Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Asymmetric graphs

Bachelor Thesis

2022
Simona Dubeková

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Asymmetric graphs

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	doc. RNDr. Tatiana Jajcayová, PhD.
Consultant:	Mgr. Dominika Mihálová

Bratislava, 2022
Simona Dubeková

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname: Simona Dubeková
Study programme: Applied Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study: Computer Science
Type of Thesis: Bachelor´s thesis
Language of Thesis: English
Secondary language: Slovak

Title: Asymmetric graphs

Annotation: We study non-oriented simple graphs. We call a graph symmetric, if there exists a non-identical permutation of its vertices, which leaves the graph invariant, i.e. a graph is called symmetric if the group of its automorphisms is not trivial. A graph which is not symmetric will be called asymmetric. The degree of symmetry of a symmetric graph is measured by the size of its group of automorphisms. We will measure the degree of asymmetry of an asymmetric graph by the number of vertices which we have to delete to obtain a symmetric graph.

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

Assigned: 01.10.2021

Approved: 06.10.2021 doc. RNDr. Damas Gruska, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Simona Dubeková
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Asymmetric graphs
Asymetrické grafy

Anotácia:

Vedúci: doc. RNDr. Tatiana Jajcayová, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 01.10.2021

Dátum schválenia: 06.10.2021 doc. RNDr. Damas Gruska, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgements: I would like to thank my supervisor doc. RNDr. Tatiana Jajcayová, PhD. for her great patience and support throughout the creation of the bachelor's thesis. I would also like to thank my consultant Mgr. Dominika Mihálová for her detailed explanation of system for computational discrete algebra.

Abstrakt

V tejto bakalárskej práci skúmame neorientované jednoduché grafy. Graf nazývame symetrický, ak existuje neidentická permutácia jeho vrcholov, ktorá ponechá graf invariantný, t.j. graf sa nazýva symetrický, ak grupa jeho automorfizmov nie je triviálna. Graf, ktorý nie je symetrický, nazývame asymetrický. Stupeň asymetrie asymetrického grafu meriame počtom vrcholov, ktoré musíme odstrániť, aby sme získali graf symetrický. Neorientované jednoduché asymetrické grafy vytvárame pomocou programovacieho jazyka Kotlin a pre jednoduchšiu predstavu ich vykresľujeme pomocou systému vytvoreného pre výpočtovú diskrétnu algebru - GAP. Z týchto asymetrických grafov postupne, po jednom, odstraňujeme všetky vrcholy a všetky hrany. Pri tomto postupnom odstraňovaní vrcholov a hrán z týchto asymetrických grafov skúmame, ako sa mení symetria týchto grafov oproti grafom pôvodným. Stupeň asymetrie asymetrického grafu potom môžeme vypočítať pomocou počtu vrcholov, ktoré musíme odstrániť, aby sme získali graf symetrický.

Kľúčové slová: graf, vrchol, hrana, asymetrický graf, stupeň asymetrie

Abstract

In this bachelor thesis we study non-oriented simple graphs. A graph is called symmetric if there is a non-identical permutation of its vertices that leaves the graph invariant, i.e. a graph is called symmetric if its group of automorphisms is not trivial. A graph that is not symmetric is called asymmetric. We measure the degree of asymmetry of an asymmetric graph by the number of vertices which we have to delete to obtain a symmetric graph. We create non-oriented simple asymmetric graphs using the Kotlin programming language and we draw them for a better visualization using a system created for computational discrete algebra - GAP. From these asymmetric graphs, we remove all vertices and all edges one by one. In this gradual deletion of vertices and edges from these asymmetric graphs, we study how the symmetry of these graphs changes from the original graphs. The degree of asymmetry of an asymmetric graph can be measured by the number of vertices which we have to delete to obtain a symmetric graph.

Keywords: graph, vertex, edge, asymmetric graph, degree of asymmetry

Contents

List of Figures

List of Tables

Introduction

Graphs are a well-known and widely used structures with many applications in mathematics and computer science. However, this structure is not only usable in the theory of graphs, but also all around us, whether in constellations or only in ordinary relationships between people. The most common example is undoubtedly the road network. In this bachelor thesis we look at graphs from the informatics point of view and apply the theory of mathematics in practice.

In this thesis, we study non-oriented simple graphs. A graph is called symmetric if there exists a non-identical permutation of its vertices, which leaves the graph invariant, i.e. a graph is called symmetric if the group of its automorphisms is not trivial. A graph which is not symmetric is called asymmetric. The degree of symmetry of a symmetric graph is measured by the size of its group of automorphisms. We measure the degree of asymmetry of an asymmetric graph by the number of vertices which we have to delete to obtain a symmetric graph.

In the first chapter of this thesis we deal with basic terminology and concepts related to this topic. It is necessary for our work. In the second chapter, we will take a closer look at how the implementation process was formed, what existing software already exists for examining graphs, and what made it easier for us to examine asymmetric graphs. In the Results chapter, we describe the results we obtained using the software we programmed. We analyse in detail the data we have obtained by commands to our program. After examining the asymmetric graphs, we summarized the results in a catalogue. We will talk about what could be addressed on this topic in the future. What else could be programmed for our existing program.

Finally, we found that even by removing a few vertices we can obtain a relatively high order of the automorphism group. We confirmed that when removing vertices from minimal asymmetric graphs, we get graphs with non-trivial groups of automorphisms. By removing edges from minimal asymmetric graphs, both symmetric and asymmetric graphs can be created. We found, that these asymmetric graphs may not be only minimal.

Chapter 1

Preliminaries

This chapter is devoted to concepts of graph theory, group theory and theory of symmetries which will be used in this thesis. We start with graph theory and continue with groups and its symmetries.

1.1 Graph theory

This section contains basic definitions of graph theory and examples. It is divided into several subsections, where each subsection describes graphs and terms which are connected to our topic. We include pictures and graphs to better illustrate studied concepts.

1.1.1 Examples of graphs

Many people think of graphs just as a series of connected or unconnected dots, but they do not realize all the possibilities which graph theory offers. One of the most common examples of graph in real life is road network. Individual cities represent vertices and the roads represent the edges of the graph. Here is an example of a graph, which shows the tram network in Bratislava in Figure 1.1.

Figure 1.1: Tram Lines

Another example could be constellations, which are not immediately thought of as an easy example of a graph. Constellations consist of stars and their imaginary connections. To make a graph, we can consider stars as vertices and their imaginary connections as edges. Among other uses of graphs belong cardiovascular system, friends on social media or the internet itself. Graphs can be used for quite different things and they are effective tools because they present information quickly and easily.

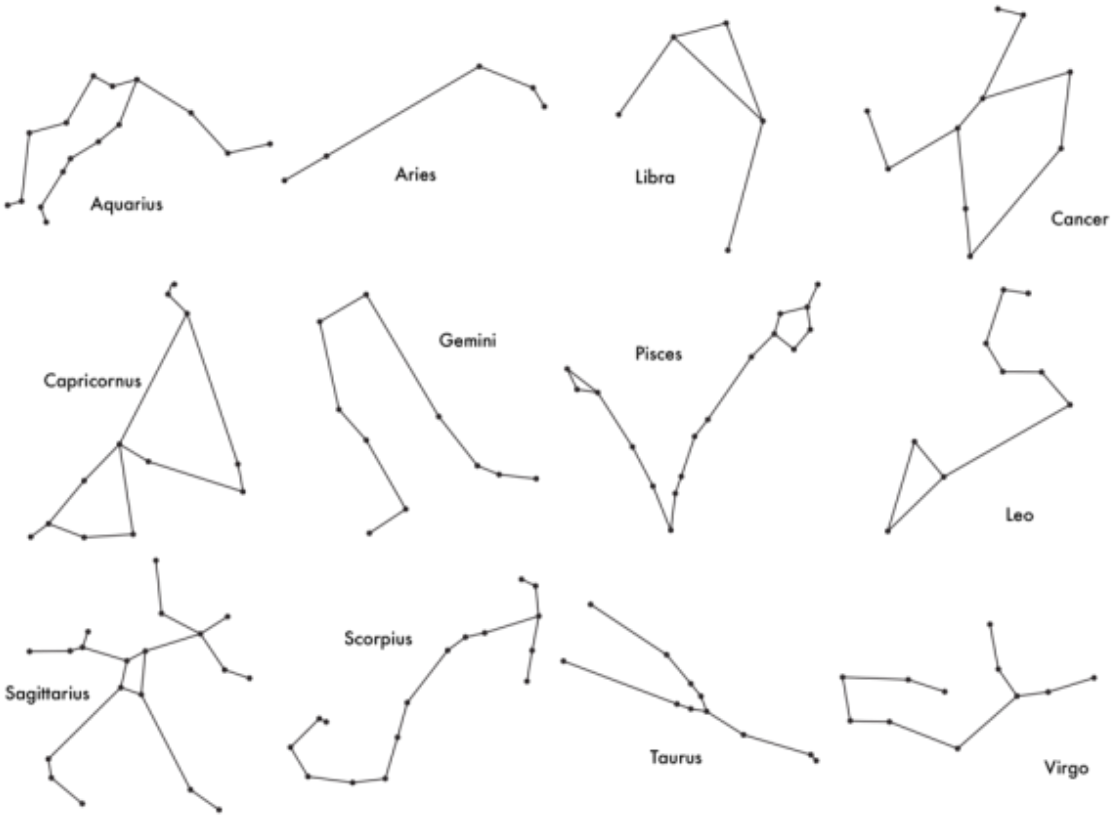

Figure 1.2: Constellations

1.1.2 Graphs basics

Now we can define graphs in mathematics. We found most of the definitions in the book Discrete Structures with Contemporary Applications written by Alexander Stanoyevitch in 2011. [7]

Definition. A *general graph* is a triple $G = (V, E, \phi)$, where V is a non-empty set of vertices (or nodes) of G, and E is set of edges of G, and ϕ, called the edgemap, is a function $\phi : E \to \mathcal{P}(V)$, where $|\phi(e)| = 1$ or 2, for each $e \in E$. The vertices in $\phi(e)$ are called endpoints of the edge e. An edge e having only one endpoint (i.e., $|\phi(e)| = 1$) is called a self-loop. Two edges, e_1, e_2 that have the same endpoints (i.e., $\phi(e_1) = \phi(e_2)$)

are called parallel edges or multiedges. [7]

In this work, we will concentrate on simple graphs where ϕ can be omitted.

Definition. A non-oriented *simple graph* is an ordered pair of sets $G = (V, E)$, where V is a non-empty set of vertices (or nodes) of G, and E, the set of edges of G is a set of two-element pairs (2-combinations) of vertices. Thus, each edge of G can be expressed as $\{u, v\}$, where u an v are distinct vertices, i.e., $u, v \in V$, $u \neq v$. The vertices u and v determining an edge $\{u, v\}$ are called the endpoints of the edge. The edge $\{u, v\}$ is said to join u and v, and the edge is said to be incident to either of its endpoints. Any two vertices in G that are joined by an edge are said to be adjacent and are called neighbours. A vertex with no neighbours is called isolated. [7]

Basically, a graph with no loops and no parallel edges is called a simple graph. The maximum number of edges possible in a single graph with n vertices is $\frac{n(n-1)}{2}$ and the number of simple graphs possible with n vertices is $2^{\frac{n\cdot(n-1)}{2}}$.

For completeness, we will use simple non-oriented graphs, where the edges of the graph G are an unordered pair of vertices (u, v).

1.1.3 Bipartite graphs

An important term in graph theory is a bipartite graph. Bipartite graphs have many applications in informatics. They are often used to represent binary relations between two types of objects.

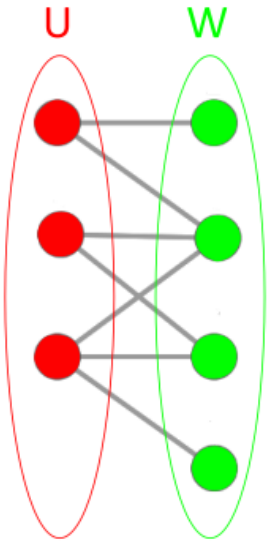

Figure 1.3: Example of bipartite graph

Definition. A graph G is bipartite if the vertex set V can be partitioned into two subsets: $V = U \cup W$, such that each edge of G has one endpoint in U and one endpoint W. The pair U, W is called a (vertex) bipartition of G. [7]

In other words, the bipartite graph or bigraph is a graph whose set of vertices can be divided into two disjoint sets so that no two vertices from the same set are connected by an edge. For better visualization there is an example of bipartite graph in 1.3. There are two sets of vertices, but no edge joins two vertices in the same set. Therefore, this graph is bipartite.

1.1.4 Subgraphs

We start with a definition of a subgraph.

Definition. If $G = (V, E)$ is a graph (directed or undirected), then $G_1 = (V_1, E_1)$ is called a *subgraph* of G if $\emptyset \neq V_1 \subseteq V$ and $E_1 \subseteq E$, where each edge in E_1 is incident with vertices in V_1. [2]

This is the definition of a general subgraph. In our thesis, we will work with more specific types of subgraphs. For example, we will use an *induced subgraph.*

Definition. Let $G = (V, E)$ be a graph (directed or undirected). If $\emptyset \neq U \subseteq V$, the *subgraph* of G *induced* by U is the subgraph whose vertex set is U and which contains all edges (from G) of either the form (a) (x, y), for $x, y \in U$ (when G is directed), or (b) (x, y), for $x, y \in U$ (when G is undirected). We denote this subgraph by $\langle U \rangle$. A subgraph G' of a graph $G = (V, E)$ is called an *induced subgraph* if there exists $\emptyset \neq U \subseteq V$, where $G' = \langle U \rangle$. [2]

To better understand the difference between a subgraph and an induced subgraph, here we have a picture 1.4.

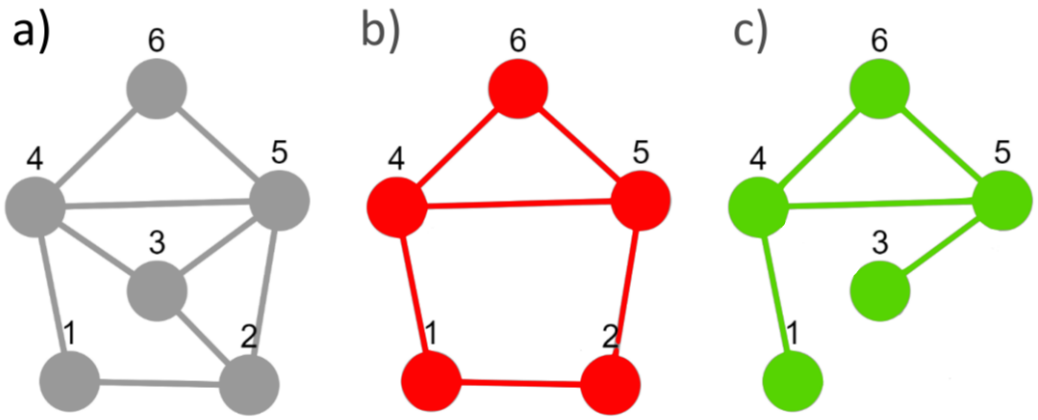

Figure 1.4: a) an example of a graph, b) induced subgraph of the first graph, c) subgraph of the first graph, that is not induced

In the picture 1.4 we can see two subgraphs of graph a). Although the first subgraph is an induced subgraph, the second is not because it lacks the edge $\{3, 4\}$ to become an induced subgraph.

In our work we also work with *spanning subgraph.*

Definition. Given a (directed or undirected) graph $G = (V, E)$, let $G_1 = (V_1, E_1)$ be a a *subgraph* of G. If $V_1 = V$, then G_1 is called a *spanning subgraph* of G. [2]

Another types of subgraphs are $G - v$ *subgraph* and $G - e$ *subgraph.*

Definition. Let v be a vertex in directed or an undirected graph $G = (V, E)$. The subgraph of G denoted by $G - v$ has the vertex set $V_1 = V - \{v\}$ and the edge set $E_1 \subseteq E$ where E_1 contains all the edges in E except for those that are incident with the vertex v. (Hence $G - v$ is the subgraph of G induced by V_1.) In similar way, if e is an edge of a directed or an undirected graph $G = (V, E)$, we obtain the subgraph $G - e = (V_1, E_1)$ of G, where the set of edges $E_1 = E - \{e\}$, and the vertex set is unchanged (that is $V_1 = V$). [2]

1.1.5 Paths, circuits, and reachability in graphs

Definition. Suppose that $G = (V, E)$ is a graph, and $v, w \in V$ are pair of vertices. A *path* in G from v to w is an alternating sequence of vertices and edges:

$$P = \langle v = v_0, e_1, v_1, e_2, v_2, ..., v_{k-1}, e_k, v_k = w \rangle , \tag{1.1}$$

such that the endpoints of edge e_i are vertices $\{v_{i-1}, v_i\}$, for $1 \leq i \leq k, v_0 = v$, and $v_k = w$. We say the path P passes through the vertices $v_0, v_1, v_2, ..., v_{k-1}, v_k$, and traverses the edges $e_1, e_2, ..., e_k$, and that the path has length k, since it traverses k edges. If such a path exists, we say that the vertex w is *reachable* from the vertex v in G. A path having positive length ($k > 0$) from any vertex to itself is called a *circuit.* A path (or circuit) is called simple if it never traverses the same edge twice. A simple circuit that does not pass through the same vertex twice (except for the initial and final vertex) is called a cycle. [7]

Observe that in the case of a simple graph, the above path can be specified by the sequence of vertices $\langle v_0, v_1, v_2, ..., v_{k-1}, v_k \rangle$, since the corresponding edges are uniquely determined. Also note, that any vertex is reachable from itself by the path $\langle v_0 \rangle$ of length zero. [7]

1.1.6 Connectivity of graphs

Connectivity is one of the basic concepts of graph theory used in mathematics and computer science. Thanks to connectivity, it is possible to traverse a graph from one vertex to another vertex.

Definition. Let $G = (V, E)$ be an undirected graph. We call G *connected* if there is a path between any two distinct vertices of G. A graph that is not connected is called *disconnected.* [2]

Definition. For any graph $G = (V, E)$, the number of components of G is denoted by $\kappa(G)$. [2]

Definition. Let $G = (V, E)$ be a connected undirected graph. An *articulation point* of G is a vertex whose removal disconects G. A *bridge* of G is an edge whose removal disconnects G. A *biconnected component* of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. [8]

1.1.7 Trees

Definition. A *forest* is a simple graph that contains no cycles. A *tree* is a connected forest. [7]

If T is a simple graph of n vertices, then the following statements are logically equivalent:

1. T is a tree.
2. T is connected and has $n - 1$ edges.
3. T has no cycles and has $n - 1$ edges.
4. Any two vertices of T are joined by a unique simple path in T.
5. T is connected, and every edge is a bridge (i.e., deleting an edge renders the graph disconnected).
6. T has no cycles and if we add any new edge to T (between two of its vertices) the resulting simple graph will have a unique cycle. [7]

Trees are very important graphs, and they have been used in numerous applications. The most familiar example is a family tree. Almost everyone has drawn a family tree already in playschool. The folder structure in any computer operating system or network has also the structure of a tree. Trees are the basis for some of the most efficient search and sorting algorithms. For these reasons trees are the most important graphs in computer science.

As we can see in figure 1.5 b), a vertex of a tree is called a *leaf* if it has degree 1, otherwise it is called an *internal vertex*. [7]

Definition. A *spanning tree* is a subgraph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges.

In our work we need to define a spanning tree, because it is used by the Prim's and Kruskal's algorithm. We used one of these algorithms in the work, because when we were removing vertices from the graph we needed to find out if the graph is still connected. Prim's and Kruskal's algorithm helped us find out.

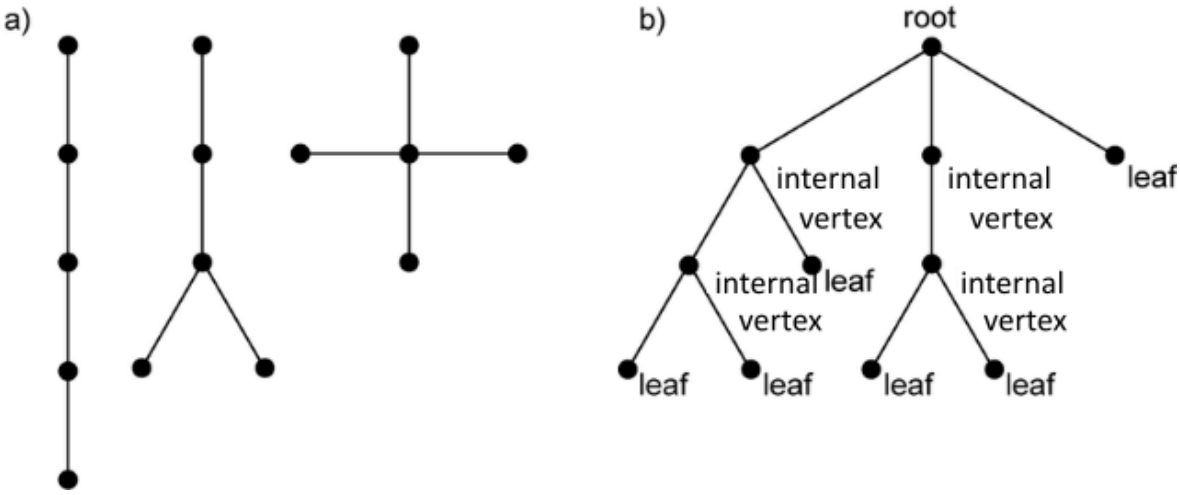

Figure 1.5: Trees: a) examples of non-isomorphic trees of 5 vertices, b) a tree with a root, four inner vertices and six leaves

1.1.8 Algorithms

In computer science, *algorithmic efficiency* is a property of an algorithm which relates to the amount of computational resources used by the algorithm. We know two types of efficiency - space and time efficiency. *Space efficiency* is measured based on the memory required by it for the computation. *Time efficiency* is measured based on the time required by it for the computation. Without going to technical details, big O notation, which will be mentioned later, is a mathematical notation that describes the limiting behaviour of a function when the argument tends towards a particular value or infinity.

This section describes two minimum-spanning-tree algorithms with a detailed elaboration of the general method. The difference between Prim's and Kruskal's algorithm is, that they each use a specific rule to determine a safe edge. As we can see in the picture 1.6, in Kruskal's algorithm, the set A is a forest whose vertices are all those of the given graph. The safe edge added to A is always the least-weight edge in the graph that connects two distinct components. In Prim's algorithm, the set A forms a single tree. The safe edge added to A is always the least-weight edge connecting the tree to a vertex not in the tree. [8]

In Kruskal's algorithm, the set A is a forest whose vertices are all those of the given graph. The safe edge added to A is always a least-weight edge in the graph that connects two distinct components. In Prim's algorithm, the set A forms a single tree. The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree. [8]

The time complexity of Kruskal's algorithm is in the worst case $O(E \log E)$, this is because we need to sort the edges. Prim's algorithm's time complexity in the worst

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

$A = \{(v, v.\pi) : v \in V - \{r\}\}$.

```
MST-PRIM(G, w, r)
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = G.V
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u, v) < v.key
10             v.π = u
11             v.key = w(u, v)
```

Figure 1.6: Kruskal's and Prim's algorithms

case is $O(E \log V)$ with priority queue or even better, $O(E + V \log V)$ with Fibonacci Heap. We should use Kruskal's algorithm when the graph has small number of edges, like $E = O(V)$, when the edges are already sorted or if we can sort them in linear time. On the other hand, we should use Prim's algorithm when the graph's number of edges is high, like $E = O(V^2)$.

1.2 Group theory

This section contains basic definitions of group theory, more precisely definitions of group, group isomorphism and group automorphism.

Definition. If G is non-empty group and $\circ$ is a binary operation on G, then $(G, \circ)$ is called a *group* if the following conditions are satisfied.

1. For all $a, b \in G$, $a \circ b \in G$. (Closure of G under $\circ$)
2. For all $a, b, c \in G$, $a \circ (b \circ c) = (a \circ b) \circ c$. (The Associate Property)
3. There exists $e \in G$ with $a \circ e = e \circ a = a$, for all $a \in G$. (The Existence of an Identity)
4. For each $a \in G$ there is an element $b \in G$ such that $a \circ b = b \circ a = e$. (Existence of Inverses)

Furthermore, if $a \circ b = b \circ a$ for all $a, b \in G$, then G is called a *commutative*, or *abelian* group. [2]

In other words, a group is a pair, where the first item is a finite or infinite set of elements and the second item is a binary operation. This binary operation is a function, which combines two elements (not necessarily distinct) from the set, forming a third one, which also has to be in that specific set. This is a fundamental property of groups

and it is called closure. Groups also have other properties, which distinguish them from other types of structures. The other three properties that a group has to satisfy are associativity, existence of identity and existence of inverse. If a group also satisfies a condition called commutativity, it is referred to as an abelian or commutative group.

Definition. For every group G the number of elements in G is called the *order* of G and this is denoted by $| G |$. If G is group and $a \in G$, the *order of* a, denoted $ord(a)$, is $| \langle a \rangle |$. [2]

1.2.1 Group isomorphism

An *isomorphism* ϕ from a group G to a group $\overline{G}$ is one-to-one mapping (or function) from G onto $\overline{G}$ that preserves the group operation. That is,

$$\phi(ab) = \phi(a)\phi(b) \tag{1.2}$$

for all a, b in G. If there is an isomorphism from G onto $\overline{G}$, we say that G and $\overline{G}$ are *isomorphic* and write $G \approx \overline{G}$.

The visualization of this definition is shown in Figure 1.7. The pairs of dashed arrows represent the group operations.

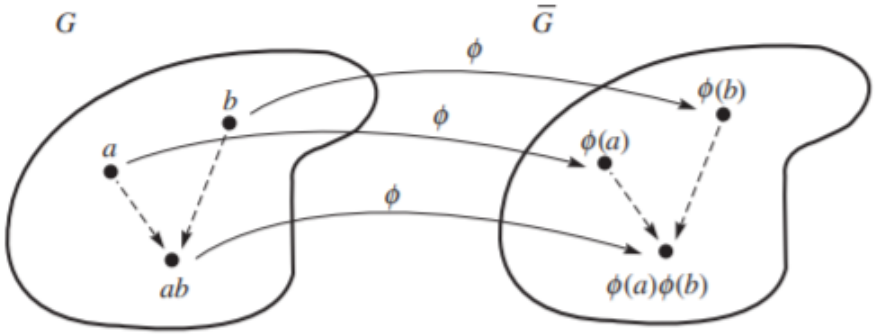

Figure 1.7: Group isomorphism

It is implicit in the definition of isomorphism that isomorphic groups have the same order. It is also implicit in the definition of isomorphism that the operation on the left side of the equal sign is that of G, whereas the operation on the right side is that of $\overline{G}$. The four cases involving $\cdot$ and $+$ are shown in Table 1.1.

[1]

1.2.2 Group automorphism

An isomorphism from a group G onto itself is called an *automorphism* of G. [1]

Table 1.1: Group isomorphism

G Operation	$\overline{G}$ Operation	Operation Preservation
$\cdot$	$\cdot$	$\phi(a \cdot b) = \phi(a) \cdot \phi(b)$
$\cdot$	$+$	$\phi(a \cdot b) = \phi(a) + \phi(b)$
$+$	$\cdot$	$\phi(a + b) = \phi(a) \cdot \phi(b)$
$+$	$+$	$\phi(a + b) = \phi(a) + \phi(b)$

1.3 Theory of symmetries

Most people think of symmetry as beauty of form arising from balanced proportions. In mathematics it means the property of being symmetrical, especially in correspondence in size, shape, and relative position of parts on opposite sides of a dividing line or median plane or about a centre or axis.

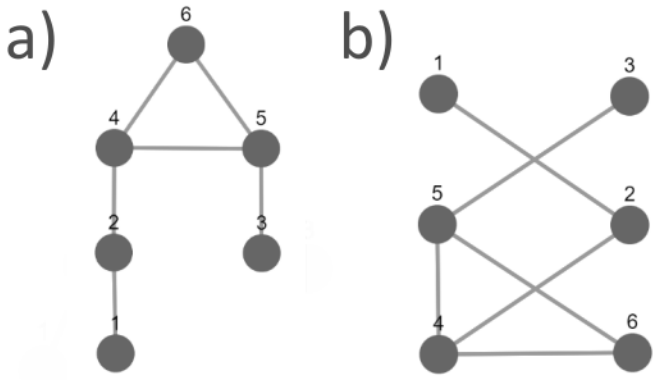

Figure 1.8: Two isomorphic representations of one of the minimal asymmetric graph

1.3.1 Graph isomorphism

Definition. Suppose that $G = (V, E)$ and $G' = \left(V', E'\right)$ are simple graphs, that $f : V \to V'$ is one-to-one (vertex) function, i.e. permutation of vertices.

(a) The function f is said to preserve adjacency if for any pair of vertices $u, v \in V$, we have $\{u, v\} \in E \Rightarrow \{f(u), f(v)\} \in E'$. In other words, if u and v are neighbours in G, then their images $f(u)$ and $f(v)$ must be neighbours in G'.

(b) The function f is said to preserve non-adjacency if for any pair of vertices $u, v \in V$, we have $\{u, v\} \notin E \Rightarrow \{f(u), f(v)\} \notin E'$. In other words, if u and v are not neighbours in G, then their images $f(u)$ and $f(v)$ must not be neighbours in G'.

(c) The function f is said to *be a graph isomorphism* from G to G' if it is bijective, and preserves both adjacency and non-adjacency. In this case we say, that the graphs G to G' are isomorphic, and write this as $G \cong G'$. If no such isomorphism exists, we say that G and G' are not isomorphic and write $G \ncong G'$. [7]

1.3.2 Graph automorphism

An isomorphism from a graph G onto itself is called an *automorphism* of G.

1.3.3 Automorphism group of graphs

All automorphisms of a graph G together on an operation of composition of functions form a group. This group is called an *automorphism group* of graph G and notation is $Aut\,(G)$. In graph theory automorphism groups are classical tools to study structures and symmetries of graphs. In particular, asymmetric graphs are exactly graphs with trivial automorphism groups.

1.3.4 Symmetric graphs

The concept of symmetry can be defined as a transformation of a mathematical structure of a specified kind, that leaves specified properties of the structure unchanged.

There are different types of symmetries, such as rotation and reflection. Graph symmetries are directly related to graph automorphisms because the structure of a graph remains the same. Identity, as it is a trivial automorphism, is also a form of a symmetry. Identity can be referred to as a 0° rotation or a 360° rotation.

The automorphism groups of a graph characterize its symmetries.

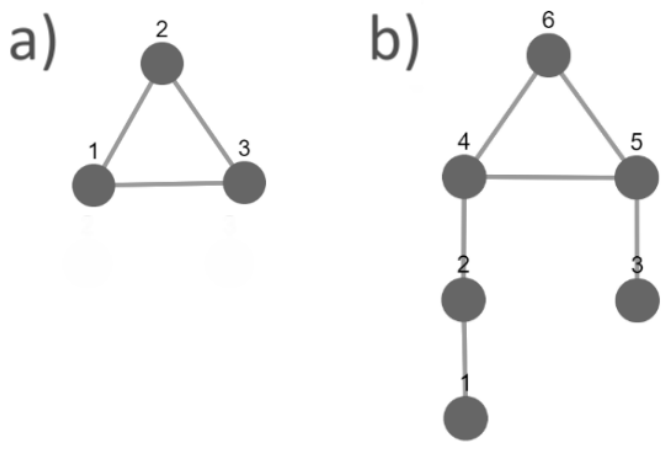

Figure 1.9: Graphs: a) example of symmetric graph, b) example of asymmetric graph

1.3.5 Partial symmetries of graphs

The large number of graphs, whether symmetric or asymmetric, contain subgraphs that have symmetries. Such graphs are known as partially symmetric.

Definition. 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 unary operation of taking inverse has the properties $(s^{-1})^{-1} = s$ and $(st)^{-1} = t^{-1}s^{-1}$ for any $s, t \in S$. [5]

Definition. Given two maps $\varphi_1 : Y_1 \to Z_1$ and $\varphi_2 : Y_2 \to Z_2$, one obtains their composition $\varphi_2\varphi_1$ by composing them on the largest subset of X where it 'makes sense' to do so, that is, on $\mathrm{dom}\ \varphi_2\varphi_1 = \varphi_{-1}(Z_1 \cap Y_2)$, where by definition $(\varphi_2\varphi_1)(x) = \varphi_2(\varphi_1(x))$ for any x. The range of $\varphi_2\varphi_1$ is $ran\varphi_2\varphi_1 = \varphi_2(Z_1 \cap Y_2)$. It may happen that $Z_1 \cap Y_2 = \emptyset$, in which case $\varphi_2\varphi_1$ is an empty map. [5]

Partial automorphism is very important for us, because when removing a vertex or an edge we get its subgraphs from the graph and we can study this partial automorphism on them.

Definition. A *partial automorphism* is an isomorphism between two vertex-induced subgraphs of Γ, that is, a bijection $\varphi : V_1 \to V_2$ between two sets of vertices $V_1, V_2 \subseteq V(\Gamma)$ such that any pair of vertices $u, v \in V_1$ satisfies the condition $(u, v) \in E$ if and only if $(\varphi(u), \varphi(v)) \in E$. The set of all partial automorphisms of Γ together with the operation of the usual composition of partial maps form an inverse monoid, which we denote by $PAut(\Gamma)$. [5]

1.3.6 Asymmetric graphs

A graph is *asymmetric* if it has no non-trivial automorphism. In this thesis we are interested primarly in asymmetric graphs. Specifically, we are interested in such asymmetric graphs, that are as small as possible - minimal. An undirected graph G on at least two vertices is *minimal asymmetric* if G is asymmetric and no proper induced subgraph G on at least two vertices is asymmetric. [6]

There are exactly 18 finite minimal asymmetric undirected graphs up to isomorphism. These 18 graphs are depicted in 1.10. [6]

In 1988, Nešetřil conjectured at an Oberwolfach Seminar that there exists only a finite number of finite minimal asymmetric graphs. Since then, Nešetřil and Sabidussi identified 18 minimal asymmetric graphs in total. [4] This conjecture was not confirmed until 2016 by Pascal Schweitzer and Patrick Schweitzer in an article Minimal Asymmetric Graphs [6]. This article is very important to us, because in this work we use these graphs and investigate them further.

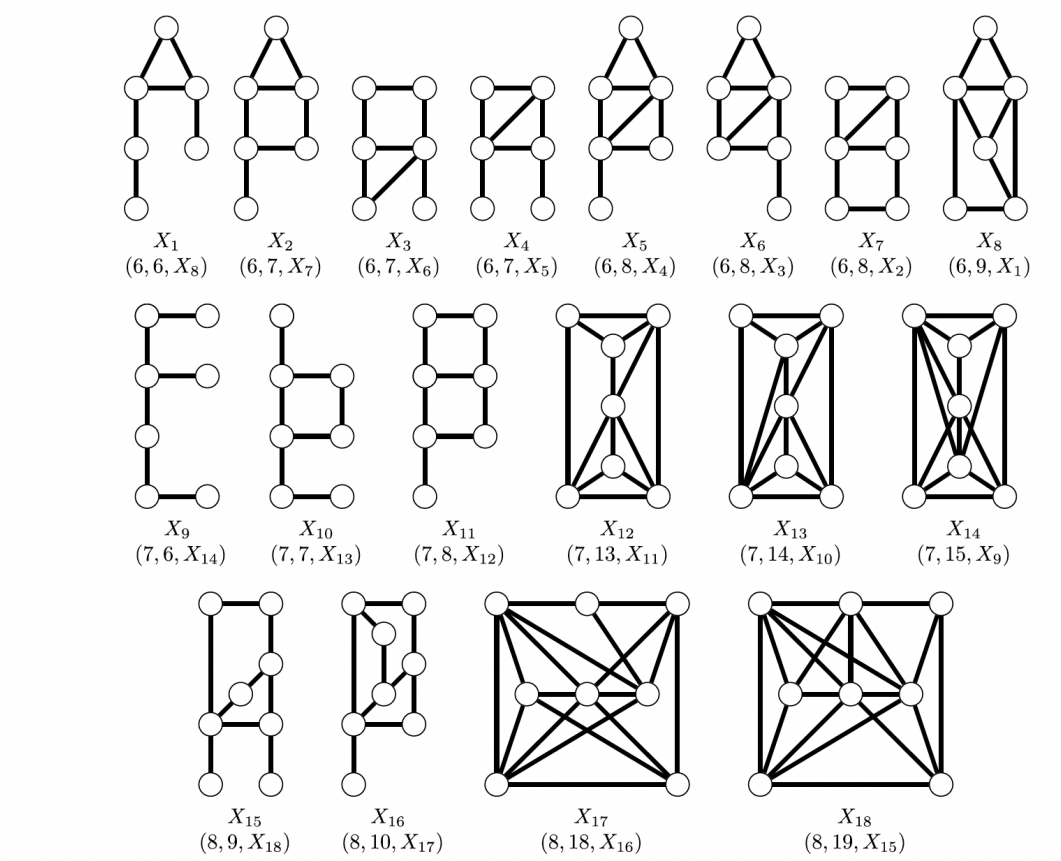

Figure 1.10: These are also the minimal involution-free graphs. For each graph the triple (n, m, co-G), describes the number of vertices, edges and the name of the complement graph, respectively. The graphs are ordered first by number of vertices and second by number of edges [6]

1.4 Motivation to study assymetric graphs

In this section we give the reason why it is interesting to abbreviate the topic of the asymmetric graphs that interest us.

Graphs are generally a very needed structure. As we mentioned above in the text, graphs occur all around us and therefore it is good to know as much as possible about them. In addition to the need for graphs, they are also very nice. They also have a visual appearance, so it is easier for them to understand. Graphs are relatively difficult to explore and difficult to navigate. Nevertheless, we tried to simplify and examine them as much as possible. Specifically, we examined asymmetric graphs. Asymmetric graphs are difficult to explore because they are not symmetric.

In this bachelor thesis, we decided to iteratively remove vertices from asymmetric graphs. By removing the vertices from the graphs, we obtain their induced subgraphs.

The article Inverse monoids of partial automorphisms of graphs [5] is also interested in partially symmetric induced subgraphs. Partial automorphism is very important for

us, because when removing a vertex or an edge we get its subgraphs from the graph and we can study this partial automorphism on them. Among other things, this article solves the problem of determining a group of automorphisms for a specific combinatorial structure, which is also of interest to us.

We examine the graphs using similar algorithms as Tatiana B. Jajcayová and Martin Masár before attending the scientific conference [3].

Chapter 2

Implementation

In this chapter, we describe the process of creating a program that examines asymmetric graphs. The program also examines the order of an automorphism group of each of the graphs that we obtain from minimal asymmetric graphs when iteratively removing vertices or edges. We take a closer look at the GAP program and its Grape and JupyterViz packages. GAP is a system for computational discrete algebra. GAP provides a library of thousands of functions implementing algebraic algorithms written in the GAP language. GAP includes Grape and JupyterViz packages. We need the Grape package in our work for creating graphs and JupyterViz for their subsequent drawing. More about GAP is written later in the work.

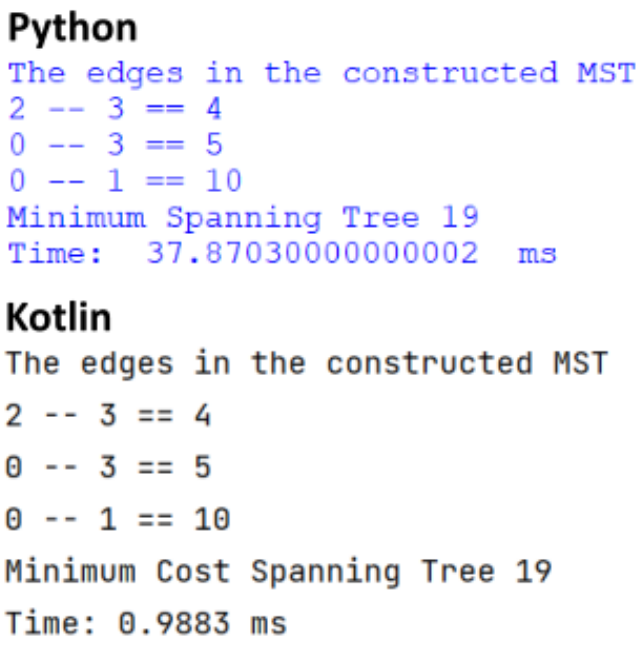

Figure 2.1: Time comparison between Python and Kotlin programming language computing Kruskal's algorithm measured in milliseconds

At the beginning we programmed in Python programming language. It was a clear first choice, as it was the first language we learned at school. However, we later found that it is too slow for so much data that we needed to go through. We can see the speed comparison of Python and Kotlin programming language in the picture 2.1. This is the reason why we started programming in Kotlin programming language.

2.1 Kotlin

Kotlin is a modern programming language that is designed to work fully with Java, but its type of inference allows its syntax to be more concise and simpler.

Kotlin is a very convenient choice even if the application could be extended in the future, because since the release of Android Studio 3.0 in October 2017, Kotlin has been included as an alternative to the standard Java compiler. In May 2019, Google announced that the Kotlin programming language is now its preferred language for Android application developers.

The part of the software program for this bachelor thesis, which was programmed in the Kotlin programming language, was coded in JetBrains IntelliJ Idea framework. We use the Kotlin plugin to provide language support in IntelliJ IDEA.

2.2 GAP

Some functions, such as finding automorphism groups, are quite lengthy and difficult to compile. Therefore, we use the GAP system to determine whether the graph is isomorphic to other graphs created by removing the same number of vertices or edges. We also use it to list the symmetry group of the graphs. GAP is a system for computational discrete algebra that has these functions easily available in its GRAPE package, which is designed to work with graphs. We can therefore use the results that the GAP system provided in our work.

In general, it is not easy to draw graphs. However, it is easier with the JupyterViz package, which is also available in the GAP system. Unfortunately, GAP is a console application and it is very difficult to make a user interface in it, but it is suitable for rendering and finding symmetry groups. Therefore, we decided to combine the program in the programming language Kotlin and its cooperation with the GAP system.

2.3 Algorithms

We used several algorithms in this thesis. Firstly, we used an algorithm to find the power set of graphs of given sizes. This algorithm has time complexity $O(n2^n)$ and

space complexity $O(1)$. This is used in our work to help remove vertices from the graph. Its based on the fact that the power set of vertices contains all the combinations and all possible numbers of vertices that the graph can contain after removing the vertices.

Then we use Prim's or Kruskal's algorithm. These algorithms find a spanning tree for a weighted undirected graph. We use these algorithms to determine if the graph is connected or disconnected.

2.3.1 Prim's Algorithm

In our work, we used Prim's algorithm, specifically in this form.

Algorithm 2.1: Prim's algorithm

```
for (count in 0 until adjVertices.size - 1) {
    val u = minKey(key, mstSet)
    if (u == -1)
        return IntArray(0)
    mstSet[u] = true
    for (v in 0 until adjVertices.size)
        if (adjVertices[Vertex(u.toString())]?.contains(Vertex(
                v.toString())) == true && mstSet[v] == false) {
            parent[v] = u
            key[v] = 1
        }
}
```

Our implementation of Prim's algorithm looks for whether the graph is connected and if it finds that the graph is not connected, it returns an array of size zero. The time complexity of the Prim's algorithm is $O(ElogV)$.

2.4 Classes and Functions

In this section we will describe the classes and functions of the program that we have programmed. The program is mostly written in Kotlin programming language and therefore all these classes and functions are written in Kotlin programming language. The relationships between them can be well seen in the class diagram 2.2. The smallest classes, but the most important for the definition of the graph, are the *Vertex* class and the *Edge* class.

In addition to the classes defined in Kotlin, we also use GAP functions. The functions we use are written in the path.txt file for better manipulation. This file and functions are very important and are mentioned at the end of this chapter.

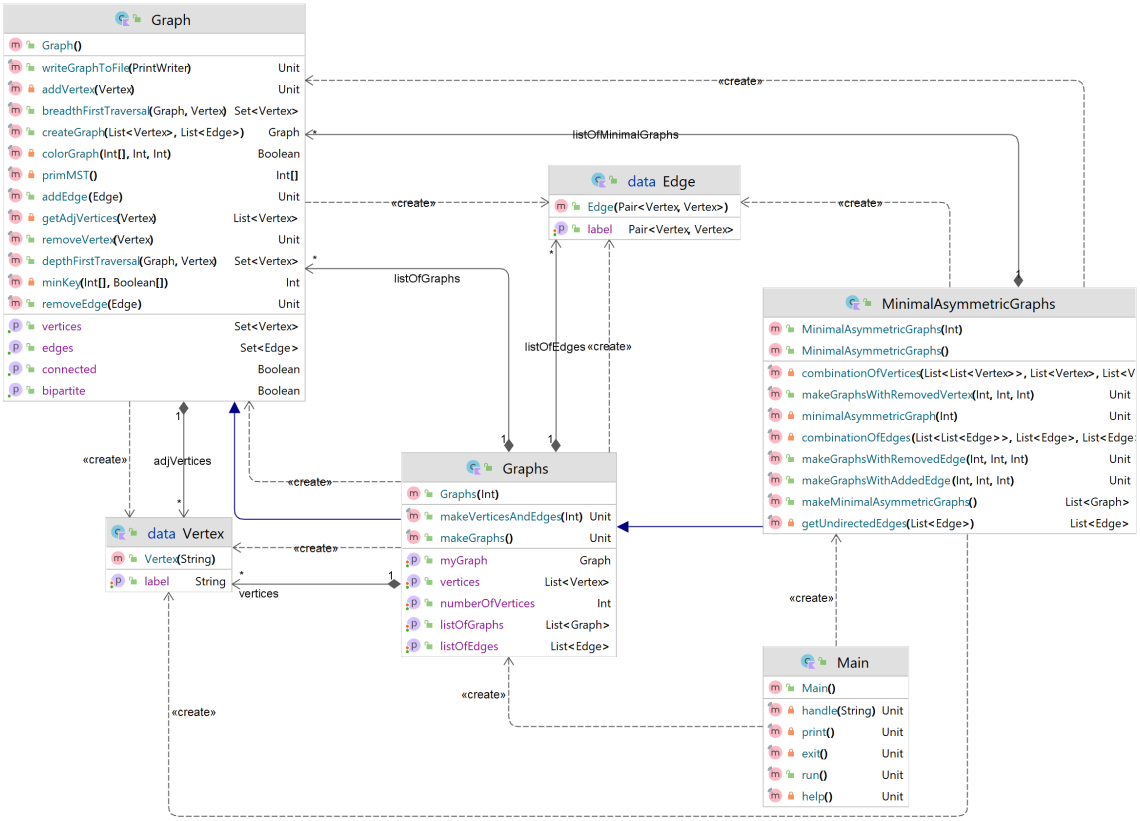

Figure 2.2: Class diagram

2.4.1 Class Vertex

The *Vertex* data class is used to define the vertex. The vertex is defined by its unique name.

```
data class Vertex(var label: String)
```

Figure 2.3: Definition of the class Vertex

2.4.2 Class Edge

The *Edge* data class to define the edge. We have defined the edge as a pair of two vertices.

```
data class Edge(var label: Pair<Vertex, Vertex>)
```

Figure 2.4: Definition of the class Edge

2.4.3 Class Graph

In our program, a graph is defined as a mutable map of vertices and their mutable list of vertices. In other words, we can assign each vertex in the graph a list of vertices to which it is connected. This way we can easily find out the edges of the graph.

In the class *Graph*, we can use the function *getVertices* to find the vertices and the function *getEdges* to find the edges of a given graph. After entering a vertex, the function *getAdjVertices* can find out which vertices the given vertex is connected to. Using the *addVertex* function, we can add vertices and by the *addEdge* function, we can add edges to a specific graph. On the other hand, using the *removeVertex* and *removeEdge* functions we can remove the vertices and edges of the specific graph.

The *writeGraphToFile* function can write a graph in a format that the GAP system can easily read and, if necessary, create a graph from it. In this format, the graph is written using edges separated by two spaces. The edge is written as two vertices separated by one space. This format can be seen in 2.5. This feature is very important for future work with the GAP system.

```
01 10 13 24 31 34 35 42 43 45 53 54
01 10 12 13 21 24 31 34 35 42 43 45 53 54
02 03 13 20 23 24 30 31 32 35 42 45 53 54
02 13 20 23 24 25 31 32 35 42 45 52 53 54
01 10 12 13 14 21 24 31 34 35 41 42 43 45 53 54
02 12 13 14 20 21 24 31 34 35 41 42 43 45 53 54
01 02 10 13 20 23 24 25 31 32 35 42 45 52 53 54
01 03 10 12 14 21 23 24 30 32 34 35 41 42 43 45 53 54
01 02 10 20 23 32 34 35 43 53 56 65
01 02 10 20 23 24 32 35 42 45 46 53 54 64
01 10 12 13 21 24 31 34 35 42 43 46 53 56 64 65
01 02 03 05 10 12 13 16 20 21 23 30 31 32 34 36 43 45 46 50 54 56 61 63 64 65
01 02 03 04 05 10 12 13 16 20 21 23 30 31 32 34 36 40 43 45 46 50 54 56 61 63 64 65
01 02 03 05 10 12 13 16 20 21 23 25 26 30 31 32 34 35 43 45 46 50 52 53 54 56 61 62 64 65
02 13 20 23 24 26 31 32 35 42 45 53 54 57 62 67 75 76
01 10 12 13 16 21 24 31 34 35 42 43 47 53 56 61 65 67 74 76
01 02 03 04 05 10 12 13 17 20 21 23 25 30 31 32 34 35 37 40 43 45 46 47 50 52 53 54 56 64 65 67 71 73 74 76
01 02 03 04 05 10 13 14 17 20 23 25 26 30 31 32 34 35 36 40 41 43 45 46 47 50 52 53 54 56 62 63 64 65 67 71 74 76
```

Figure 2.5: Minimal asymmetric graphs written in GAP format

The *isBipartite* function, as the name implies, can determine whether a graph is bipartite or not by dividing the vertices into two groups.

To determine the connectivity of the graph, the *isConnected* function, we used Prim's algorithm 2.1 to find a minimum spanning tree of the graph. If the algorithm does not find this minimum spanning tree, the graph is not connected.

2.4.4 Class Graphs

After entering the number of vertices, the *Graphs* class forms all graphs that can be created for a given number of vertices.

The *makeVerticesAndEdges* function creates all possible edges in the graph with the specified number of vertices. This function is used by the *makeGraphs* function, which creates all graphs with a given number of vertices.

2.4.5 Class MinimalAsymmetricGraphs

The *MinimalAsymmetricGraphs* class implements the Graphs class. This class uses the *makeMinimalAsymmetricGraphs* function to create minimal asymmetric graphs. If we want to create only one minimal asymmetric graph, we use the private group *minimalAsymmetricGraph*, whose parameter is a number that determines which graph we want to create.

The *makeGraphsWithRemovedEdge* function removes the specified number of edges from the minimal asymmetric graphs. This function cooperates with the *combinationOfEdges* function, which creates all combinations of all edges that can arise from the edges that the graph originally contained. The *makeGraphsWithRemovedEdge* function checks whether the graph is still connected after removing an edge, and whether such a graph has been created before, when removing other edges. It writes these graphs to a file in a format suitable for reading graphs in GAP program. This format is the same as in 2.5. The function prints how many connected graphs have been created after removing the specified number of edges.

The *makeGraphsWithRemovedVertex* function removes the specified number of vertices from the minimal asymmetric graphs. This function cooperates with the *combinationOfVertices* function, which creates all combinations of all vertices that can arise from the vertices that the graph originally contained. The *makeGraphsWithRemovedVertex* function checks whether the graph is still connected after removing a vertex, and whether such a graph has been created before, when removing other vertices. It writes the connected graphs to a file in a format suitable for reading graphs by the GAP program. The function prints how many connected graphs have been created after removing the specified number of vertices.

The *makeGraphsWithAddedEdge* function adds the specified number of edges to the minimal asymmetric graphs. This function, like the *makeGraphsWithRemovedEdge* function, cooperate with the *combinationOfEdges* function. The *makeGraphsWithAddedEdge* function checks, if such a graph has already been created. Graphs that have not yet been created are written to a file in a format suitable for reading graphs by the GAP program. The function prints how many connected graphs have been created after adding the specified number of edges.

2.4.6 Class Main

The *Main* class runs the whole program. When we run the program, main runs the *run* function, which prints the *help* and *print* functions or calls the *exit* function. The *exit* function can be called by user when he wants to quit the program. The program uses the *print* function to list several options 2.6, from which the user chooses which option they want to use, or exit, by pressing the E key, the program. To better understand

how the program works, the *help* function prints what to do when the program starts. The *handle* function is determined by what the program does when the key is pressed.

```
~~            Graphs            ~~

1. Create graph with 1 vertex.
2. Create graphs with 2 vertices.
3. Create graphs with 3 vertices.
4. Create graphs with 4 vertices.
5. Create graphs with 5 vertices.
6. Create graphs with 6 vertices.

~~              Minimal asymmetric graphs              ~~

7.  Create minimal asymmetric graphs.
8.  Print if minimal asymmetric graphs are bipartite.
9.  Create minimal asymmetric graphs without 1 vertex.
10. Create minimal asymmetric graphs without 2 vertices.
11. Create minimal asymmetric graphs without 3 vertices.
12. Create minimal asymmetric graphs without 4 vertices.
13. Create minimal asymmetric graphs without 5 vertices.
14. Create minimal asymmetric graphs without 1 edge.
15. Create minimal asymmetric graphs without 2 edges.
16. Create minimal asymmetric graphs without 3 edges.
17. Create minimal asymmetric graphs without 4 edges.
18. Create minimal asymmetric graphs without 5 edges.
19. Create minimal asymmetric graphs with 1 more edge.
20. Create minimal asymmetric graphs with 2 more edges.
21. Create minimal asymmetric graphs with 3 more edges.
22. Create minimal asymmetric graphs with 4 more edges.
23. Create minimal asymmetric graphs with 5 more edges.

E.  exit
```

Figure 2.6: Main menu

2.4.7 File path.txt

When we run the GAP program, for the simplicity of the program, we load the text file path.txt 2.7, in which we have written all the commands that we would otherwise write to the console. We load the packages we will need when working with graphs and its visualization. From the file we created in Kotlin, we load the graphs we created

after removing the vertices from the minimal asymmetric graphs. We test these graphs to see if they are isomorphic. In general, it is a very difficult (NP complete) problem to decide whether two graphs are isomorphic, or more generally, whether one graph is a subgraph of another graph. [7] This problem is solved by the classical Weisfeiler-Lehman algorithm, a graph-isomorphism test based on colour refinement.

We also determine the group of automorphism and its order by the graph. The problem of determining a group of automorphisms for a particular combinatorial structure or class of combinatorial structures is a notoriously complex computational task, the exact complexity of which is the subject of intensive research efforts. [5] Finally, we draw the resulting graphs.

```
simon@LAPTOP-5NTBCIVG ~
$ cd C:/gap-4.11.1

simon@LAPTOP-5NTBCIVG /cygdrive/c/gap-4.11.1
$ ./gap
  ┌─────┐   GAP 4.11.1 of 2021-03-02
  │ GAP │   https://www.gap-system.org
  └─────┘   Architecture: x86_64-pc-cygwin-default64-kv7
 Configuration:  gmp 6.2.0, GASMAN, readline
 Loading the library and packages ...
 Packages:   AClib 1.3.2, Alnuth 3.1.2, AtlasRep 2.1.0, AutoDoc 2020.08.11,
             AutPGrp 1.10.2, Browse 1.8.11, CaratInterface 2.3.3, CRISP 1.4.5,
             Cryst 4.1.23, CrystCat 1.1.9, CTblLib 1.3.1, FactInt 1.6.3,
             FGA 1.4.0, Forms 1.2.5, GAPDoc 1.6.4, genss 1.6.6, IO 4.7.0,
             IRREDSOL 1.4.1, LAGUNA 3.9.3, orb 4.8.3, Polenta 1.3.9,
             Polycyclic 2.16, PrimGrp 3.4.1, RadiRoot 2.8, recog 1.3.2,
             ResClasses 4.7.2, SmallGrp 1.4.2, Sophus 1.24, SpinSym 1.5.2,
             TomLib 1.2.9, TransGrp 3.0, utils 0.69
 Try '??help' for help. See also '?copyright', '?cite' and '?authors'
gap> Read("path.txt");
```

Figure 2.7: Loading file path.txt in GAP

Chapter 3

Results

In this chapter, we will analyse the results obtained by our program. We will answer a few questions and evaluate in detail what results we have obtained and whether such conclusion was expected or rather surprising.

3.0.1 Minimal asymmetric graphs without one vertex

Firstly, we evaluate the graphs that were created after removing one vertex from the minimal asymmetric graphs. As we mentioned earlier, our program is used to remove and add vertices and edges to minimal asymmetric graphs.

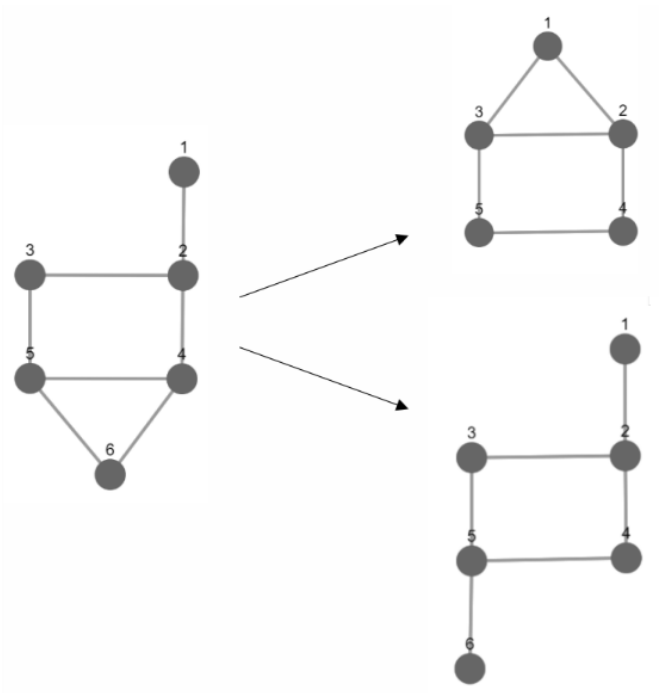

Figure 3.1: One of the minimal asymmetric graph and the examples of the graphs, which were created after removing one vertex from that minimal asymmetric graph

After removing one vertex, we did not just remove the vertex itself. With its

removal, the edges with which it was connected also disappeared. After removing one vertex, a total of forty-four non-isomorphic connected graphs were generated. The resulting graphs are no longer asymmetric. The reason is because vertices have been removed from the minimal asymmetric graphs.

Minimal asymmetric graphs are minimal precisely because when we take even one vertex from them, they will no longer be asymmetric. We confirmed this statement as we really got only graphs with non-trivial symmetries, with groups of automorphism of orders 2 or 4. Most of them were made with group of automorphism of orders 2, which could be assumed. In the picture 3.1 we have an example of one of the minimal asymmetric graphs and the graphs, which were created after removing one vertex from that minimal asymmetric graph. First graph has the group of automorphism of order 2 and the second of order 4.

3.0.2 Minimal asymmetric graphs without iteratively removed vertices

As a next step, we tried to remove two vertices from the minimal asymmetric graphs. After this removal, we obtained thirty-one non-isomorphic graphs. These graphs are not asymmetric, which confirms that the original minimal asymmetric graphs are really minimal. After removing two vertices from the minimal asymmetric graphs, we obtained more different groups of automorphism than by removing only one vertex. These graphs already contain automorphism groups of orders 2, 4, 6, 8, 12, and even 16.

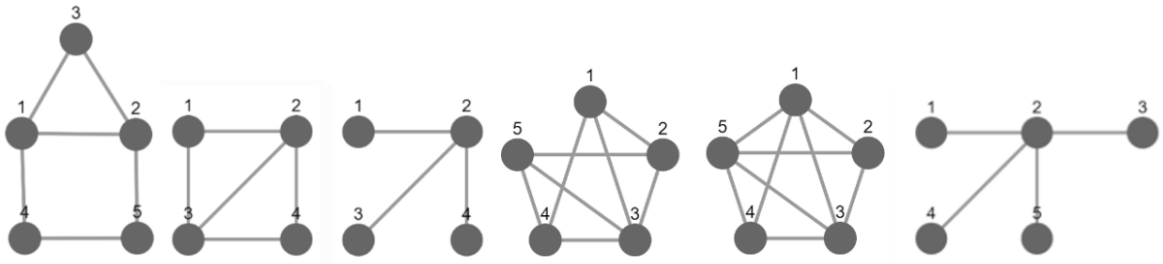

Figure 3.2: Example of the minimal asymmetric graph without three vertices

After removing three vertices from the minimal asymmetric graphs, we obtained nineteen non-isomorphic connected graphs. These graphs are not asymmetric, and have automorphism groups of orders 2, 4, 6, 8, 12, and even 24. By removing more vertices from the minimal asymmetric graphs, we get a higher order of the automorphism group. The figure 3.2 shows six graphs created by removing exactly three vertices from their respective minimal asymmetric graphs. Each of these graphs has a different group of automorphism. The size of the automorphism group increases from left to right,

starting at 2, gradually continuing 4, 6, 8, 12 and ending at 24. As we can see in the figure 3.2, graphs with a higher group of automorphism are more symmetric than graphs with a lower group. It looks like the graphs will have a progressively higher group of automorphism after removing more vertices. It is very interesting that we only need to remove a few vertices from a graph that does not have symmetry, and suddenly the group of automorphism is so large. Therefore, we iteratively removed additional vertices from the minimal asymmetric graphs.

After removing four vertices from the minimal asymmetric graphs, we obtained eight non-isomorphic connected graphs. These graphs have automorphism groups of orders 2, 4, 6, and 24. We have already achieved these sizes of automorphism groups. After removing five vertices from the minimal asymmetric graphs, we obtained three non-isomorphic connected graphs. These graphs have automorphism groups of orders 2 and 6. We have already reached the largest group of automorphism by removing three vertices from the minimal asymmetric graphs.

Figure 3.3: The minimal asymmetric graph without six vertices and with an automorphism group of order 2

After removing six vertices from the minimal asymmetric graphs, we get only one non-isomorphic connected graph 3.3 with an automorphism group of order 2, and removing the other vertices would lead to graph with + 1 vertex.

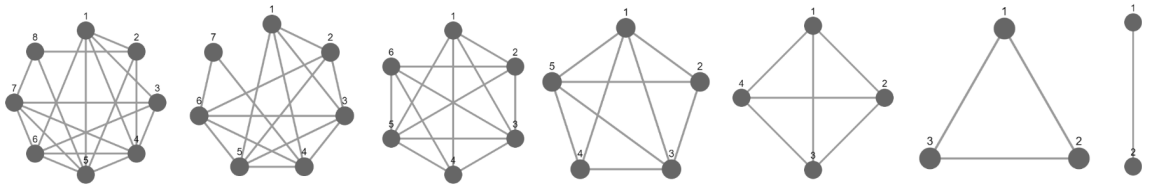

Figure 3.4: One of the minimal asymmetric graphs with an example of iteratively removed vertices

In the image 3.4 we can see one of the minimal asymmetric graphs, from which we iteratively removed the vertices. It is interesting to see how a graph that was initially asymmetric, without symmetry, with only one vertex removed, became symmetric with an automorphism group of order 4 and by removing four vertices the graph became symmetric with an automorphism group of order 24.

By removing two vertices the graph reached an automorphism group of order 16. After removing three vertices, we obtain a graph with a group of automorphism of

order 12. After removing five vertices, a graph with a group of automorphism of order 6 and after removing six vertices, a graph with a group of automorphisms of order 2.

Removing multiple vertices lead to graph with 1 vertex. The complete catalogue with all graphs that were created by iteratively removing vertices from minimal asymmetric graphs can be found in the appendix of this bachelor thesis.

3.0.3 Minimal asymmetric graphs without one edge

Similarly, we tried to remove the edges from the minimal asymmetric graphs. After removing exactly one edge, we get 144 non-isomorphic connected graphs. These graphs have automorphism groups of the order 1, 2, 4, and 6.

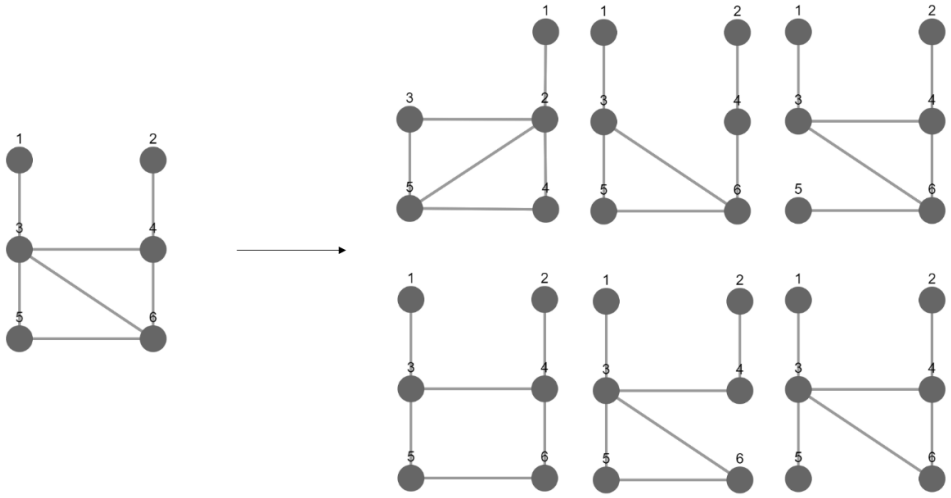

Figure 3.5: One of the minimal asymmetric graphs and all the graphs, that are created by removing one edge from that minimal asymmetric graph

In the figure 3.5 we can see one of the minimal asymmetric graphs and the graphs that were created from it after removing just one edge. As we can see, removing an edge can also remove a vertex if no other edge led from that vertex, only the one we removed from the graph.

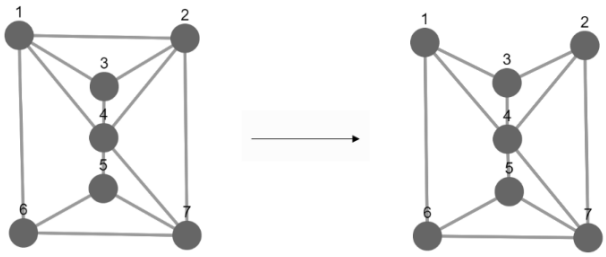

Figure 3.6: One of the minimal asymmetric graphs and one of the graphs, that is created by removing one edge from that minimal asymmetric graph

By removing the edge from the minimal asymmetric graphs several options may occurs. The resulting graph may be: symmetric, asymmetric but not minimal or minimal asymmetric. The figure 3.5 shows one minimal asymmetric graph (second from the left) and five symmetric graphs created by removing one edge from one minimal asymmetric graph (the one on the left side of an arrow). These graphs have groups of automorphism, sequentially from the left, of orders 2, 1, 6, 2, 2, and 2. The figure 3.6 shows one of the graphs created by removing one edge from another one of the minimal asymmetric graphs. This graph is asymmetric, but not minimal with an automorphism group of order 1.

3.0.4 Minimal asymmetric graphs without iteratively removed edges

After removing two edges from the minimal asymmetric graphs, we obtained 507 non-isomorphic connected graphs. These graphs have automorphism groups of orders 1, 2, 4, 6, 8, 10, and 12. After removing three edges from the minimal asymmetric graphs, we obtained 1303 non-isomorphic connected graphs. These graphs have automorphism groups of orders 1, 2, 4, 6, 8, 10, 12 and 16.

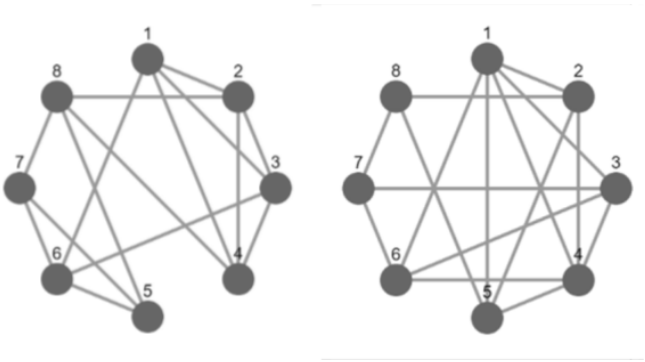

Figure 3.7: The minimal asymmetric graphs without three edges and with an automorphism group of order 16

By removing edges from minimal asymmetric graphs, we have obtained a huge number of graphs, whether symmetric or asymmetric. In the picture 3.7 we can see graphs with the largest automorphism group of order 16.

3.0.5 Minimal asymmetric graphs with one added edge

After adding exactly one edge to the minimal asymmetric graphs, we obtained 159 non-isomorphic graphs. These graphs have automorphism groups of order 1, 2, 4, and 6. The automorphism group of order 1 means that these graphs are asymmetric. Thus,

adding edges to minimal asymmetric graphs can create symmetric, asymmetric but not minimal or minimal asymmetric graphs.

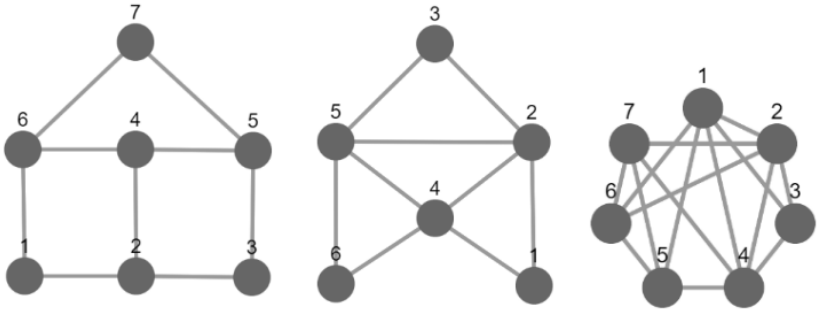

Figure 3.8: List of the minimal asymmetric graphs with one added edge and with an automorphism group of order 6

In the image 3.8 we can see all the graphs with a group of automorphism of order 6, which were created from minimal asymmetric graphs by adding exactly one edge. These graphs are the most interesting for us, because such graphs with an automorphism group of order 6 were created from graphs that had no symmetry.

3.0.6 Minimal asymmetric graphs with more added edges

After adding two edges to the minimal asymmetric graphs, we get 548 non-isomorphic graphs. These graphs have automorphism groups of orders 1, 2, 4, 6, and even 8, 12, or 16.

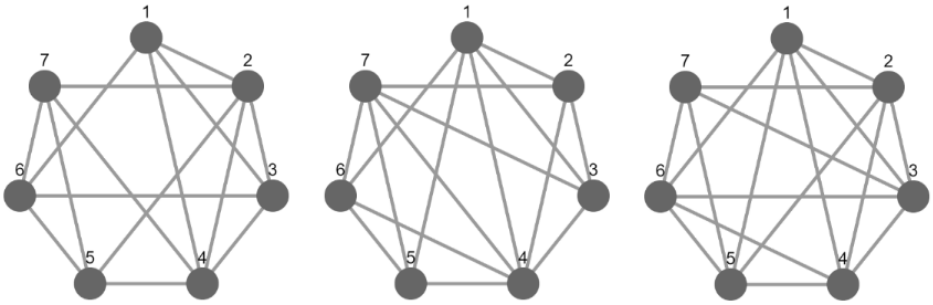

Figure 3.9: List of the minimal asymmetric graphs with two added edges and with an automorphism group of order 16

In the figure 3.9 we can see all graphs with a group of automorphisms of order 16, which were created from minimal asymmetric graphs by adding just two edges.

By adding more edges to the minimal asymmetric graphs, we obtain more and more groups of automorphism until we obtain a complete graph that contains all the edges it can contain.

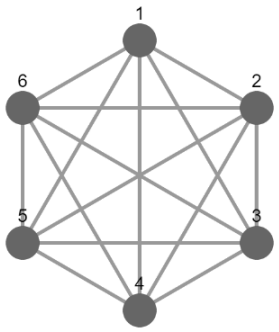

Figure 3.10: Complete graph with 6 vertices

In the picture 3.10 we can see a complete simple non-oriented graph with an automorphism group of order 720. We obtained this graph by adding six edges to one of the minimal asymmetric graph.

Summary

This work was focused on the study of non-oriented simple graphs. We measured the degree of asymmetry of an asymmetric graph by the number of vertices which we have to delete to obtain a symmetric graph.

To examine the minimal asymmetric graphs, we created a console application in the Kotlin programming language. We iteratively removed vertices and later edges from these graphs. Finally, we added edges to see how these minimal asymmetric graphs behave when adding or removing edges and vertices. By adding and removing edges and vertices from the minimal asymmetric graphs, we got connected graphs. The graphs we generated in this application were written to a file in a format that is suitable for creating graphs by the GAP program. We read these graphs in the GAP program and selected only non-isomorphic graphs using the GRAPE package. Finding non-isomorphic graphs is a relatively complex process with great complexity. We used the JupyterViz package to plot these graphs.

By iteratively removing the vertices from the minimal asymmetric graphs, we found that even by removing a few vertices we can obtain a relatively high order of the automorphism group. We confirmed that when removing vertices from minimal asymmetric graphs, we get graphs with non-trivial groups of automorphisms. We described these groups.

Iterative removal of edges from minimal asymmetric graphs creates a huge number of connected non-isomorphic graphs. However, we also learned the interesting fact that by removing edges from minimal asymmetric graphs, both symmetric and asymmetric graphs can be created. These asymmetric graphs may not be minimal, which was quite an interesting finding.

By iteratively adding edges to the minimal asymmetric graphs, we obtained symmetric, asymmetric but not minimal or minimal asymmetric graphs. Adding edges to a graph without adding another vertices is limited, and when we added the maximum number of edges to the graph, we got symmetric non-oriented simple graphs with very large automorphism groups.

In this bachelor thesis we work with a console application. In the future, we could come up with a user-friendly GUI or a mobile application. The application is written in the Kotlin language, which is currently used for the development of mobile applications

and therefore is a good foundation for possible completion as a mobile application.

The application can be extended with the possibility of exploring all graphs and not just minimal asymmetric graphs. One of the challenges for the future work would be to solve a memory space problem. Our analysis of minimal asymmetric graphs and complete catalogue of induced subgraphs and supergraphs is a good starting point to the study the general asymmetric graphs.

Bibliography

[1] Joseph A. Gallian. *Contemporary Abstract Algebra.* Brooks/Cole Publishing Co., 2010.

[2] Ralph P. Grimaldi. *Discrete and combinatorial mathematics.* Pearson Education, Inc., 2004.

[3] Tatiana B. Jajcayová and Martin Masár. Computer assisted search for graphs with prescribed degrees and cycle structure. In *APLIMAT : 16th Conference on Applied Mathematics*, pages 694–703. STU, 2017.

[4] Jaroslav Nešetřil and Gert Sabidussi. Minimal asymmetric graphs of induced length 4. *Graphs and Combinatorics*, 8(4):343–359, 1992.

[5] Nóra Szakács Robert Jajcay, Tatiana Jajcayová and Mária B. Szendrei. Inverse monoids of partial graph automorphisms. *Journal of Algebraic Combinatorics*, 53(3):829–849, 2021.

[6] Pascal Schweitzer and Patrick Schweitzer. Minimal asymmetric graphs. *Journal of Combinatorial Theory, Series B*, 127(4):215–227, 2016. [Citované 2021-11-29] Dostupné z `https://www.researchgate.net/publication/301896061_Minimal_Asymmetric_Graphs`.

[7] Alexander Stanoyevitch. *Discrete Structures with Contemporary Applications.* Taylor & Francis Books, 2011.

[8] Ronald L. Rivest Clifford Thomas H. Cormen, Charles E. Leiserson and Stein. *Introduction to Algorithms.* Massachusetts Institute of Technology, 2009.

Appendix A: The content of the electronic attachment

The source code can be found in the electronic appendix to the thesis program and experiment results files. The source code is also published on my student website, or I can send it to those who contact me.

Appendix B: Catalogue of minimal asymmetric graphs with iteratively removed vertices

The aim of this bachelor thesis is to measure the degree of asymmetry of an asymmetric graph by the number of vertices which we have to delete to obtain a symmetric graph.

This catalogue shows all minimal asymmetric graphs 3.11 and graphs, from which we iteratively remove vertices to make them symmetric. We obtained these graphs using the GAP program and their visual display using the JupyterViz package. Since the original asymmetric graphs are minimal, each new graph obtained after gradual deletion of vertices has an automorphism group of at least two. All symmetric graphs obtained by removing the same number of vertices from the minimal asymmetric graphs are non-isomorphic to each other. Non-isomorphic graphs may appear in different groups after removing a different number of vertices, which is consistent with our proposal. This helps the program to render better and has a visually better system. Using the GAP program and its GRAPE package, we obtained the automorphism group for each symmetric graph.

3.1 Minimal asymmetric graphs

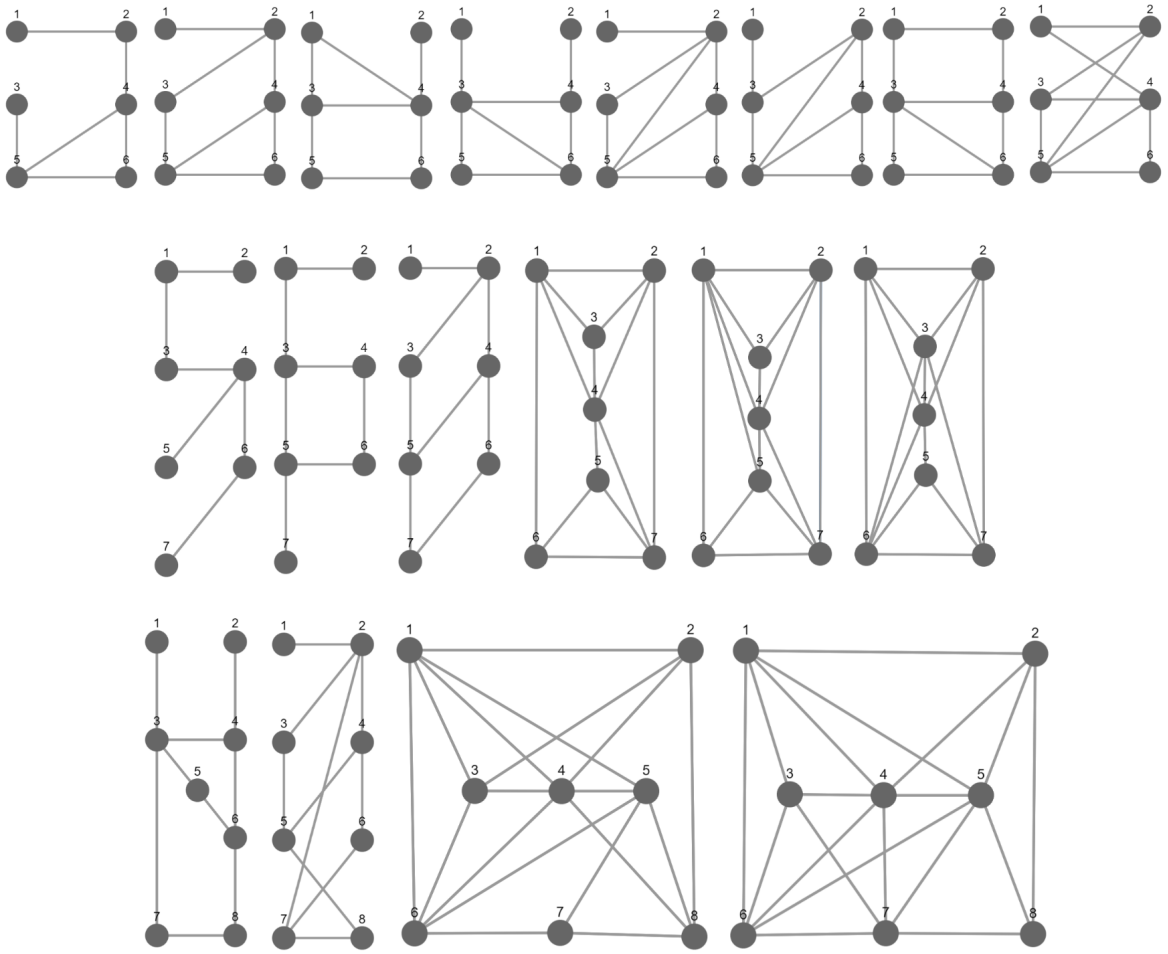

Figure 3.11: Final list of 18 minimal asymmetric graphs. [6]

3.2 Minimal asymmetric graphs without one vertex

Minimal asymmetric graphs without one vertex have an automorphism group mostly of size 2. Using our program, we found forty-four different graphs, which we obtained by deleting one vertex from minimal asymmetric graphs. Of these, nine graphs have an automorphism group of size 4. The other ones have an automorphism group of size 2.

3.2.1 Graphs with an automorphism group of size 2

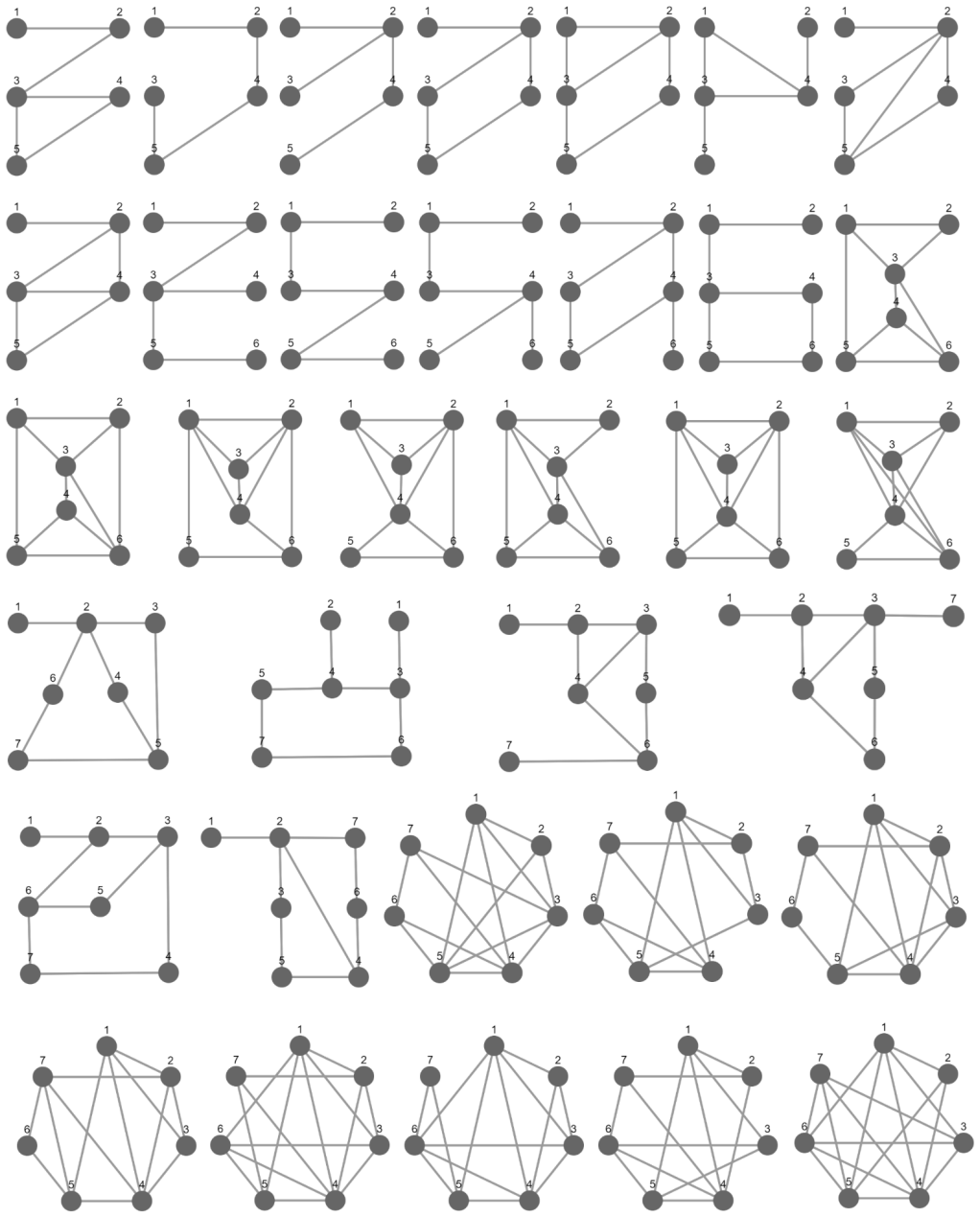

Figure 3.12: List of minimal asymmetric graphs without one vertex with an automorphism group of size 2

3.2.2 Graphs with an automorphism group of size 4

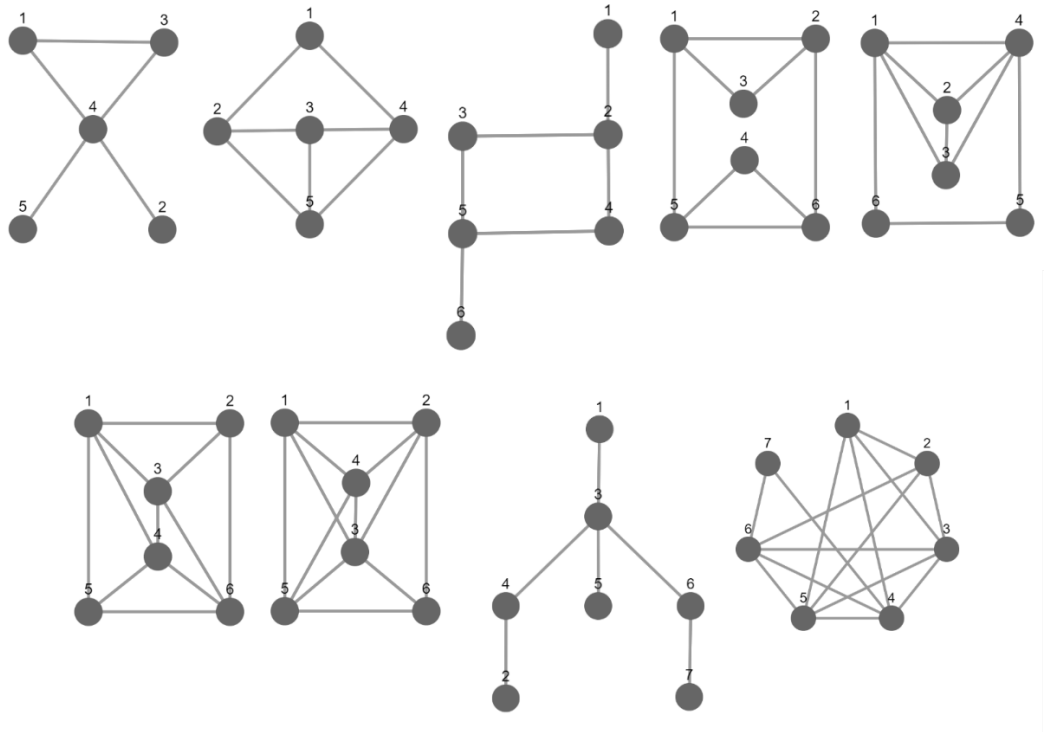

Figure 3.13: List of minimal asymmetric graphs without one vertex with an automorphism group of size 4

3.3 Minimal asymmetric graphs without two vertices

There are thirty-one connected graphs, that we can obtain from minimal asymmetric graphs, when we delete three vertices from all of them. They have an automorphism group size of 2, 4, 6, 8, 12, or 16.

3.3.1 Graphs with an automorphism group of size 2

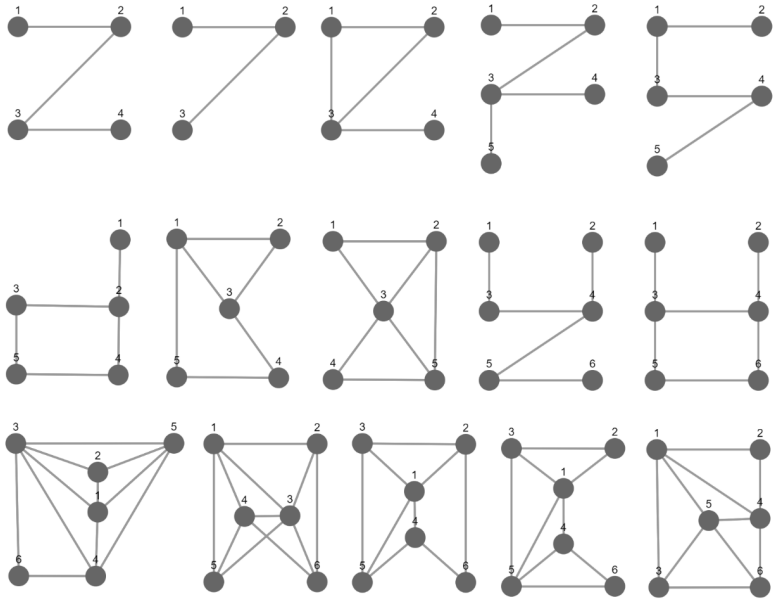

Figure 3.14: List of minimal asymmetric graphs without two vertices with an automorphism group of size 2

3.3.2 Graphs with an automorphism group of size 4

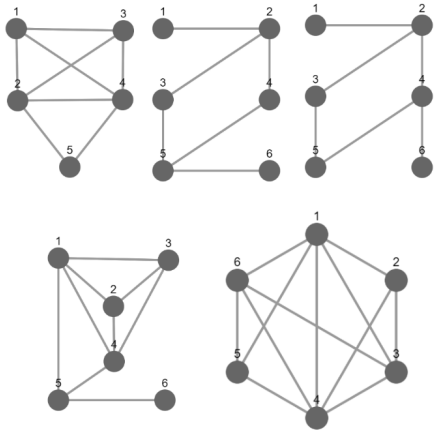

Figure 3.15: List of minimal asymmetric graphs without two vertices with an automorphism group of size 4

3.3.3 Graphs with an automorphism group of size 6

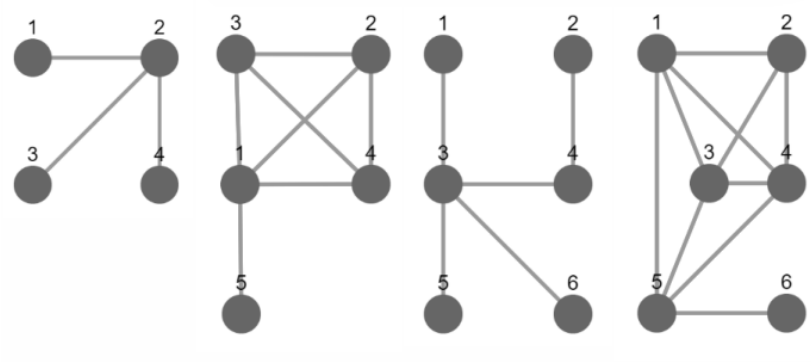

Figure 3.16: List of minimal asymmetric graphs without two vertices with an automorphism group of size 6

3.3.4 Graphs with an automorphism group of size 8

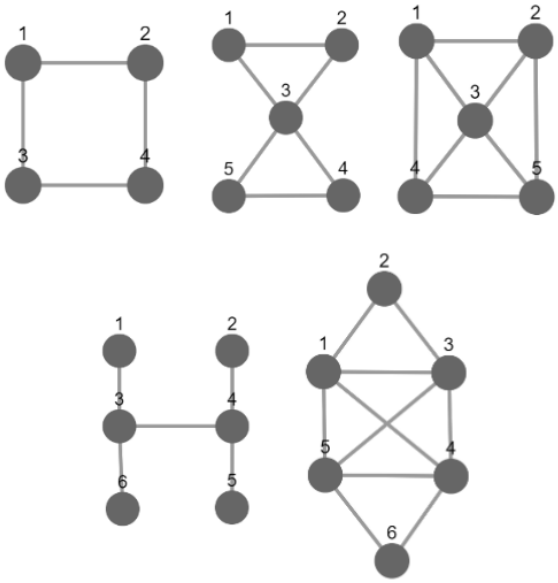

Figure 3.17: List of minimal asymmetric graphs without two vertices with an automorphism group of size 8

3.3.5 Graphs with an automorphism group of size 12

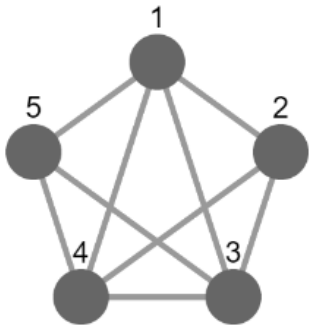

Figure 3.18: Minimal asymmetric graph without two vertices with an automorphism group of size 12

3.3.6 Graphs with an automorphism group of size 16

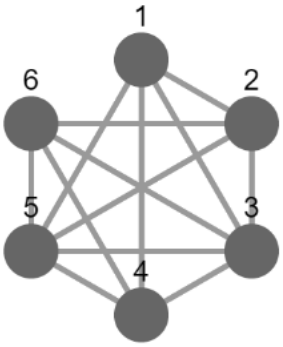

Figure 3.19: Minimal asymmetric graph without two vertices with an automorphism group of size 16

3.4 Minimal asymmetric graphs without three vertices

There are nineteen connected graphs, that can be obtained from minimal asymmetric graphs, when we delete three vertices from all of them. They have an automorphism group size of 2, 4, 6, 8, 12, or 24.

3.4.1 Graphs with an automorphism group of size 2

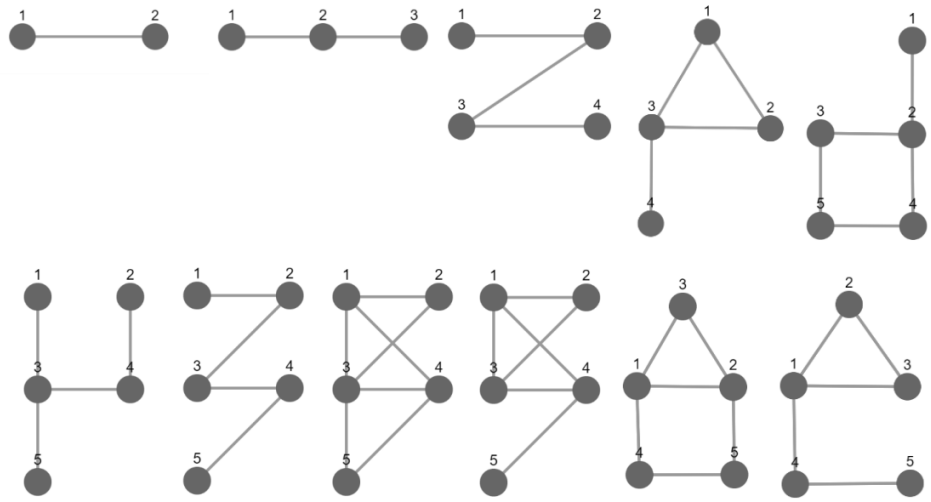

Figure 3.20: List of minimal asymmetric graphs without three vertices with an automorphism group of size 2

3.4.2 Graphs with an automorphism group of size 4

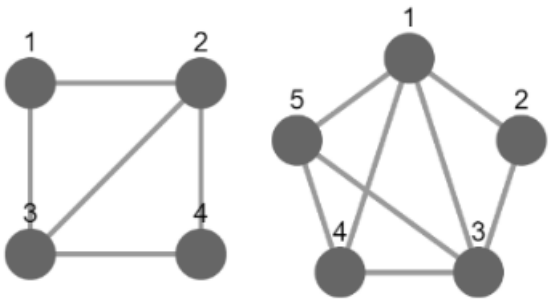

Figure 3.21: List of minimal asymmetric graphs without three vertices with an automorphism group of size 4

3.4.3 Graphs with an automorphism group of size 6

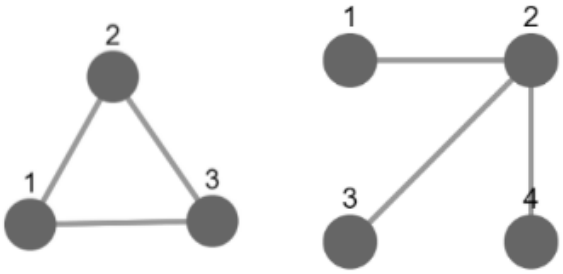

Figure 3.22: List of minimal asymmetric graphs without three vertices with an automorphism group of size 6

3.4.4 Graphs with an automorphism group of size 8

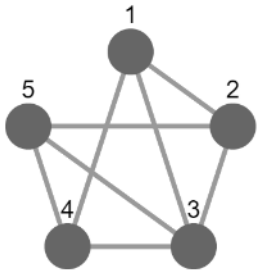

Figure 3.23: Minimal asymmetric graph without three vertices with an automorphism group of size 8

3.4.5 Graphs with an automorphism group of size 12

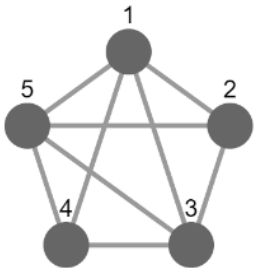

Figure 3.24: Minimal asymmetric graph without three vertices with an automorphism group of size 12

3.4.6 Graphs with an automorphism group of size 24

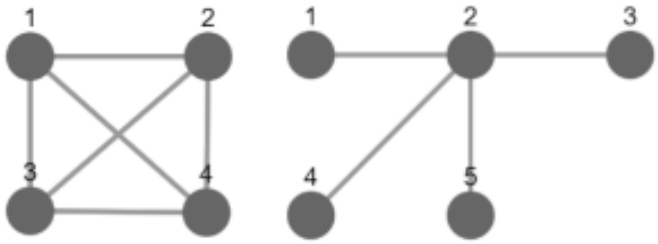

Figure 3.25: List of minimal asymmetric graphs without three vertices with an automorphism group of size 24

3.5 Minimal asymmetric graphs without four vertices

There are eight connected graphs, that we can obtain from minimal asymmetric graphs, when we delete four vertices from all of them. They have an automorphism group size of 2, 4, 6, or 24.

3.5.1 Graphs with an automorphism group of size 2

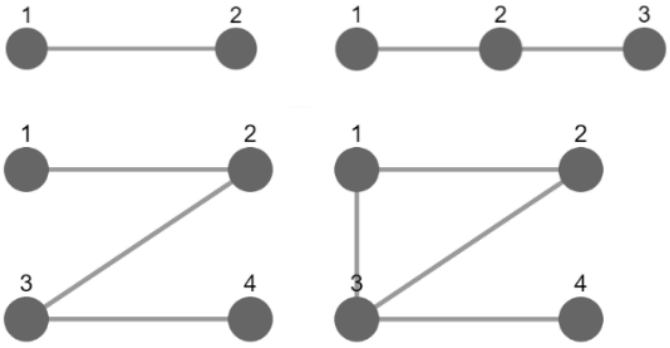

Figure 3.26: List of minimal asymmetric graphs without four vertices with an automorphism group of size 2

3.5.2 Graphs with an automorphism group of size 4

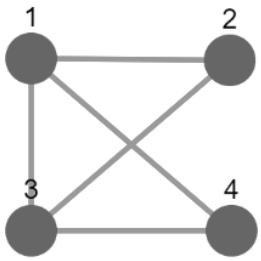

Figure 3.27: Minimal asymmetric graph without four vertices with an automorphism group of size 4

3.5.3 Graphs with an automorphism group of size 6

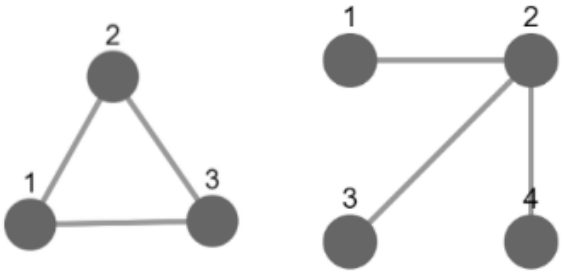

Figure 3.28: List of minimal asymmetric graphs without four vertices with an automorphism group of size 6

3.5.4 Graphs with an automorphism group of size 24

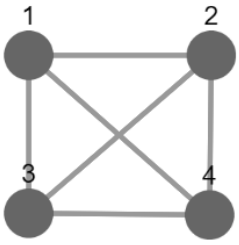

Figure 3.29: Minimal asymmetric graph without four vertices with an automorphism group of size 24

3.6 Minimal asymmetric graphs without five vertices

There are three connected graphs, that can be obtained from minimal asymmetric graphs, when we delete five vertices from all of them. Two of these graphs has an automorphism group of size 2 and one has an automorphism group of size 6.

3.6.1 Graphs with an automorphism group of size 2

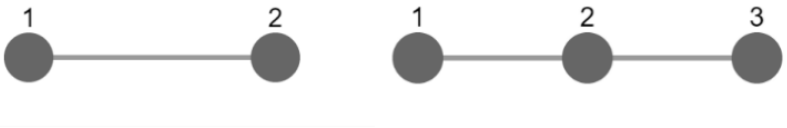

Figure 3.30: List of minimal asymmetric graphs without five vertices with an automorphism group of size 2

3.6.2 Graphs with an automorphism group of size 6

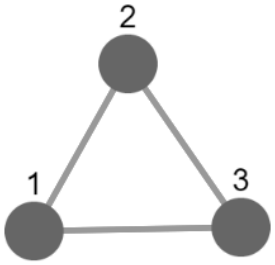

Figure 3.31: Minimal asymmetric graph without five vertices with an automorphism group of size 6

3.7 Minimal asymmetric graphs without six vertices

There is only one connected graph, that can be obtained from minimal asymmetric graphs, when we delete six vertices from all of them. That is caused by the number of vertices, that have minimal asymmetric graphs originally. This graph has an automorphism group of size 2.

3.7.1 Graphs with an automorphism group of size 2

Figure 3.32: Minimal asymmetric graph without six vertices with an automorphism group of size 2

Deleting more vertices, based on the text above, would lead to a graph with 1 vertex.