs with 9 or 10 vertices.

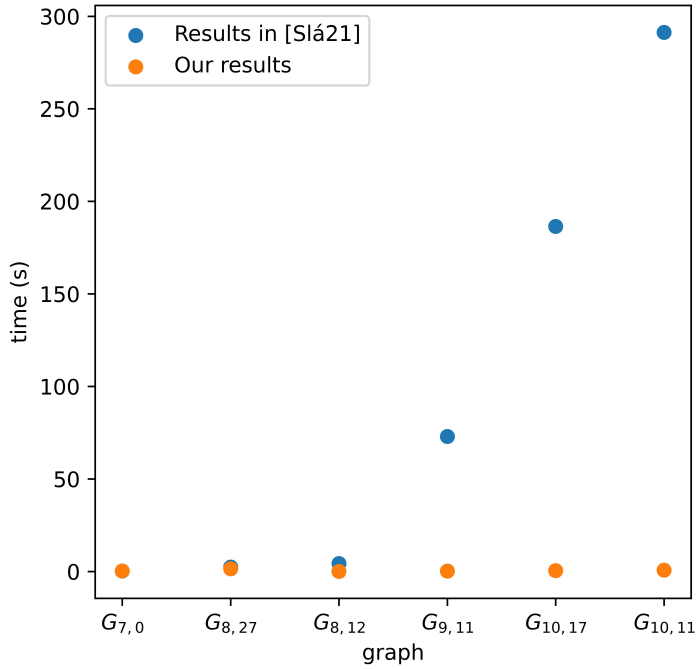


Figure 3.2: Comparison of the runtime for graphs $G_{v,e}$, where v is the number of vertices and e is the number of edges.

3.3 Tests on random graphs

We ran a test involving finding all isomorphism classes, the members of each class, and constructing the partial automorphisms monoid for 151 different graphs, with 6 to 17 vertices. A vast majority of these graphs was generated randomly, with some entered manually to check for particular properties. We ran these tests using all graph filtering approaches described in 2.5. We ran the test 3 times for each graph and then calculated the averages from these runs.

Let's first compare the no filtering (nf) approach to degree sequence filtering (df), with the results shown in Table 3.2. Degree filtering was faster

# of vertices	# of graphs	# times df < nf	df / nf (%)
any	151	102	85.75
10	24	5	108.22
11	24	9	101.96
12	27	23	94.46
13	28	25	80.66
14	25	24	64.44
15	6	6	39.36
16+	6	6	31.25

Table 3.2: Performance comparison between degree sequence filtering (df) and no filtering (nf) approaches.

# of vertices	# of graphs	# times dtf < df	dtf / df (%)
any	151	113	94.47
10	24	13	99.53
11	24	18	97.86
12	27	18	97.57
13	28	21	95.77
14	25	24	88.35
15	6	6	87.19
16+	6	6	69.28

Table 3.3: Performance comparison between degree triangle sequence filtering (dtf) and degree sequence filtering (df) approaches.

for 102 graphs, and no filtering achieved better time in 49 cases. Overall, we saved around 15% by using the df approach. We see that for 10-vertex graphs, the nf was around 8% faster, but the runtime for these graphs was mostly less than 1 second with a minimal time difference between the two approaches. Generally, nf achieved better performance for sparse and dense graphs ($\delta \leq 0.2$ or $\delta \geq 0.8$). In conclusion, the df approach was faster on average for graphs with 12 or more vertices, with the time savings increasing for larger graphs.

Table 3.3 shows the comparison between the degree sequence filtering (df) and the degree triangle sequence filtering (dtf) approaches. In this case, dtf

# of vertices	# of graphs	# times hf < dtf	hf / dtf (%)
any	151	118	96.67
10	24	8	102.38
11	24	21	94.14
12	27	24	95.11
13	28	25	95.76
14	25	21	94.51
15	6	6	93.62
16+	6	4	96.14

Table 3.4: Performance comparison between degree triangle filtering (dtf) and heuristic filtering (hf) approaches.

achieved better performance in 113 of 151 graphs, saving 5.5% on average. The performance of dtf and df was very similar for all graphs with 10 to 13 vertices, with the difference starting to be more noticeable for larger graphs with 14+ vertices.

As we already explained in Section 2.5 and as we can see in Table 3.2 and Table 3.3, all 3 approaches achieve relatively similar performance for graphs with less than 12 vertices. Therefore, we tried one additional approach using the following heuristic: we use the degree sequence filtering for all graphs with less than 11 vertices and for all graphs with densities $\delta \leq 0.2$ or $\delta \geq 0.8$, for other graphs, we use degree triangle sequence filtering. The results of this test are in Table 3.4. Using the heuristic filtering, the runtime decrease by an average of 3% compared to the previously quickest approach, the degree triangle filtering. It was only slower on average for graphs with 10 vertices, however, the difference was negligible.

3.4 Asymmetric depth

Especially for larger graphs, it might be useful to study the structure of the graph to predict the number of partial symmetries the graph has. We

introduce a definition for *asymmetric depth*, a metric that can measure the asymmetricity of graphs.

Definition 3.1 *For any n vertex graph, asymmetricity level $AsymL = i$ means all $n, n-1, n-2, \cdots, n-i$ vertex-induced subgraphs are asymmetric and pairwise non-isomorphic.*

The asymmetric depth of a graph G is a number i, such that G has an asymmetricity level i, but not $i+1$. We use $AsymD(G)$ to denote the asymmetric depth of graph G.

For example, when given two graphs with the same number of vertices, one can make the assumption the graph with higher asymmetric depth will have fewer partial symmetries than the other graph.

We created our application to provide an interface for easy work with graphs, graph symmetries, and partial symmetries. As a result, it allowed us to create a simple script to answer the following question:

Does any asymmetric graph with $n, n \leq 10$ vertices exist with $AsymL = 1$?

We used the collection of all unlabelled simple graphs generated by McKay and published in GAP format [mck]. We know there are no asymmetric graphs with $2 \leq n \leq 5$ vertices [PE63]. Since there are no asymmetric graphs with 5 vertices, we know that none of the 8 asymmetric graphs with 6 vertices, of which all are minimal, have asymmetric subgraphs with 5 vertices[oeia][SS17].

We then checked all 1044 graphs with 7 vertices, of which 152 are asymmetric [oeib]. We found that none of these graphs have $AsymL = 1$.

We then found that of the 3696 asymmetric unlabelled graphs with 8 vertices, there are 8 graphs with asymmetricity level 1. We also checked

all asymmetric graphs with 9 and 10 vertices and found 2608 such 9-vertex graphs and more than a million 10-vertex graphs with asymmetricity level 1.

Based on the findings from [PE63], we know that "almost all finite graphs are asymmetric". The number of asymmetric graphs with n vertices gets closer to the total number of possible graphs with n unlabeled vertices as n grows. We also expect the number of asymmetric graphs with n vertices with asymmetricity level 1 to increase as n grows. We can see this in our experiments, as we found 8 8-vertex graphs with asymmetricity level 1, from a total of 3696 8-vertex asymmetric graphs. For $n = 9$ there are 135004 asymmetric graphs, of those 2608 have asymmetricity level 1, and for $n = 10$ there are more than a million graphs with asymmetricity level 1 of almost 8 million asymmetric graphs.

We then extended our original question:

Are there graphs with asymmetricity level 2?

We checked all previously found graphs and discovered no such graphs among graphs with 10 or fewer vertices. We did not check all unlabelled graphs with 11 vertices, since there are more than a billion such graphs. Instead, we changed our approach. We took an asymmetric graph with 10 vertices and created all possible 11-vertex graphs by adding a new vertex to it. In total, there are 2^{10} possible ways of adding a new vertex to a 10-vertex graph. Using this approach, we found 11-vertex graphs with asymmetricity level 2.

We assume the reason why there are such graphs with 11 vertices but not with 10 vertices is that for a graph with 10 vertices, there are $\binom{10}{8} = 45$ induced subgraphs with 8 vertices, but there are only a total of 3696 8-vertex asymmetric graphs, but for 11 vertex graph there are $\binom{11}{9} = 55$ 9-vertex induced subgraphs, but a total of 135004 9-vertex asymmetric graphs.

Since we were able to answer our previous question, we proposed the following tasks:

1. *For any given graph G, find AsymD(G).*

2. *Find an n-vertex graph with asymmetricity level i.*

3. *Find the maximal asymmetric depth any n-vertex graph can have.*

Utilizing the previous findings and our extensive library, we quickly answered task 1 by implementing a function $find\_asym\_d$, which takes a graph as an argument and returns its asymmetric depth.

For task 2, we extended the approach described previously. By taking any n-vertex asymmetric graph, we can create a $n + 1$-vertex graph by adding a new vertex. In total, there are 2^n different ways of doing this, since the new vertex is either isolated or we add it as a neighbor to any $k, 1 \leq k \leq n$-vertex subset of the set of vertices. We then use the function *has_asymmetricity_level* to verify if the new graph has asymmetricity level i. Using this approach, we found graphs with 14 vertices and asymmetricity level 3. However, this approach is computationally difficult and would not be appropriate for finding all n vertex graphs with a given asymmetricity level.

Clearly, for a graph with n vertices, there is no point in checking if it has asymmetricity level i if $\binom{n}{i} > Asym(i)$, where $Asym(i)$ is the total number of asymmetric graphs with i vertices. For example, for 11 vertex graphs and asymmetricity level 4, we have $\binom{11}{7} = 330$ vertex-induced subgraphs, yet only 152 asymmetric 7-vertex graphs exist.

Finally, we wanted to find graphs with an asymmetric depth of 4 or more. We randomly generated 15-30 vertex graphs, calculated their asymmetric depth, and were able to find graphs with asymmetric depth 4.

Due to the combinatorial explosion occurring when working with graphs, their subgraphs, and symmetries, answering tasks 2 and 3 is a computationally difficult problem. We believe the initial findings presented here can serve as a baseline for future research in this area.

3.5 Number of partial automorphisms

Due to how quickly the number of partial symmetries rises when the number of vertices increases, we wanted to know if we can predict the number of partial symmetries of a graph by looking at its structure. For this reason, we calculated the number of partial symmetries for all unlabelled graphs with $n, 3 \leq n \leq 9$ vertices.

To find the number of partial automorphisms of a given graph, it is only necessary to find all isomorphism classes and all their members. We can calculate the number of partial symmetries of an isomorphism class I using the formula $|I|^2 \times |Aut(rep_I)|$, where rep acts as a representative of I.

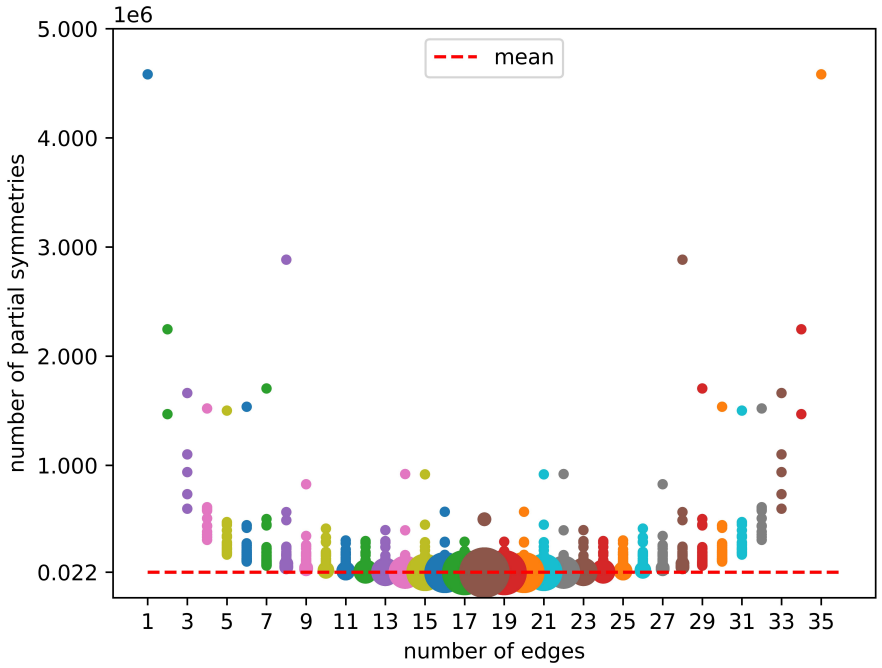


Figure 3.3: The number of partial symmetries for all 9-vertex graphs, dot size denotes the number of graphs.

Fig. 3.3 shows the number of partial symmetries for graphs with 9 vertices, where we excluded K_9 and its complement. The size of the dot scales proportionally to the number of graphs having that number of partial symmetries. One can quickly notice how symmetrical this image is. It is to be expected since we know that a graph and its complement share the same partial automorphism monoid structure and therefore have the same number of partial symmetries.

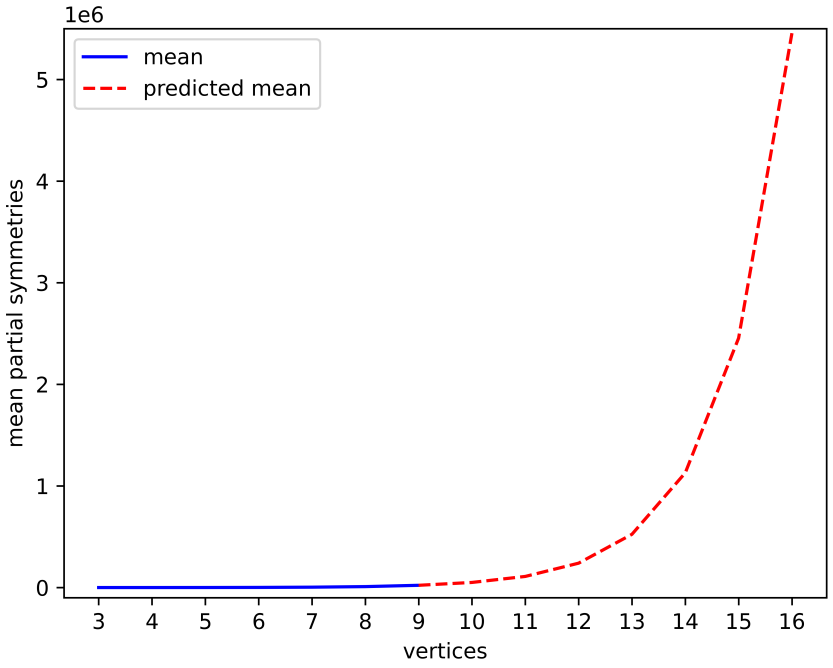


Figure 3.4: Mean number of partial symmetries for graphs with 3-9 vertices with predictions for graphs with 10-16 vertices.

We made the following observations by studying the results obtained for all graphs with 9 or fewer vertices. Firstly, for any n-vertex graph, the number of partial symmetries is equal to the number of partial permutations of an n-set in two cases: for the complete graph K_n and its complement (graph with n vertices and 0 edges) [par]. Removal of just one edge from K_n significantly reduces the number of partial symmetries. For $n = 9$, we go from 17572114 partial symmetries for K_9 to 4582270 partial symmetries for $K_9 \setminus \{e\}$.

Next, we notice that for graphs with k edges, $1 \leq k \leq 9$, there is always one graph with a number of partial symmetries higher than other k-edge graphs. These graphs consist of a k-vertex star with all other vertices being isolated.

Despite K_9 having more than 4.5 million partial symmetries, the mean number of partial symmetries for 9 vertex graphs is only 22154, a decrease of 99.5%. We also calculated the mean values for graphs with 3 to 8 vertices, and based on the data, we predicted the mean of partial symmetries for graphs with less than 17 vertices. The prediction can be seen in Fig. 3.4.

Conclusion

The main goal of our thesis was the study of minimal asymmetric graphs and their partial symmetries. For this purpose, we implemented a computer application in Python that constructs partial automorphism monoids for graphs. We programmed our application to work not only with minimal asymmetric graphs but with any graph. The process of finding all partial automorphisms of a graph is a computationally difficult task, due to combinatorial explosion, the rapid growth in the number of partial permutations, where for a 20 vertex graph, it is possible to have 1.7 sextillion ($1.7 * 10^{21}$) partial permutations. As we demonstrated in Section 3.3, our application can find all partial permutations for randomly generated graphs with at most 17 vertices and we achieved better performance than previous solutions, as we showed in Section 3.2.

Certain improvements could be made to our program to make it even more effective, such as optimizing it for certain families of graphs. For example, the graph isomorphism problem has a polynomial-time solution for trees [ZTWX11].

In our graph filtering approach, we filtered graphs by their degree and triangle sequences. A triangle graph is one of 30 2- to 5-node graphlets [EMP$^+$20], connected induced subgraphs. Future research may involve further filtering graphs by graphlets. This would involve examining the struc-

ture of a graph and determining what graphlets would offer the biggest time save, by decreasing the number of times we need to check whether graphs are isomorphic. However, if we choose the graphlet incorrectly, the time cost of finding the graphlets might result in slower performance.

We also examined the number of partial automorphisms for all graphs with less than 10 vertices. Our initial results might serve as a starting ramp in determining if we can approximate the number of partial automorphisms of a graph based on its structure. This would be useful in predicting how long it would take to find all partial automorphisms for a given graph.

We also introduced the definition of *asymmetric depth*. Asymmetric depth might be used to measure the asymmetricity of graphs, determining whether a given graph is more or less asymmetric than other graphs. A graph with n vertices and maximal asymmetric depth d is more asymmetric than other graphs with n vertices and maximal asymmetric depth $d-1$. We were able to find graphs with maximal asymmetric depth 4. Research in this area should focus on answering the question: *What is the maximal asymmetric depth any graph with n vertices can have?*

Bibliography

[axi] What is axios? `https://axios-http.com/docs/intro`. Accessed: 2023-05-02.

[Bab15] László Babai. Graph isomorphism in quasipolynomial time. *CoRR*, abs/1512.03547, 2015.

[CFSV04] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[dic] Time complexity of python dictionary. `https://wiki.python.org/moin/TimeComplexity#dict`. Accessed: 2023-05-01.

[dja] Django, python web framework. `https://docs.djangoproject.com/en/4.2/`. Accessed: 2023-05-02.

[drf] Django rest framework. `https://www.django-rest-framework.org/`. Accessed: 2023-05-02.

[Dub22] Simona Dubeková. Asymmetric graphs. Bachelor thesis, Comenius University in Bratislava, 2022.

[ec2] Amazon ec2 c5 instances. `https://aws.amazon.com/ec2/instance-types/c5/`. Accessed: 2023-05-01.

[EMP+20] Rafael Espejo, Guillermo Mestre, Fernando Postigo, Sara Lumbreras, Andrés Ramos, Tao Huang, and Ettore Francesco Bompard. Exploiting graphlet decomposition to explain the structure of complex networks: the ghust framework. *Scientific Reports*, 10, 2020.

[Gal12] Joseph A. Gallian. *Contemporary Abstract Algebra*. Brooks/Cole Cengage Learning, 8 edition, 2012.

[gap] Gap - groups, algorithms, programming - a system for computational discrete algebra. `https://www.gap-system.org/`. Accessed: 2023-05-01.

[GKSF21] Zahra Gharaee, Shreyas Kowshik, Oliver Stromann, and Michael Felsberg. Graph representation learning for road type classification. *Pattern Recognition*, 120:108174, 2021.

[ite] Python itertools combinations. `https://docs.python.org/3/library/itertools.html#itertools.combinations`. Accessed: 2023-05-01.

[Jaj19] Tatiana Jajcayová. Representations of permutation groups and semigroups on combinatiorial structures. *Fifth Russian Finnish Symposium on Discrete Mathematics.*, 5:137–145, 2019.

[Jaj22] Tatiana B. Jajcayová. On computational aspects of finding inverse monoids of partial automorphisms (logic, algebraic system,

language and related areas in computer science). *Logic, Algebraic system, Language and Related Areas in Computer Science*, 2229:88–96, 2022.

[JJSS21] Robert Jajcay, Tatiana Jajcayová, Nóra Szakács, and Mária B. Szendrei. Inverse monoids of partial graph automorphisms. *Journal of Algebraic Combinatorics*, 53(3):831–851, 2021.

[JM18] Alpár Jüttner and Péter Madarasi. Vf2++—an improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, 242:69–81, 2018. Computational Advances in Combinatorial Optimization.

[Law98] Mark V. Lawson. *Inverse Semigroups: The Theore of Parial Symmetries*. World Scientific Publishing Company, 1998.

[mck] Collection of graphs. `http://users.cecs.anu.edu.au/~bdm/data/graphs.html`. Accessed: 2023-05-01.

[Mér19] Mária Mériová. Symmetries of combinatorial structures. Diploma thesis, Comenius University in Bratislava, 2019.

[nau] Nauty. `https://pallini.di.uniroma1.it/Guide.html`. Accessed: 2023-05-01.

[net] Networkx, network analysis in python. `https://networkx.org/documentation/stable/index.html#`. Accessed: 2023-05-01.

[oeia] Number of asymmetric (not necessarily connected) graphs with n nodes. `https://oeis.org/A003400`. Accessed: 2023-05-01.

[oeib] Number of graphs on n unlabeled nodes. `https://oeis.org/A000088`. Accessed: 2023-05-01.

[par] Number of partial permutations of an n-set. `https://oeis.org/A002720`. Accessed: 2023-05-01.

[PE63] Alfréd Rényi Paul Erdős. Asymmetric graphs. *Acta Mathematica Academiae Scientiarum Hungarica*, 14:295–315, 1963.

[pin] Pinia, the intuitive store for vue.js. `https://pinia.vuejs.org/`. Accessed: 2023-05-02.

[pyp] Pypy. `https://www.pypy.org/features.html`. Accessed: 2023-05-01.

[Slá21] Michal Slávik. Partial symmetries of graphs. Bachelor thesis, Comenius University in Bratislava, 2021.

[SS17] Pascal Schweitzer and Patrick Schweitzer. Minimal asymmetric graphs. *Journal of Combinatorial Theory, Series B*, 127:215–227, 2017.

[Tuc12] Alan Tucker. *Applied Combinatorics*. John Wiley & Sons, 6 edition, 2012.

[vuea] Vue.js, the progressive javascript framework. `https://vuejs.org/`. Accessed: 2023-05-02.

[vueb] Vuetify, vue component framework. `https://vuetifyjs.com/en/`. Accessed: 2023-05-02.

[Wes01] Douglas B. West. *Introduction to Graph Theory*. Pearson Education, 2 edition, 2001.

[ZTWX11] Baida Zhang, Yuhua Tang, Junjie Wu, and Shuai Xu. Ld: A polynomial time algorithm for tree isomorphism. *Procedia Engineering*, 15:2015–2020, 2011. CEIS 2011.

Appendix

We use the following notation for groups:

- D_n - dihedral group of order n,
- S_n - symmetric group of degree n,
- C_n - cyclic group of order n,
- GxH - direct product of groups G and H

A.1 Partial automorphism monoid of minimal asymmetric graph X1

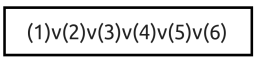


Figure A.1: $\mathcal{D}$-class of induced subgraph with 6 vertices and 6 edges.

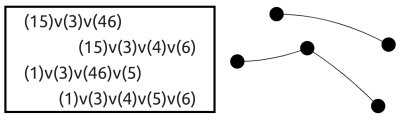


Figure A.2: $\mathcal{D}$-class of induced subgraph with 5 vertices and 3 edges (group C2xC2).

(16)v(24)v(5)
(1)v(2)v(4)v(5)v(6)

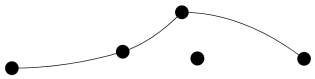


Figure A.3: $\mathcal{D}$-class of induced subgraph with 5 vertices and 3 edges (group C2).

(2)v(34)v(56)
(2)v(3)v(4)v(5)v(6)

Figure A.4: $\mathcal{D}$-class of induced subgraph with 5 vertices and 4 edges (group C2).

(12)v(3)v(5)v(6)
(1)v(2)v(3)v(5)v(6)

Figure A.5: $\mathcal{D}$-class of induced subgraph with 5 vertices and 4 edges (group C2).

(1)v(23)v(45)
(1)v(2)v(3)v(4)v(5)

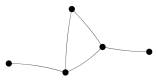


Figure A.6: $\mathcal{D}$-class of induced subgraph with 5 vertices and 5 edges (group C2).

(13)v(2)v(4)v(6)
(1)v(2)v(3)v(4)v(6)

Figure A.7: $\mathcal{D}$-class of induced subgraph with 5 vertices and 5 edges (group C2).

(1)v(2)v(5)v(6) (1)v(2)v(56) (12)v(5)v(6) (12)v(56)	[26514] [2614)v(5) [24)v(156] [24)v(16)v(5)
[41562) [4162)v(5) [42)v(165] [42)v(16)v(5)	(1)v(4)v(5)v(6) (15)v(4)v(6) (1)v(46)v(5) (15)v(46)

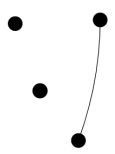


Figure A.8: $\mathcal{D}$-class of induced subgraph with 4 vertices and 1 edge (subgroup C2xC2).

(1)v(3)v(4)v(6) (1)v(3)v(46) (13)v(4)v(6) (13)v(46) (14)v(36) (1436) (1634) (16)v(34)	[135)v(4)v(6) [135)v(46) [15)v(3)v(4)v(6) [15)v(3)v(46) [14365) [165)v(34) [145)v(36) [16345)
[531)v(4)v(6) [531)v(46) [541)v(36) [56341) [51)v(3)v(46) [51)v(3)v(4)v(6) [561)v(34) [54361)	(3)v(4)v(5)v(6) (3)v(46)v(5) (3456) (34)v(56) (35)v(46) (35)v(4)v(6) (3654) (36)v(45)

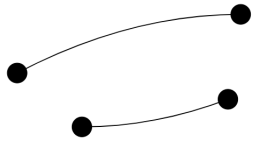


Figure A.9: $\mathcal{D}$-class of induced subgraph with 4 vertices and 2 edges (subgroup D8).

(1)v(2)v(4)v(5) (14)v(2)v(5)	[23)v(1)v(45) [23)v(145)	[23)v[456)v(1) [23)v[4156)	[123)v[456) [423)v[156)	[1246)v(5) [16)v(24)v(5)
[32)v(1)v(45) [32)v(154)	(1)v(3)v(4)v(5) (15)v(3)v(4)	[46)v(1)v(3)v(5) [46)v(15)v(3)	[12)v[46)v(3)v(5) [152)v[46)v(3)	[12)v[3456) [3452)v[16)
[32)v[654)v(1) [32)v[6514)	[64)v(1)v(3)v(5) [64)v(15)v(3)	(1)v(3)v(5)v(6) (15)v(3)v(6)	[12)v(3)v(5)v(6) [152)v(3)v(6)	[34)v[12)v(56) [34)v[1652)
[321)v[654) [651)v[324)	[21)v[64)v(3)v(5) [251)v[64)v(3)	[21)v(3)v(5)v(6) [251)v(3)v(6)	(2)v(3)v(5)v(6) (25)v(3)v(6)	[34)v(2)v(56) [34)v(256)
[6421)v(5) [61)v(24)v(5)	[6543)v[21) [2543)v[61)	[43)v[21)v(56) [43)v[2561)	[43)v(2)v(56) [43)v(265)	(2)v(4)v(5)v(6) (26)v(4)v(5)

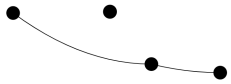


Figure A.10: $\mathcal{D}$-class of induced subgraph with 4 vertices and 2 edges (subgroup C2).

(13)v(2)v(6) (132)v(6) (123)v(6) (12)v(3)v(6) (1)v(23)v(6) (1)v(2)v(3)v(6)

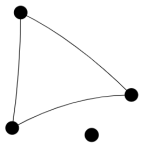


Figure A.11: $\mathcal{D}$-class of induced subgraph with 4 vertices and 3 edges (group S3).

(16)v(24) (1)v(2)v(4)v(6)	[143)v[65)v(2) [6423)v[15)	[13)v(2)v(4)v(6) [163)v(24)
[341)v[56)v(2) [51)v[3246)	(23)v(45) (2)v(3)v(4)v(5)	[56)v(2)v(34) [53246)
[31)v(2)v(4)v(6) [361)v(24)	[65)v(2)v(34) [64235)	(24)v(36) (2)v(3)v(4)v(6)

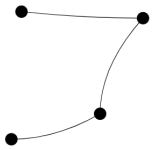


Figure A.12: $\mathcal{D}$-class of induced subgraph with 4 vertices and 3 edges (subgroup C2).

(1)v(2)v(3)v(4) (13)v(2)v(4)	[45)v(1)v(23) [45)v(132)
[54)v(1)v(23) [54)v(123)	(1)v(2)v(3)v(5) (12)v(3)v(5)

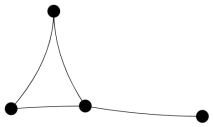


Figure A.13: $\mathcal{D}$-class of induced subgraph with 4 vertices and 4 edges (subgroup C2).

(15)v(4) (154) (145) (14)v(5) (1)v(45) (1)v(4)v(5)	[456)v(1) [46)v(1)v(5) [4156) [416)v(5) [46)v(15) [4516)	[12)v[456) [12)v[46)v(5) [42)v[156) [42)v[16)v(5) [152)v[46) [452)v[16)
[654)v(1) [64)v(1)v(5) [6514) [64)v(15) [614)v(5) [6154)	(16)v(5) (165) (156) (15)v(6) (1)v(56) (1)v(5)v(6)	[12)v(5)v(6) [12)v(56) [152)v(6) [1652) [1562) [162)v(5)
[21)v[654) [21)v[64)v(5) [651)v[24) [251)v[64) [61)v[24)v(5) [61)v[254)	[21)v(5)v(6) [21)v(56) [251)v(6) [2651) [2561) [261)v(5)	(26)v(5) (265) (256) (25)v(6) (2)v(56) (2)v(5)v(6)

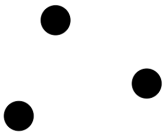


Figure A.14: $\mathcal{D}$-class of induced subgraph with 3 vertices and 0 edges (subgroup S3).

(1)v(2)v(5) (12)v(5)	[56)v(1)v(2) [56)v(12)	[23)v[54)v(1) [213)v[54)	[23)v[56)v(1) [213)v[56)	[514)v[26) [24)v[516)	[123)v[56) [13)v[56)v(2)	[124)v(5) [14)v(2)v(5)	[13)v[254) [23)v[154)	[14)v[26)v[53) [24)v[16)v[53)	[13)v[256) [23)v[156)	[14)v[26)v(5) [24)v[16)v(5)
[65)v(1)v(2) [65)v(12)	(1)v(2)v(6) (12)v(6)	[23)v[64)v(1) [213)v[64)	[23)v(1)v(6) [213)v(6)	[2614) [24)v(16)	[123)v(6) [13)v(2)v(6)	[124)v[65) [14)v[65)v(2)	[13)v[25)v[64) [23)v[15)v[64)	[14)v[263) [24)v[163)	[13)v[25)v(6) [23)v[15)v(6)	[14)v[265) [24)v[165)
[32)v[45)v(1) [312)v[45)	[32)v[46)v(1) [312)v[46)	(1)v(3)v(4) (13)v(4)	[46)v(1)v(3) [46)v(13)	[36)v(14) [3416)	[12)v[46)v(3) [132)v[46)	[12)v[345) [32)v[145)	[135)v(4) [15)v(3)v(4)	[1436) [16)v(34)	[135)v[46) [15)v[46)v(3)	[36)v[145) [16)v[345)
[32)v[65)v(1) [312)v[65)	[32)v(1)v(6) [312)v(6)	[64)v(1)v(3) [64)v(13)	(1)v(3)v(6) (13)v(6)	[3614) [34)v(16)	[12)v(3)v(6) [132)v(6)	[12)v[34)v[65) [32)v[14)v[65)	[135)v[64) [15)v[64)v(3)	[14)v(36) [1634)	[135)v(6) [15)v(3)v(6)	[14)v[365) [34)v[165)
[62)v[415) [42)v[615)	[4162) [42)v(16)	[63)v(14) [6143)	[4163) [43)v(16)	(1)v(4)v(6) (1)v(46)	[42)v[163) [162)v[43)	[642)v[15) [62)v[15)v(4)	[143)v[65) [63)v[145)	[13)v(4)v(6) [13)v(46)	[43)v[165) [163)v[45)	[15)v(4)v(6) [15)v(46)
[321)v[65) [31)v[65)v(2)	[321)v(6) [31)v(2)v(6)	[21)v[64)v(3) [231)v[64)	[21)v(3)v(6) [231)v(6)	[24)v[361) [34)v[261)	(2)v(3)v(6) (23)v(6)	[34)v[65)v(2) [324)v[65)	[235)v[64) [25)v[64)v(3)	[24)v(36) [2634)	[235)v(6) [25)v(3)v(6)	[24)v[365) [34)v[265)
[421)v(5) [41)v(2)v(5)	[421)v[56) [41)v[56)v(2)	[21)v[543) [541)v[23)	[21)v[43)v[56) [41)v[23)v[56)	[246)v[51) [26)v[51)v(4)	[43)v[56)v(2) [423)v[56)	(2)v(4)v(5) (24)v(5)	[23)v(45) [2543)	[246)v[53) [26)v[53)v(4)	[23)v[456) [43)v[256)	[246)v(5) [26)v(4)v(5)
[31)v[452) [451)v[32)	[31)v[52)v[46) [51)v[32)v[46)	[531)v(4) [51)v(3)v(4)	[531)v[46) [51)v[46)v(3)	[56)v[341) [36)v[541)	[532)v[46) [52)v[46)v(3)	[32)v(45) [3452)	(3)v(4)v(5) (35)v(4)	[56)v(34) [5436)	[46)v(3)v(5) [46)v(35)	[3456) [36)v(45)
[41)v[62)v[35) [61)v[42)v[35)	[41)v[362) [361)v[42)	[6341) [61)v(34)	[41)v(36) [4361)	[31)v(4)v(6) [31)v(46)	[42)v(36) [4362)	[642)v[35) [62)v[35)v(4)	[65)v(34) [6345)	(3)v(4)v(6) (3)v(46)	[4365) [45)v(36)	[35)v(4)v(6) [35)v(46)
[31)v[652) [651)v[32)	[31)v[52)v(6) [51)v[32)v(6)	[531)v[64) [51)v[64)v(3)	[531)v(6) [51)v(3)v(6)	[34)v[561) [54)v[361)	[532)v(6) [52)v(3)v(6)	[32)v[654) [652)v[34)	[64)v(3)v(5) [64)v(35)	[5634) [54)v(36)	(3)v(5)v(6) (35)v(6)	[34)v(56) [3654)
[41)v[62)v(5) [61)v[42)v(5)	[41)v[562) [561)v[42)	[541)v[63) [61)v[543)	[41)v[563) [561)v[43)	[51)v(4)v(6) [51)v(46)	[42)v[563) [562)v[43)	[642)v(5) [62)v(4)v(5)	[6543) [63)v(45)	[53)v(4)v(6) [53)v(46)	[43)v(56) [4563)	(4)v(5)v(6) (46)v(5)

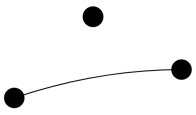


Figure A.15: $\mathcal{D}$-class of induced subgraph with 3 vertices and 1 edge (subgroup C2).

(14)v(2) (1)v(2)v(4)	[23)v[45)v(1) [23)v[415)	[13)v(2)v(4) [143)v(2)	[123)v[45) [423)v[15)	[1246) [16)v(24)
[32)v[54)v(1) [32)v[514)	(15)v(3) (1)v(3)v(5)	[132)v[54) [532)v[14)	[12)v(3)v(5) [152)v(3)	[12)v[34)v[56) [52)v[34)v[16)
[31)v(2)v(4) [341)v(2)	[231)v[45) [41)v[235)	(2)v(34) (2)v(3)v(4)	[45)v(23) [4235)	[3246) [36)v(24)
[321)v[54) [51)v[324)	[21)v(3)v(5) [251)v(3)	[54)v(23) [5324)	(25)v(3) (2)v(3)v(5)	[34)v[56)v(2) [34)v[526)
[6421) [61)v(24)	[21)v[43)v[65) [61)v[43)v[25)	[6423) [63)v(24)	[43)v[65)v(2) [43)v[625)	(26)v(4) (2)v(4)v(6)

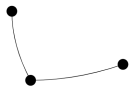


Figure A.16: $\mathcal{D}$-class of induced subgraph with 3 vertices and 2 edges (subgroup C2).

(13)v(2) (132) (123) (12)v(3) (1)v(23) (1)v(2)v(3)

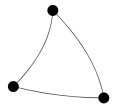


Figure A.17: $\mathcal{D}$-class of induced subgraph with 3 vertices and 3 edges (group S3).

(14) (1)v(4)	[45)v(1) [415)	[46)v(1) [416)	[12)v[45) [42)v[15)	[12)v[46) [42)v[16)	[13)v(4) [143)	[13)v[46) [43)v[16)	[145) [15)v(4)	[15)v[46) [45)v[16)
[54)v(1) [514)	(15) (1)v(5)	[56)v(1) [516)	[12)v(5) [152)	[12)v[56) [52)v[16)	[13)v[54) [53)v[14)	[13)v[56) [53)v[16)	[14)v(5) [154)	[156) [16)v(5)
[64)v(1) [614)	[65)v(1) [615)	(16) (1)v(6)	[12)v[65) [62)v[15)	[12)v(6) [162)	[13)v[64) [63)v[14)	[13)v(6) [163)	[14)v[65) [64)v[15)	[15)v(6) [165)
[21)v[54) [51)v[24)	[21)v(5) [251)	[21)v[56) [51)v[26)	(25) (2)v(5)	[56)v(2) [526)	[23)v[54) [53)v[24)	[23)v[56) [53)v[26)	[24)v(5) [254)	[256) [26)v(5)
[21)v[64) [61)v[24)	[21)v[65) [61)v[25)	[21)v(6) [261)	[65)v(2) [625)	(26) (2)v(6)	[23)v[64) [63)v[24)	[23)v(6) [263)	[24)v[65) [64)v[25)	[25)v(6) [265)
[31)v(4) [341)	[31)v[45) [41)v[35)	[31)v[46) [41)v[36)	[32)v[45) [42)v[35)	[32)v[46) [42)v[36)	(34) (3)v(4)	[46)v(3) [436)	[345) [35)v(4)	[35)v[46) [45)v[36)
[31)v[64) [61)v[34)	[31)v[65) [61)v[35)	[31)v(6) [361)	[32)v[65) [62)v[35)	[32)v(6) [362)	[64)v(3) [634)	(36) (3)v(6)	[34)v[65) [64)v[35)	[35)v(6) [365)
[541) [51)v(4)	[41)v(5) [451)	[41)v[56) [51)v[46)	[42)v(5) [452)	[42)v[56) [52)v[46)	[543) [53)v(4)	[43)v[56) [53)v[46)	(45) (4)v(5)	[456) [46)v(5)
[51)v[64) [61)v[54)	[651) [61)v(5)	[51)v(6) [561)	[652) [62)v(5)	[52)v(6) [562)	[53)v[64) [63)v[54)	[53)v(6) [563)	[654) [64)v(5)	(56) (5)v(6)

Figure A.18: $\mathcal{D}$-class of induced subgraph with 2 vertices and 0 edges (subgroup C2).

(12) (1)v(2)	[23)v(1) [213)	[123) [13)v(2)	[124) [14)v(2)	[13)v[25) [23)v[15)	[14)v[26) [24)v[16)
[32)v(1) [312)	(13) (1)v(3)	[12)v(3) [132)	[12)v[34) [32)v[14)	[135) [15)v(3)	[14)v[36) [34)v[16)
[321) [31)v(2)	[21)v(3) [231)	(23) (2)v(3)	[34)v(2) [324)	[235) [25)v(3)	[24)v[36) [34)v[26)
[421) [41)v(2)	[21)v[43) [41)v[23)	[43)v(2) [423)	(24) (2)v(4)	[23)v[45) [43)v[25)	[246) [26)v(4)
[31)v[52) [51)v[32)	[531) [51)v(3)	[532) [52)v(3)	[32)v[54) [52)v[34)	(35) (3)v(5)	[34)v[56) [54)v[36)
[41)v[62) [61)v[42)	[41)v[63) [61)v[43)	[42)v[63) [62)v[43)	[642) [62)v(4)	[43)v[65) [63)v[45)	(46) (4)v(6)

Figure A.19: $\mathcal{D}$-class of induced subgraph with 2 vertices and 1 edge (subgroup C2).

(1)	[12)	[13)	[14)	[15)	[16)
[21)	(2)	[23)	[24)	[25)	[26)
[31)	[32)	(3)	[34)	[35)	[36)
[41)	[42)	[43)	(4)	[45)	[46)
[51)	[52)	[53)	[54)	(5)	[56)
[61)	[62)	[63)	[64)	[65)	(6)

Figure A.20: $\mathcal{D}$-class of induced subgraph with 1 vertex and 0 edges.

A.2 Partial automorphism monoid of minimal asymmetric graph X9

(1)v(2)v(3)v(4)v(5)v(6)v(7)

Figure A.21: $\mathcal{D}$-class of induced subgraph with 7 vertices and 6 edges.

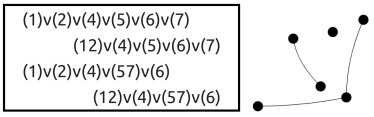


Figure A.22: $\mathcal{D}$-class of induced subgraph with 6 vertices and 3 edges (group C2xC2).

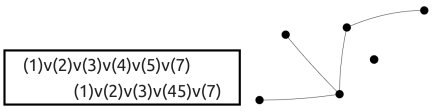


Figure A.23: $\mathcal{D}$-class of induced subgraph with 6 vertices and 4 edges (group C2).

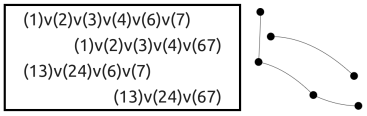


Figure A.24: $\mathcal{D}$-class of induced subgraph with 6 vertices and 4 edges (group C2xC2).

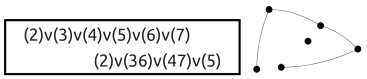


Figure A.25: $\mathcal{D}$-class of induced subgraph with 6 vertices and 4 edges (group C2).

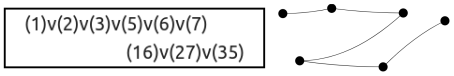


Figure A.26: $\mathcal{D}$-class of induced subgraph with 6 vertices and 5 edges (group C2).

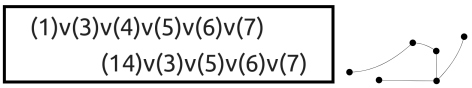


Figure A.27: $\mathcal{D}$-class of induced subgraph with 6 vertices and 5 edges (group C2).

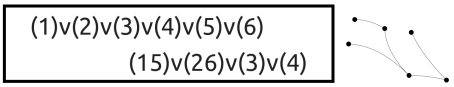


Figure A.28: $\mathcal{D}$-class of induced subgraph with 6 vertices and 5 edges (group C2).

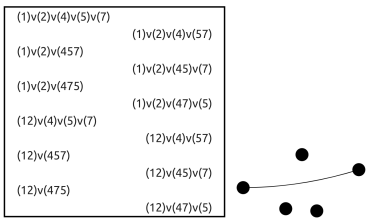


Figure A.29: $\mathcal{D}$-class of induced subgraph with 5 vertices and 1 edge (group D12).

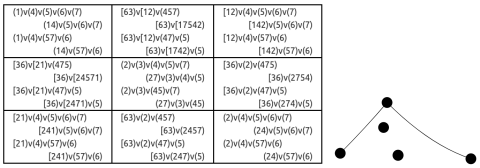


Figure A.30: $\mathcal{D}$-class of induced subgraph with 5 vertices and 2 edges (subgroup C2xC2).

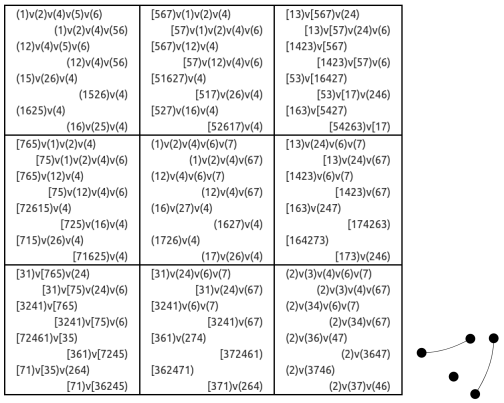


Figure A.31: $\mathcal{D}$-class of induced subgraph with 5 vertices and 2 edges (subgroup C2).

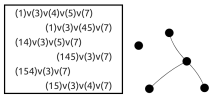


Figure A.32: $\mathcal{D}$-class of induced subgraph with 5 vertices and 3 edges (group S3).

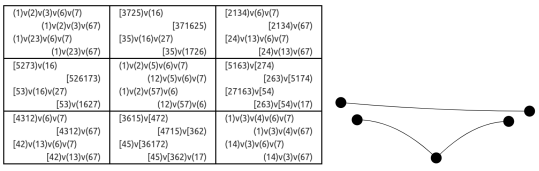


Figure A.33: $\mathcal{D}$-class of induced subgraph with 5 vertices and 3 edges (subgroup C2xC2).

(1)v(2)v(3)v(4)v(6) (13)v(24)v(6)	[67)v(1)v(2)v(3)v(4) [67)v(13)v(24)	[45)v[67)v(1)v(2)v(3) [425)v[67)v(13)	[135)v(264) [15)v(26)v(3)v(4)	[15)v[47)v(263) [435)v[1627)
[76)v(1)v(2)v(3)v(4) [76)v(13)v(24)	(1)v(2)v(3)v(4)v(7) (13)v(24)v(7)	[45)v(1)v(2)v(3)v(7) [425)v(13)v(7)	[135)v[7246) [15)v[726)v(3)v(4)	[15)v[47236) [435)v[16)v(27)
[54)v[76)v(1)v(2)v(3) [524)v[76)v(13)	[54)v(1)v(2)v(3)v(7) [524)v(13)v(7)	(1)v(2)v(3)v(5)v(7) (13)v(25)v(7)	[724)v[1356) [154)v[726)v(3)	[157236) [16)v(27)v(35)
[531)v(246) [51)v(26)v(3)v(4)	[531)v[6427) [51)v[627)v(3)v(4)	[6531)v[427) [451)v[627)v(3)	(2)v(3)v(4)v(5)v(6) (2)v(35)v(46)	[43567)v(2) [47)v(2)v(36)v(5)
[51)v[74)v(236) [7261)v[534)	[51)v[63274) [61)v[534)v(27)	[632751) [61)v(27)v(35)	[76534)v(2) [74)v(2)v(36)v(5)	(2)v(3)v(5)v(6)v(7) (2)v(37)v(56)

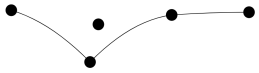


Figure A.34: $\mathcal{D}$-class of induced subgraph with 5 vertices and 3 edges (subgroup C2).

(1)v(2)v(3)v(5)v(6) (15)v(26)v(3)	[213567) [27)v(16)v(35)	[24)v[13567) [164)v[27)v(35)
[765312) [72)v(16)v(35)	(1)v(3)v(5)v(6)v(7) (17)v(36)v(5)	[14)v(3)v(5)v(6)v(7) [174)v(36)v(5)
[76531)v[42) [461)v[72)v(35)	[41)v(3)v(5)v(6)v(7) [471)v(36)v(5)	(3)v(4)v(5)v(6)v(7) (36)v(47)v(5)

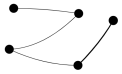


Figure A.35: $\mathcal{D}$-class of induced subgraph with 5 vertices and 4 edges (subgroup C2).

(1)v(2)v(3)v(4)v(5) (1)v(2)v(3)v(45)	[26)v(145)v(3) [26)v(15)v(3)v(4)
[62)v(154)v(3) [62)v(15)v(3)v(4)	(1)v(3)v(4)v(5)v(6) (14)v(3)v(5)v(6)

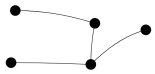


Figure A.36: $\mathcal{D}$-class of induced subgraph with 5 vertices and 4 edges (subgroup C2).

(1)v(4)v(5)v(7) (1)v(4)v(57) (1)v(457) (1)v(45)v(7) (1)v(475) (1)v(47)v(5) (1457) (145)v(7) (1475) (147)v(5) (14)v(5)v(7) (14)v(57) (15)v(47) (1547) (154)v(7) (1574) (157)v(4) (15)v(4)v(7) (1754) (174)v(5) (17)v(4)v(5) (175)v(4) (1745) (17)v(45)	[12)v(4)v(5)v(7) [12)v(4)v(57) [12)v(457) [12)v(45)v(7) [12)v(475) [12)v(47)v(5) [17542) [1542)v(7) [15742) [1742)v(5) [142)v(5)v(7) [142)v(57) [152)v(47) [17452) [1452)v(7) [14752) [1752)v(4) [152)v(4)v(7) [14572) [1472)v(5) [172)v(4)v(5) [1572)v(4) [15472) [172)v(45)
[21)v(4)v(5)v(7) [21)v(4)v(57) [21)v(457) [21)v(45)v(7) [21)v(475) [21)v(47)v(5) [27541) [2541)v(7) [25741) [2741)v(5) [241)v(5)v(7) [241)v(57) [251)v(47) [27451) [2451)v(7) [24751) [2751)v(4) [251)v(4)v(7) [24571) [2471)v(5) [271)v(4)v(5) [2571)v(4) [25471) [271)v(45)	(2)v(4)v(5)v(7) (2)v(4)v(57) (2)v(457) (2)v(45)v(7) (2)v(475) (2)v(47)v(5) (2457) (245)v(7) (2475) (247)v(5) (24)v(5)v(7) (24)v(57) (25)v(47) (2547) (254)v(7) (2574) (257)v(4) (25)v(4)v(7) (2754) (274)v(5) (27)v(4)v(5) (275)v(4) (2745) (27)v(45)

Figure A.37: $\mathcal{D}$-class of induced subgraph with 4 vertices and 0 edges (subgroup S4).

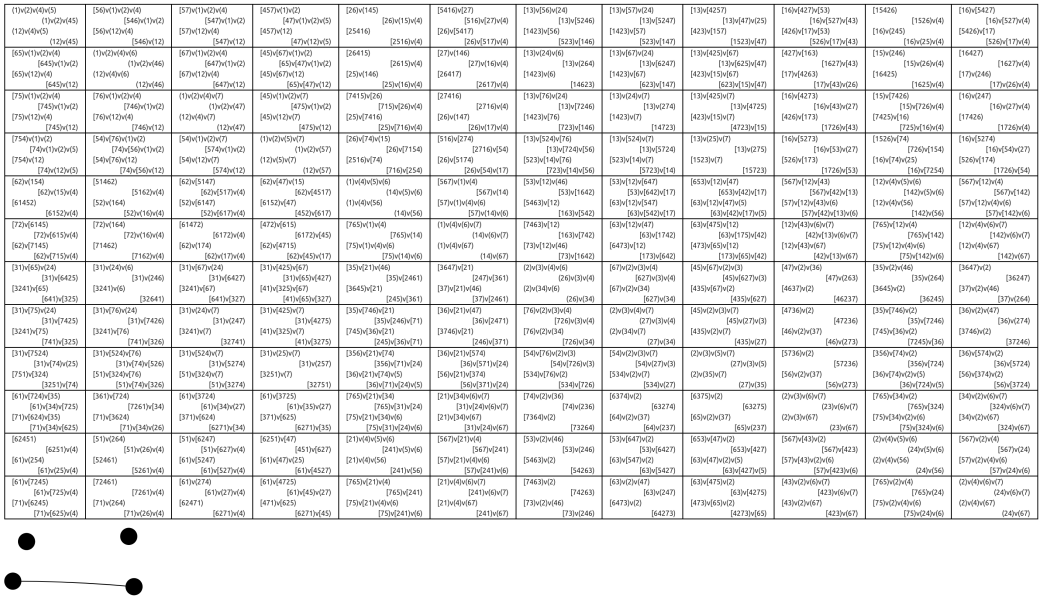


Figure A.38: $\mathcal{D}$-class of induced subgraph with 4 vertices and 1 edge (subgroup C2xC2).

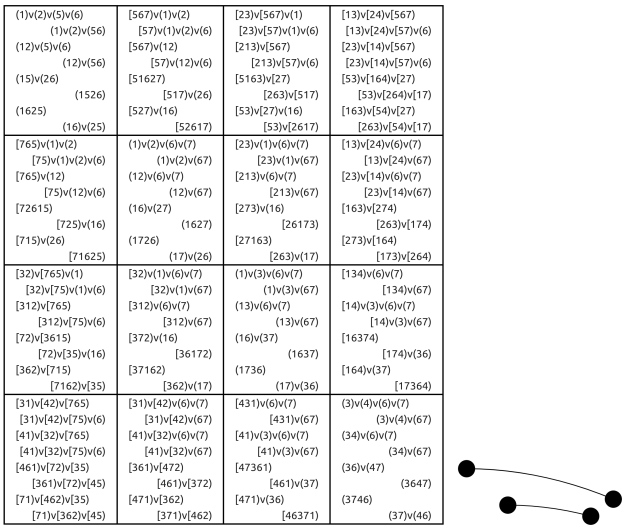


Figure A.39: $\mathcal{D}$-class of induced subgraph with 4 vertices and 2 edges (subgroup D8).

(1)v(2)v(3)v(6) (1)v(23)v(6)	[67)v(1)v(2)v(3) [67)v(1)v(23)	[2134)v(6) [24)v(13)v(6)	[2134)v[67) [24)v[67)v(13)	[2135)v[67) [25)v[67)v(13)	[25)v[37)v(16) [35)v[27)v(16)	[624)v[135) [134)v[625)	[15)v(263) [15)v(26)v(3)	[1625)v[37) [35)v[1627)	[24)v[135)v[67) [134)v[25)v[67)	[25)v[37)v[164) [35)v[27)v[164)
[76)v(1)v(2)v(3) [76)v(1)v(23)	(1)v(2)v(3)v(7) (1)v(23)v(7)	[2134)v[76) [24)v[76)v(13)	[2134)v(7) [24)v(13)v(7)	[2135)v(7) [25)v(13)v(7)	[3716)v[25) [2716)v[35)	[724)v[135) [134)v[725)	[15)v[7236) [15)v[726)v(3)	[16)v[3725) [16)v[35)v(27)	[24)v[135)v(7) [134)v[25)v(7)	[16)v[25)v[374) [16)v[35)v[274)
[4312)v(6) [42)v(13)v(6)	[4312)v[67) [42)v[67)v(13)	(1)v(3)v(4)v(6) (14)v(3)v(6)	[67)v(1)v(3)v(4) [67)v(14)v(3)	[45)v[67)v(1)v(3) [415)v[67)v(3)	[3615)v[47) [45)v[3617)	[145)v(62)v(3) [15)v[62)v(3)v(4)	[135)v[462) [435)v[162)	[15)v[47)v[362) [45)v[17)v[362)	[145)v[67)v(3) [15)v[67)v(3)v(4)	[15)v[3647) [3645)v[17)
[4312)v[76) [42)v[76)v(13)	[4312)v(7) [42)v(13)v(7)	[76)v(1)v(3)v(4) [76)v(14)v(3)	(1)v(3)v(4)v(7) (14)v(3)v(7)	[45)v(1)v(3)v(7) [415)v(3)v(7)	[36)v[4715) [36)v[45)v(17)	[145)v[72)v(3) [15)v[72)v(3)v(4)	[135)v[46)v[72) [435)v[16)v[72)	[36)v[15)v[472) [36)v[45)v[172)	[145)v(3)v(7) [15)v(3)v(4)v(7)	[36)v[15)v(47) [36)v[1745)
[5312)v[76) [52)v[76)v(13)	[5312)v(7) [52)v(13)v(7)	[54)v[76)v(1)v(3) [514)v[76)v(3)	[54)v(1)v(3)v(7) [514)v(3)v(7)	(1)v(3)v(5)v(7) (15)v(3)v(7)	[36)v(175) [36)v(17)v(5)	[14)v[72)v(3)v(5) [154)v[72)v(3)	[1356)v[72) [16)v[72)v(35)	[36)v[1572) [36)v[172)v(5)	[14)v(3)v(5)v(7) [154)v(3)v(7)	[36)v[1574) [36)v[174)v(5)
[52)v[73)v(16) [72)v[53)v(16)	[52)v[6173) [6172)v[53)	[5163)v[74) [7163)v[54)	[63)v[5174) [63)v[54)v(17)	[63)v(157) [63)v(17)v(5)	(1)v(5)v(6)v(7) (1)v(57)v(6)	[63)v[754)v[12) [63)v[74)v[12)v(5)	[7653)v[12) [73)v[12)v(56)	[12)v(5)v(6)v(7) [12)v(57)v(6)	[63)v[1754) [63)v[174)v(5)	[14)v(5)v(6)v(7) [14)v(57)v(6)
[531)v[426) [431)v[526)	[531)v[427) [431)v[527)	[541)v[26)v(3) [51)v[26)v(3)v(4)	[541)v[27)v(3) [51)v[27)v(3)v(4)	[41)v[27)v(3)v(5) [451)v[27)v(3)	[36)v[457)v[21) [36)v[47)v[21)v(5)	(2)v(3)v(4)v(5) (2)v(3)v(45)	[4356)v(2) [46)v(2)v(35)	[36)v[457)v(2) [36)v[47)v(2)v(5)	[27)v(3)v(4)v(5) [27)v(3)v(45)	[36)v[2457) [36)v[247)v(5)
[51)v(236) [51)v(26)v(3)	[51)v[6327) [51)v[627)v(3)	[531)v[264) [261)v[534)	[531)v[64)v[27) [61)v[534)v[27)	[6531)v[27) [61)v[27)v(35)	[3567)v[21) [37)v[21)v(56)	[6534)v(2) [64)v(2)v(35)	(2)v(3)v(5)v(6) (2)v(36)v(5)	[3567)v(2) [37)v(2)v(56)	[6534)v[27) [64)v[27)v(35)	[3567)v[24) [37)v[24)v(56)
[5261)v[73) [7261)v[53)	[61)v[5273) [61)v[53)v(27)	[263)v[51)v[74) [263)v[71)v[54)	[63)v[51)v[274) [63)v[271)v[54)	[63)v[2751) [63)v[271)v(5)	[21)v(5)v(6)v(7) [21)v(57)v(6)	[63)v[754)v(2) [63)v[74)v(2)v(5)	[7653)v(2) [73)v(2)v(56)	(2)v(5)v(6)v(7) (2)v(57)v(6)	[63)v[2754) [63)v[274)v(5)	[24)v(5)v(6)v(7) [24)v(57)v(6)
[531)v[42)v[76) [431)v[52)v[76)	[531)v[42)v(7) [431)v[52)v(7)	[541)v[76)v(3) [51)v[76)v(3)v(4)	[541)v(3)v(7) [51)v(3)v(4)v(7)	[41)v(3)v(5)v(7) [451)v(3)v(7)	[36)v[4571) [36)v[471)v(5)	[72)v(3)v(4)v(5) [72)v(3)v(45)	[4356)v[72) [46)v[72)v(35)	[36)v[4572) [36)v[472)v(5)	(3)v(4)v(5)v(7) (3)v(45)v(7)	[36)v(475) [36)v(47)v(5)
[461)v[52)v[73) [461)v[72)v(53)	[61)v[52)v[473) [61)v[472)v(53)	[7463)v[51) [5463)v[71)	[63)v[51)v(47) [63)v[5471)	[63)v[4751) [63)v[471)v(5)	[41)v(5)v(6)v(7) [41)v(57)v(6)	[63)v[7542) [63)v[742)v(5)	[7653)v[42) [73)v[42)v(56)	[42)v(5)v(6)v(7) [42)v(57)v(6)	[63)v(457) [63)v(47)v(5)	(4)v(5)v(6)v(7) (4)v(57)v(6)

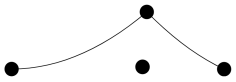


Figure A.40: $\mathcal{D}$-class of induced subgraph with 4 vertices and 2 edges (subgroup C2).

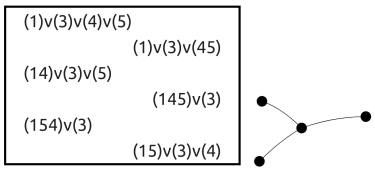


Figure A.41: $\mathcal{D}$-class of induced subgraph with 4 vertices and 3 edges (group S3).

(1)v(2)v(3)v(4) (13)v(24)	[45)v(1)v(2)v(3) [425)v(13)	[2135)v(46) [415)v[26)v(3)	[135)v[246) [15)v[26)v(3)v(4)	[15)v[236)v[47) [435)v[16)v[27)
[54)v(1)v(2)v(3) [524)v(13)	(1)v(2)v(3)v(5) (13)v(25)	[21356) [26)v(15)v(3)	[24)v(1356) [154)v[26)v(3)	[236)v(157) [16)v[27)v(35)
[5312)v(64) [62)v[514)v(3)	[65312) [62)v(15)v(3)	(1)v(3)v(5)v(6) (16)v(35)	[14)v(3)v(5)v(6) [164)v(35)	[13567) [17)v(36)v(5)
[531)v[642) [51)v[62)v(3)v(4)	[6531)v[42) [451)v[62)v(3)	[41)v(3)v(5)v(6) [461)v(35)	(3)v(4)v(5)v(6) (35)v(46)	[43567) [47)v(36)v(5)
[51)v[632)v[74) [61)v[72)v[534)	[751)v[632) [61)v[72)v(35)	[76531) [71)v(36)v(5)	[76534) [74)v(36)v(5)	(3)v(5)v(6)v(7) (37)v(56)

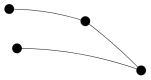


Figure A.42: $\mathcal{D}$-class of induced subgraph with 4 vertices and 3 edges (subgroup C2).

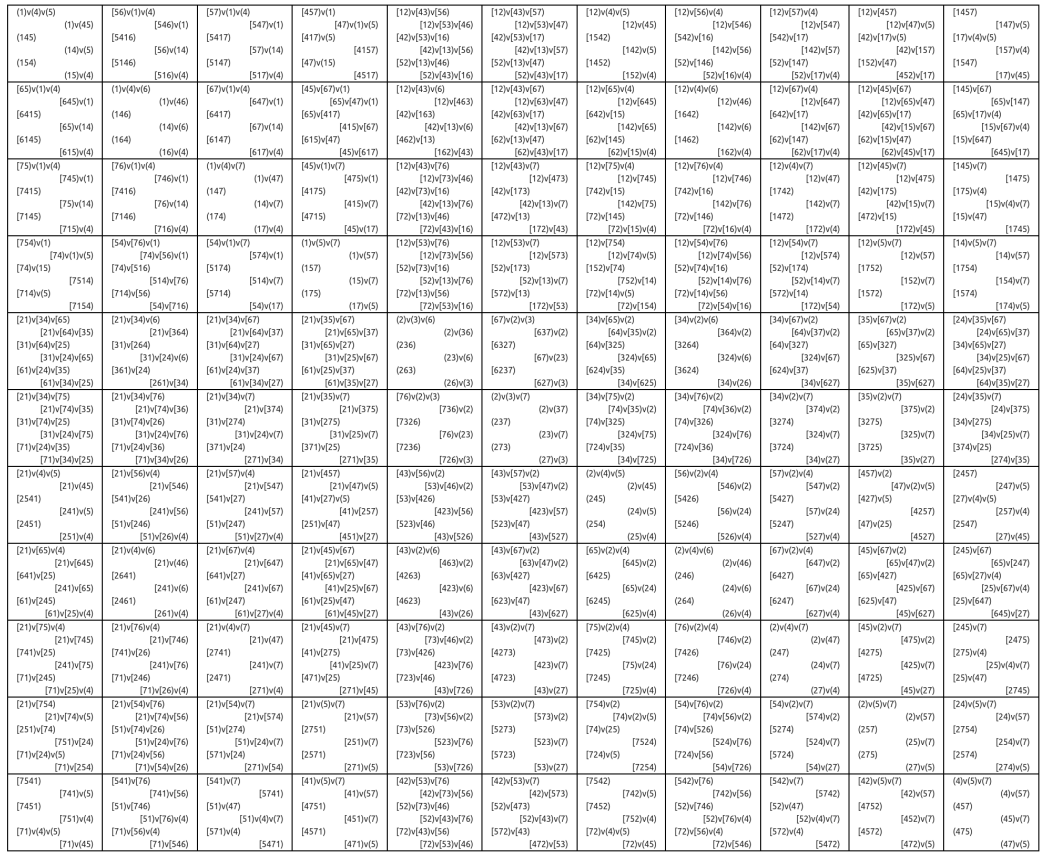


Figure A.43: $\mathcal{D}$-class of induced subgraph with 3 vertices and 0 edges (subgroup S3).

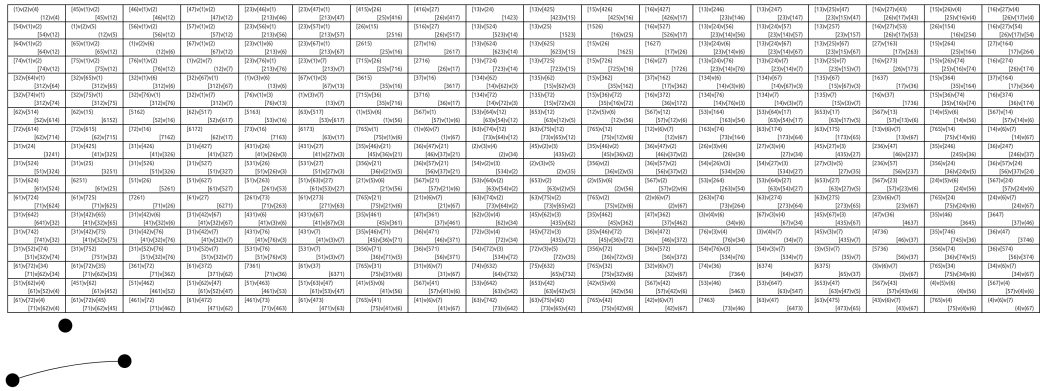


Figure A.44: $\mathcal{D}$-class of induced subgraph with 3 vertices and 1 edge (subgroup C2).

(1)v(2)v(3) (1)v(23)	[2134] [24]v(13)	[2135] [25]v(13)	[24]v[135] [134]v[25]	[15]v[236] [15]v[26]v(3)	[16]v[25]v[37] [16]v[35]v[27]
[4312] [42]v(13)	(1)v(3)v(4) (14)v(3)	[45]v(1)v(3) [415]v(3)	[145]v(3) [15]v[3]v(4)	[135]v[46] [435]v(16)	[36]v[15]v[47] [36]v[45]v(17)
[5312] [52]v(13)	[54]v(1)v(3) [514]v(3)	(1)v(3)v(5) (15)v(3)	[14]v(3)v(5) [154]v(3)	[1356] [16]v(35)	[36]v[157] [36]v(17)v(5)
[531]v[42] [43]v[52]	[541]v(3) [51]v(3)v(4)	[41]v(3)v(5) [451]v(3)	(3)v(4)v(5) (3)v(45)	[4356] [46]v(35)	[36]v[457] [36]v[47]v(5)
[51]v[632] [31]v[62]v(3)	[5311]v[64] [61]v[534]	[6531] [61]v[35]	[6534] [64]v[35]	(3)v(5)v(6) (36)v(5)	[3567] [37]v[56]
[61]v[52]v[73] [61]v[72]v[53]	[63]v[51]v[74] [63]v[71]v[54]	[63]v[751] [63]v[71]v(5)	[63]v[754] [63]v[74]v[5]	[7653] [73]v[56]	(5)v(6)v(7) (57)v(6)

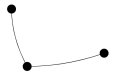


Figure A.45: $\mathcal{D}$-class of induced subgraph with 3 vertices and 2 edges (subgroup C2).

(1)v(4) (14)	[45]v(1) [415]	[46]v(1) [416]	[47]v(1) [417]	[12]v[43] [42]v[13]	[12]v(4) [142]	[12]v[45] [42]v[15]	[12]v[46] [42]v[16]	[12]v[47] [42]v[17]	[13]v[46] [43]v[16]	[13]v[47] [43]v[17]	[145] [15]v(4)	[146] [16]v(4)	[147] [17]v(4)	[15]v[47] [45]v[17]
[54]v(1) [514]	(1)v(5) (15)	[56]v(1) [516]	[57]v(1) [517]	[12]v[53] [52]v[13]	[12]v[54] [52]v[14]	[12]v(5) [152]	[12]v[56] [52]v[16]	[12]v[57] [52]v[17]	[13]v[56] [53]v[16]	[13]v[57] [53]v[17]	[14]v(5) [154]	[14]v[56] [54]v[16]	[14]v[57] [54]v[17]	[157] [17]v(5)
[64]v(1) [614]	[65]v(1) [615]	(1)v(6) (16)	[67]v(1) [617]	[12]v[63] [62]v[13]	[12]v[64] [62]v[14]	[12]v[65] [62]v[15]	[12]v(6) [162]	[12]v[67] [62]v[17]	[13]v(6) [163]	[13]v[67] [63]v[17]	[14]v[65] [64]v[15]	[14]v(6) [164]	[14]v[67] [64]v[17]	[15]v[67] [65]v[17]
[74]v(1) [714]	[75]v(1) [715]	[76]v(1) [716]	(1)v(7) (17)	[12]v[73] [72]v[13]	[12]v[74] [72]v[14]	[12]v[75] [72]v[15]	[12]v[76] [72]v[16]	[12]v(7) [172]	[13]v[76] [73]v[16]	[13]v(7) [173]	[14]v[75] [74]v[15]	[14]v[76] [74]v[16]	[14]v(7) [174]	[15]v(7) [175]
[21]v[34] [31]v[24]	[21]v[35] [31]v[25]	[21]v[36] [31]v[26]	[21]v[37] [31]v[27]	(2)v(3) (23)	[34]v(2) [324]	[35]v(2) [325]	[36]v(2) [326]	[37]v(2) [327]	[236] [26]v(3)	[237] [27]v(3)	[24]v[35] [34]v[25]	[24]v[36] [34]v[26]	[24]v[37] [34]v[27]	[25]v[37] [35]v[27]
[21]v(4) [241]	[21]v[45] [41]v[25]	[21]v[46] [41]v[26]	[21]v[47] [41]v[27]	[43]v(2) [423]	(2)v(4) (24)	[45]v(2) [425]	[46]v(2) [426]	[47]v(2) [427]	[23]v[46] [43]v[26]	[23]v[47] [43]v[27]	[245] [25]v(4)	[246] [26]v(4)	[247] [27]v(4)	[25]v[47] [45]v[27]
[21]v[54] [51]v[24]	[21]v(5) [251]	[21]v[56] [51]v[26]	[21]v[57] [51]v[27]	[53]v(2) [523]	[54]v(2) [524]	(2)v(5) (25)	[56]v(2) [526]	[57]v(2) [527]	[23]v[56] [53]v[26]	[23]v[57] [53]v[27]	[24]v(5) [254]	[24]v[56] [54]v[26]	[24]v[57] [54]v[27]	[257] [27]v(5)
[21]v[64] [61]v[24]	[21]v[65] [61]v[25]	[21]v(6) [261]	[21]v[67] [61]v[27]	[63]v(2) [623]	[64]v(2) [624]	[65]v(2) [625]	(2)v(6) (26)	[67]v(2) [627]	[23]v(6) [263]	[23]v[67] [63]v[27]	[24]v[65] [64]v[25]	[24]v(6) [264]	[24]v[67] [64]v[27]	[25]v[67] [65]v[27]
[21]v[74] [71]v[24]	[21]v[75] [71]v[25]	[21]v[76] [71]v[26]	[21]v(7) [271]	[73]v(2) [723]	[74]v(2) [724]	[75]v(2) [725]	[76]v(2) [726]	(2)v(7) (27)	[23]v[76] [73]v[26]	[23]v(7) [273]	[24]v[75] [74]v[25]	[24]v[76] [74]v[26]	[24]v(7) [274]	[25]v(7) [275]
[31]v[64] [61]v[34]	[31]v[65] [61]v[35]	[31]v(6) [361]	[31]v[67] [61]v[37]	[632] [62]v(3)	[32]v[64] [62]v[34]	[32]v[65] [62]v[35]	[32]v(6) [362]	[32]v[67] [62]v[37]	(3)v(6) (36)	[67]v(3) [637]	[34]v[65] [64]v[35]	[34]v(6) [364]	[34]v[67] [64]v[37]	[35]v[67] [65]v[37]
[31]v[74] [71]v[34]	[31]v[75] [71]v[35]	[31]v[76] [71]v[36]	[31]v(7) [371]	[732] [72]v(3)	[32]v[74] [72]v[34]	[32]v[75] [72]v[35]	[32]v[76] [72]v[36]	[32]v(7) [372]	[76]v(3) [736]	(3)v(7) (37)	[34]v[75] [74]v[35]	[34]v[76] [74]v[36]	[34]v(7) [374]	[35]v(7) [375]
[541] [51]v(4)	[41]v(5) [451]	[41]v[56] [51]v[46]	[41]v[57] [51]v[47]	[42]v[53] [52]v[43]	[542] [52]v(4)	[42]v(5) [452]	[42]v[56] [52]v[46]	[42]v[57] [52]v[47]	[43]v[56] [53]v[46]	[43]v[57] [53]v[47]	(4)v(5) (45)	[56]v(4) [546]	[57]v(4) [547]	[457] [47]v(5)
[641] [61]v(4)	[41]v[65] [61]v[45]	[41]v(6) [461]	[41]v[67] [61]v[47]	[42]v[63] [62]v[43]	[642] [62]v(4)	[42]v[65] [62]v[45]	[42]v(6) [462]	[42]v[67] [62]v[47]	[43]v(6) [463]	[43]v[67] [63]v[47]	[65]v(4) [645]	(4)v(6) (46)	[67]v(4) [647]	[45]v[67] [65]v[47]
[741] [71]v(4)	[41]v[75] [71]v[45]	[41]v[76] [71]v[46]	[41]v(7) [471]	[42]v[73] [72]v[43]	[742] [72]v(4)	[42]v[75] [72]v[45]	[42]v[76] [72]v[46]	[42]v(7) [472]	[43]v[76] [73]v[46]	[43]v(7) [473]	[75]v(4) [745]	[76]v(4) [746]	(4)v(7) (47)	[45]v(7) [475]
[51]v[74] [71]v[54]	[751] [71]v(5)	[51]v[76] [71]v[56]	[51]v(7) [571]	[52]v[73] [72]v[53]	[52]v[74] [72]v[54]	[752] [72]v(5)	[52]v[76] [72]v[56]	[52]v(7) [572]	[53]v[76] [73]v[56]	[53]v(7) [573]	[754] [74]v(5)	[54]v[76] [74]v[56]	[54]v(7) [574]	(5)v(7) (57)

Figure A.46: $\mathcal{D}$-class of induced subgraph with 2 vertices and 0 edges (subgroup C2).

(1)v(2) (12)	[23]v(1) [213]	[13]v[24] [23]v[14]	[13]v[25] [23]v[15]	[15]v[26] [25]v[16]	[16]v[27] [26]v[17]
[32]v(1) [312]	(1)v(3) (13)	[134] [14]v(3)	[135] [15]v(3)	[15]v[36] [35]v[16]	[16]v[37] [36]v[17]
[31]v[42] [41]v[32]	[431] [41]v(3)	(3)v(4) (34)	[45]v(3) [435]	[35]v[46] [45]v[36]	[36]v[47] [46]v[37]
[31]v[52] [51]v[32]	[531] [51]v(3)	[54]v(3) [534]	(3)v(5) (35)	[356] [36]v(5)	[36]v[57] [56]v[37]
[51]v[62] [61]v[52]	[51]v[63] [61]v[53]	[53]v[64] [63]v[54]	[653] [63]v(5)	(5)v(6) (56)	[567] [57]v(6)
[61]v[72] [71]v[62]	[61]v[73] [71]v[63]	[63]v[74] [73]v[64]	[63]v[75] [73]v[65]	[765] [75]v(6)	(6)v(7) (67)

Figure A.47: $\mathcal{D}$-class of induced subgraph with 2 vertices and 1 edge (subgroup C2).

(1)	[12]	[13]	[14]	[15]	[16]	[17]
[21]	(2)	[23]	[24]	[25]	[26]	[27]
[31]	[32]	(3)	[34]	[35]	[36]	[37]
[41]	[42]	[43]	(4)	[45]	[46]	[47]
[51]	[52]	[53]	[54]	(5)	[56]	[57]
[61]	[62]	[63]	[64]	[65]	(6)	[67]
[71]	[72]	[73]	[74]	[75]	[76]	(7)

Figure A.48: $\mathcal{D}$-class of induced subgraph with 1 vertex and 0 edges.