An Integer Programming Approach to Exact and Fuzzy Symmetry Detection

Inaugural-Dissertation

zur

Erlangung des Doktorgrades

der Mathematisch-Naturwissenschaftlichen Fakultät

der Universität zu Köln

vorgelegt von

Christoph Buchheim

aus Opladen

Köln 2003

Berichterstatter:	**Prof. Dr. Michael Jünger**
	Prof. Dr. Ewald Speckenmeyer
Tag der mündlichen Prüfung:	**3. Juni 2003**

Die Tatsachen gehören alle nur
zur Aufgabe, nicht zur Lösung.

Ludwig Wittgenstein

Zusammenfassung

Das Ziel beim Automatischen Zeichnen von Graphen besteht in der Erzeugung von schönen und übersichtlichen Zeichnungen von Diagrammen, die abstrakt als Menge von Objekten und deren Beziehungen untereinander gegeben sind. Zum Zwecke der Automatisierung werden Diagramme als Graphen modelliert. Darüber hinaus werden formale Kriterien festgelegt, mit deren Hilfe die Qualität einer Zeichnung bewertet werden kann. Empirische Studien zeigen, dass ein wichtiges Kriterium in der Darstellung von Symmetrie besteht. Wenn möglich, sollen die Zeichnungen also symmetrisch sein. Es ist jedoch ein NP-schweres Problem, für einen abstrakt gegebenen Graphen zu entscheiden, ob und wie er symmetrisch gezeichnet werden kann. In dieser Arbeit wird ein Ansatz zur Symmetrieerkennung vorgestellt, der auf Techniken der ganzzahligen Programmierung basiert. Mit diesem Ansatz können für einen beliebigen Graphen Symmetrien sowie andere spezielle Automorphismen bestimmt werden. Da die meisten Graphen aus praktischen Anwendungen jedoch keine exakt symmetrische Zeichnung zulassen, wird zusätzlich eine Erweiterung der Suche auf unscharfe Symmetrien betrachtet.

Danksagung

An dieser Stelle danke ich allen, die diese Arbeit direkt oder indirekt ermöglicht haben. Das gilt vor allem für Prof. Michael Jünger, der dies nicht nur durch seine wissenschaftliche Anleitung getan hat, sondern auch durch sein persönliches Vorbild und nicht zuletzt durch die Schaffung einer entspannten und zugleich anregenden Arbeitsumgebung. Für diese sehr angenehme Atmosphäre und für all die andere Hilfe danke ich auch Michael Belling, Matthias Elf, Stefan Hachul, Thomas Lange, Frauke Liers, Ursula Neugebauer und Merijam Percan.

Weiterhin danke ich den Kollegen von Caesar, vor allem Dr. Sebastian Leipert, der mich auf die richtige Bahn gebracht hat. Mein Dank gilt auch Prof. Petra Mutzel und ihren Mitarbeiterinnen und Mitarbeitern für die schöne und produktive Zeit in Wien, sowie Prof. Peter Eades und seiner Gruppe für drei spannende Monate in Sydney. Insbesondere danke ich Dr. Seok-Hee Hong, von der ich viel gelernt habe.

Genauso dankbar bin ich allen, die mein Leben außerhalb der Wissenschaft lebenswert machen: Meinen Eltern, meiner Schwester und allen Freundinnen und Freunden, die verhindert haben, dass ich mich zu sehr in meine Forschung vergrabe und dabei den Überblick verliere.

Am meisten danke ich Dr. Jana Kremer.

Abstract

The aim of Automatic Graph Drawing is to compute nice and well-readable layouts of diagrams given abstractly as a set of entities and a set of relations between them. For the sake of automation, diagrams are modeled by graphs. Moreover, formal criteria are specified that measure the quality of such layouts. Empirical studies show that one important criterion is the display of symmetry: if they exist, symmetric layouts are preferred. However, it is an NP-hard problem to decide whether and how an abstractly given graph can be drawn in a symmetrical way. We present an approach for detecting symmetries based on integer programming techniques. This approach can be used to find symmetries or different kinds of automorphisms for general graphs. As most practical graphs do not admit any exact symmetry, we also consider an extension for detecting fuzzy symmetries.

Acknowledgments

I wish to express my thanks to everyone who supported me in writing this thesis, whether directly or indirectly. This applies in particular to Michael Jünger, who did this not only by his scientific advice but also by his personal example and not least by providing a relaxed and stimulating working environment. For this most pleasant atmosphere and for all the other help, I am also grateful to Michael Belling, Matthias Elf, Stefan Hachul, Thomas Lange, Frauke Liers, Ursula Neugebauer, and Merijam Percan.

Furthermore, I would like to thank the guys at Caesar, mainly Sebastian Leipert, who put me on the right track. Many thanks also to Petra Mutzel and her staff for the nice and fruitful time in Vienna, and to Peter Eades and his group for three exciting months in Sydney. I am especially grateful to Seok-Hee Hong for learning much from her.

I am just as indebted to everyone making my life outside the science world worth living: to my parents, to my sister, and to all my friends who prevented me from digging myself into the research too deep to keep the overview.

Most of all, I am thankful to Jana Kremer.

Contents

Introduction

The concept of symmetry is a central element in most attempts to understand and formalize aesthetic perception. Greek philosophy connected symmetry to beauty and perfection, or even divinity. More recent theories also emphasize the importance of symmetry; see e.g. George Birkhoff's aesthetic measure [12] or Max Bense's information-theoretic aesthetics [9]. In these theories, the beauty of an object is defined to be more or less proportional to its degree of symmetry and order.

In addition to its aesthetic relevance, symmetry is often preferred for practical reasons. On one hand, it may be imposed by nature, as in architecture, where statics requires symmetry to some extent. On the other hand, symmetric structures are recognized much easier and faster than asymmetric structures of the same complexity. According to Gestalt theory, symmetric structures are preferred in case of ambiguity because of their simplicity; see Fig. 1 for an example taken from [69].

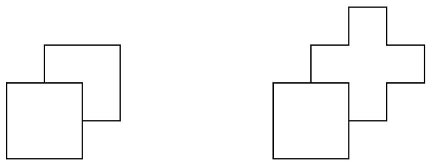


Figure 1: A simple example showing the preference of symmetry. In both drawings, we do not know whether and how the two figures cover each other. However, in the left hand drawing we see two squares, whereas in the right hand drawing we see a square and a cross

For both its aesthetic and perceptual importance, symmetry plays a central role in automatic graph drawing. In this research field, the aim is to develop algorithms for creating nice and well-readable drawings of abstractly given graphs. Nodes are represented by points in the plane and edges are represented by curves connecting the corresponding nodes; see Fig. 2 for examples. The quality of a graph drawing can be measured by many different—and usually conflicting—criteria, e.g., the number of crossings of edges, the number of bends of edges, the required area.

One of the most important criteria in automatic graph drawing is the display of symmetry, as emphasized by empirical studies [67]. We call a drawing of a graph G symmetric if there exists a non-trivial isometry of the plane that fixes the drawing, i.e., that maps nodes to nodes and curves to curves. Examples are given in Fig. 2(b) and Fig. 2(c); these drawings are fixed under rotations of order six, i.e., under rotations by 60 degrees.

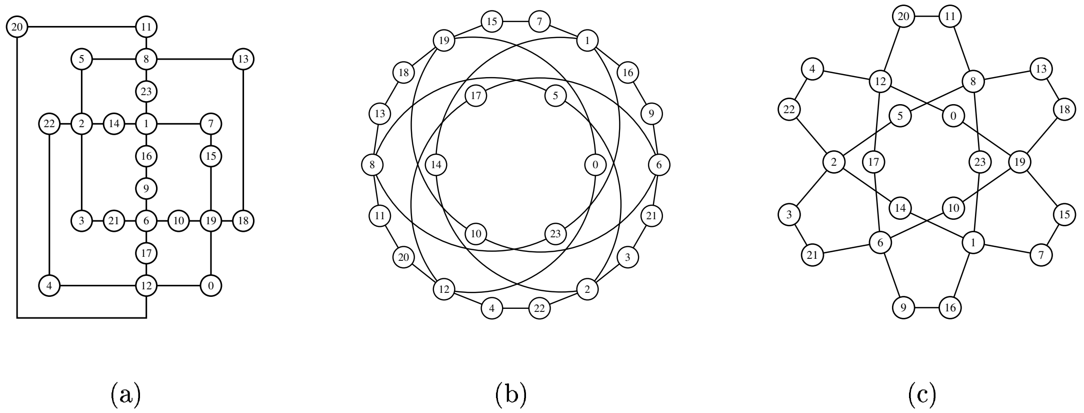


Figure 2: Three drawings of the same abstract graph

It is a well-known result that every isometry of the plane is a composition of rotations at a center point, reflections at an axis, and translations by a vector. Since translations cannot fix a graph drawing, every symmetry is a compositions of rotations and reflections. In fact, such compositions are rotations or reflections again, so we derive that every symmetric graph drawing is fixed by a rotation or a reflection. Fig. 2(b) and Fig. 2(c) are fixed by both rotational and reflectional symmetries.

As argued above, an algorithm for the problem of drawing a given abstract graph G in a symmetrical way is desirable. Usually, this problem is divided into the following two subproblems: first, find an abstract symmetry of G. Such an abstract symmetry does not determine the actual node positions but only the node permutation induced by a symmetric drawing, i.e., which node is mapped to which other node if the corresponding isometry of the plane is applied to the graph drawing. On one hand, by fixing this node permutation, the number of possible drawings of G is restricted significantly. On the other hand, even for a given abstract symmetry, we still have many possible drawings; e.g., the two drawings in Fig 2(b) and 2(c) show the same graph and display the same symmetries. Finding a good drawing of a given symmetry is the second subproblem. Here we have to consider further aesthetic criteria like crossing minimization; see Buchheim and Hong [17]. However, finding just any drawing of G realizing a given abstract symmetry is a trivial problem. In the following, we focus on the first subproblem, the problem of symmetry detection.

In his PhD-thesis [53], Joseph Manning showed that the symmetry detection problem is NP-hard for general graphs. Furthermore, he devised linear time algorithms for symmetry detection in trees and outerplanar graphs; see [55] and [56]. Finally, he gives an algorithm for finding symmetries in embedded graphs. Building on this work, Seok-Hee Hong and Peter Eades were able to find a linear time symmetry detection algorithm for general planar graphs. For the triconnected case, Hong et al. [42] show how an algorithm of Fontet [31] can be used to fix one of the linearly many embeddings of G so that the result for embedded graphs can be used. The biconnected case is reduced to the triconnected case by using SPQR-trees [39]. Similarly, the one-connected case is reduced to the biconnected case by using block-cut-trees [40]. Finally, the disconnected case is considered in [41].

Three significant differences between the approach of Manning and Hong et al. on one hand and our approach on the other hand have to be pointed out. First, Manning and Hong only search for planar symmetries, i.e., symmetries that can be displayed by a planar drawing of G. However, even in planar graphs, there may be a lot of non-planar symmetries. For example, all planar symmetries of the complete graph on four nodes are reflections or rotations of order three, whereas we have a non-planar rotation of order four; see Fig. 3. Second, Manning and Hong not only search for a single symmetry but a maximum size group of symmetries that can be displayed simultaneously by a single drawing of G. In our approach, we search for a single symmetry of maximal order. Finally, in the reflection case, Manning and Hong only allow disjoint unions of paths as fixed subgraphs. The reason is that these subgraphs can be drawn on the reflection axis without overlapping edges. We drop this restriction by allowing arbitrary fixed subgraphs.

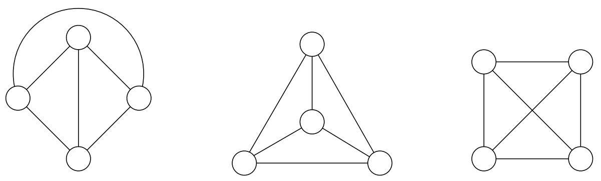


Figure 3: Symmetric drawings of the complete graph on four nodes. The first drawing is planar and fixed by a reflection, the second one is planar and fixed by a rotation of order three. The last drawing shows a rotational symmetry of order four that cannot be displayed by any planar drawing of the graph

The NP-completeness of the symmetry detection problem basically leaves three ways to go on: to show P = NP, to develop heuristic algorithms, or to find exact algorithms running in exponential time in general but fast enough in practice. Finding heuristic algorithms is difficult, since the structure of the problem is hardly compatible with greedy or similar strategies. In fact, changing a single edge in a symmetric graph usually results in an asymmetric graph. However, there are a few heuristic algorithms trying to compute symmetric drawings: the spring embedder has a tendency to display symmetries; see Eades and Lin [29]. A heuristic algorithm specifically designed for symmetric graph drawing was devised by de Fraysseix [26].

The first exact symmetry detection algorithm for general graphs, based on the branch & cut-technique, was proposed by Buchheim and Jünger [18, 19]. The problem is modeled by an integer linear program (ILP) containing a binary variable for each pair of nodes. This variable has value one if and only if the first node is mapped to the second one by the represented node permutation. In other words, the symmetric group of the node set of G is identified with a group of permutation matrices. In the basic ILP, the solutions correspond to automorphisms of G. These are defined as node permutations preserving adjacency. Any linear objective function can be combined with the automorphism ILP, allowing for example to find automorphisms minimizing the number of fixed nodes. By assigning appropriate objective function coefficients, we can also forbid to map a node to certain other nodes. Both problems are NP-hard as shown by Lubiw [51]. We describe the automorphism detection algorithm in Chapter 2.

Obviously, every symmetry of G is an automorphism of G, but the converse is not true. To detect symmetries, we further restrict the automorphism ILP to exclude automorphisms that do not form symmetries. We do this separately for rotations of a fixed order in Chapter 3 and for reflections in Chapter 4. For finding a symmetry of maximum order, we can combine both cases. This is explained in Chapter 5, where we also include runtime results. Our algorithm runs very fast for most graphs.

In the meantime, another exact algorithm for symmetry detection in general graphs was presented by Abelson et al. [2]. This approach is based on a group theoretic characterization of symmetries and uses the commercial algebra system Magma [52] to find appropriate groups. This approach runs faster than the branch & cut-algorithm. Nevertheless, our approach leaves much space for improvement.

However, the main advantage of the branch & cut-approach over the group-theoretic method is its flexibility. Not only can we use arbitrary objective functions, we can also extend or restrict the set of feasible solutions by adding new variables or linear constraints to the ILP. An important extension is fuzzy symmetry detection. This is motivated by the fact that most graphs do not admit any exact symmetry at all. By changing the adjacency status of certain pairs of nodes, i.e., by deleting or creating edges, every graph can be made symmetric. If a small number of modifications suffices, it may still be useful to create a drawing displaying this fuzzy symmetry; see Fig. 4 for examples. In Chapter 6, we present a modified branch & cut-approach finding the closest graph to G that admits a symmetry of a fixed order. Observe that other algorithms like the group-theoretic approach or the spring embedder cannot be adjusted to recognize fuzzy symmetries.

In spite of the widespread conviction that fuzzy symmetry detection is much more important than exact symmetry detection, earlier research on this problem is rare. As far as we know, the only publication concerning this problem is by Chen et al. [23], mainly containing negative complexity results. In fact, fuzzy symmetries are much harder to determine than exact symmetries; this is true both theoretically and practically. Consequently, our runtime figures for the fuzzy approach cannot compete with the figures for exact symmetry detection.

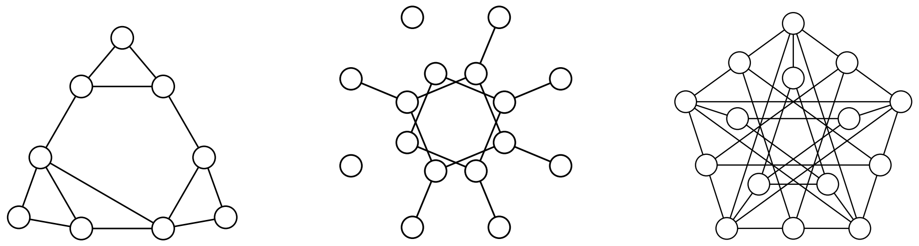


Figure 4: Some fuzzy symmetries

Before we can start with automorphism detection in Chapter 2, we have to fix some notation concerning graphs and homomorphisms in Chapter 1. There, we also introduce accurate definitions of the problems to be tackled and give more information about their complexity status. A short description of the branch & cut-technique is also included. In the conclusion, we give a summary of our results and discuss possible extensions and future work.

Chapter 1

Preliminaries

On the following pages, we collect definitions and basic facts used throughout the remaining chapters. We discuss problems and results established in the literature; furthermore, we define some basic technical constructions used at several points in the branch & cut-approach to be developed.

We first fix notation about graphs and their homomorphisms in Sect. 1.1. Then we define the problems we are going to solve by branch & cut and give some background on their complexity status and related problems. These problems all ask for special graph homomorphisms: in Sect. 1.2, we start with the notorious graph isomorphism problem. In Sect. 1.3, we consider general graph automorphisms, whereas geometric automorphisms are discussed in Sect. 1.4. Finally, a brief description of the branch & cut-technique is given in Sect. 1.5.

1.1 Graphs

In general, we define a *graph* as a pair $G = (V, c)$, where V is a finite set of *nodes* and c is a *coloring* of V^2, i.e., a map $V^2 \to \mathbb{N}$. For any coloring c of V^2, we define a *node-coloring* $c\colon V \to \mathbb{N}$ by $c(i) = c(i,i)$. A coloring c' is *finer* than a coloring c if $c'(i_1, j_1) = c'(i_2, j_2)$ implies $c(i_1, j_1) = c(i_2, j_2)$. For the sake of simplicity, we always assume $V = \{1, \ldots, n\}$ in the following. Observe that the graph $G = (V, c)$ is given by its *adjacency matrix* A_G, where $(A_G)_{ij} = c(i,j)$.

This definition of graphs is more general than many other common definitions: A *simple graph* $G = (V, E)$ can be viewed as a graph $G = (V, c)$ by setting

$$c(i,j) = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i \neq j \text{ and } (i,j) \in E \\ 2 & \text{if } i = j\ . \end{cases}$$

An *undirected graph* corresponds to a graph $G = (V, c)$ with $c(i,j) = c(j,i)$ for all $i, j \in V$, i.e., a graph with a symmetric adjacency matrix. For a *multigraph*, we can define $c(i,j)$ as the number of edges from i to j, if $i \neq j$, and $c(i,i)$ as the number of loops of i. Our class of graphs agrees with the class of directed multigraphs that may contain loops (and multiloops). For technical reasons, we prefer to use the definition given above.

If $G = (V, c)$ and $G' = (V', c')$ are two graphs, a *homomorphism* $G \to G'$ is a map $\pi\colon V \to V'$ with $c(i,j) \leq c'(\pi(i), \pi(j))$ for all $i, j \in V$. For simple graphs, this means that each pair of adjacent nodes in G is either mapped to the same node or to an adjacent pair of nodes in G'.

The graphs and their homomorphisms form a category. In this category, a *monomorphism* is an injective homomorphism, an *epimorphism* is a surjective homomorphism, and an *isomorphism* is a bijective homomorphism. Two graphs G and G' are called *isomorphic* if there is an isomorphism $G \to G'$. An *endomorphism* of G is a homomorphism $G \to G$ and an *automorphism* is an isomorphism $G \to G$. The *trivial automorphism* of G is given by the identity map $\mathrm{id}_V\colon V \to V$.

An *(edge-induced) subgraph* of G is a graph $G' = (V', c')$ together with a monomorphism $G' \to G$, i.e., an injective map $\pi\colon V' \to V$ with $c'(i,j) \leq c(\pi(i), \pi(j))$ for $i, j \in V$. If we even have $c'(i,j) = c(\pi(i), \pi(j))$ for all $i, j \in V$, the subgraph G' is *node-induced*; this is equivalent to the fact that $A_{G'}$ is a submatrix of A_G.

The *complete graph* K_n is the simple undirected graph on n nodes such that all edges between different nodes are present, i.e., the graph with coloring

$$c(i,j) = \begin{cases} 1 & \text{if } i \neq j \\ 2 & \text{if } i = j\ . \end{cases}$$

Similarly, the *edgeless graph* E_n is the graph on n nodes without edges, i.e., the graph with coloring

$$c(i,j) = \begin{cases} 0 & \text{if } i \neq j \\ 2 & \text{if } i = j\ . \end{cases}$$

Finally, the *cycle graph* C_n is given by $V = \{1, \ldots, n\}$ and the coloring

$$c(i,j) = \begin{cases} 1 & \text{if } i = (j \bmod n) + 1 \text{ or } j = (i \bmod n) + 1 \\ 2 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

1.2 Isomorphisms of Graphs

For both theoretical and practical reasons, the problem of deciding whether a given pair of graphs is isomorphic has attracted much attention:

(GI)	Given two graphs G and G'. Decide whether there exists an isomorphism $G \to G'$.

The complexity status of the graph isomorphism problem (GI) is unknown. No polynomial time algorithm for the general case is known. On the other hand, (GI) is not known to be NP-complete. Many researchers believe that the complexity of (GI) is intermediate, i.e., neither polynomial nor NP-complete. Such problems exist if P $\neq$ NP, as shown by Ladner [49]. However, no such problem has been found yet; graph isomorphism is one of the hottest candidates. Problems polynomial time equivalent to (GI) are called *isomorphism-complete.*

In fact, (GI) is polynomial time solvable for large classes of graphs. For an overview, see Read and Corneil [68]. Even for the general case, there are algorithms working fast enough in practice, the most popular one being `nauty` by McKay [60].

1.3 Automorphisms of Graphs

As defined in Sect. 1.1, an automorphism of G is an isomorphism $G \to G$. The underlying map $V \to V$ is a permutation of V in this case. We gather notation concerning permutations in Sect. 1.3.1. In Sect. 1.3.2, we define basic problems related to graph automorphisms. Finally, we introduce some technical constructions needed in the following chapters: the automorphism partitioning in Sect. 1.3.3, the coloring graph in Sect. 1.3.4, and the orbit graphs in Sect. 1.3.5.

1.3.1 Permutations

Let V be a finite set with $|V| = n$. A *permutation* of V is a bijective map $V \to V$. The set of permutations forms a group under composition, the *symmetric group* S_V; its neutral element is the identity id_V. We may denote S_V by S_n, since the symmetric groups of sets of the same cardinality are isomorphic. For $n \geq 3$, the group S_n is not Abelian. Any subgroup of S_n is called a *permutation group* with *domain* V. We have $|S_n| = n!$. Nevertheless, every permutation group with domain V can be generated by $\lfloor n \log_2 n \rfloor$ elements; see e.g. Hoffmann [38], Theorem 2.5.

The order of a permutation π is defined by $\mathrm{ord}(\pi) = |\langle\pi\rangle|$, where $\langle\pi\rangle$ is the cyclic subgroup of S_n generated by π, i.e., the group $\{\pi^e \mid e \in \mathbb{Z}\}$. We have $\mathrm{ord}(\pi) = 1$ if and only if $\pi = \mathrm{id}_V$. The set of *fixed* elements of π is $\mathrm{Fix}(\pi) = \{i \in V \mid \pi(i) = i\}$.

For any subgroup H of S_n, the *orbit* of i under H is $\mathrm{orb}_H(i) = \{\pi(i) \mid \pi \in H\}$. For $\pi \in S_n$, the orbit of i under π is the set $\mathrm{orb}_\pi(i) = \mathrm{orb}_{\langle\pi\rangle}(i) = \{\pi^e(i) \mid e \in \mathbb{Z}\}$. We have $i \in \mathrm{Fix}(\pi)$ if and only if $|\mathrm{orb}_\pi(i)| = 1$. An orbit consisting of a single fixed node is called *trivial*.

Now let $t \in \mathbb{N}$. A permutation π of V defines a permutation of the t-tuples V^t, by setting $\pi_t(i_1, \ldots, i_t) = (\pi(i_1), \ldots, \pi(i_t))$. The orbits of π_t are called *t-orbits* of π; the 2-orbits are called *orbitals*. We have $\mathrm{Fix}(\pi_t) = \mathrm{Fix}(\pi)^t$. We will denote π_t by π, too, since no confusion can arise. By this identification, a permutation group H with domain V gives rise to an isomorphic permutation group H_t with domain V^t.

A *k-rotation* is a permutation π of V such that $|\mathrm{Fix}(\pi)| \leq 1$ and $|\mathrm{orb}_\pi(i)| = k$ for all $i \notin \mathrm{Fix}(\pi)$. Every k-rotation has order k if $n \geq 2$. A *reflection* is a permutation π of V such that $\pi^2 = \mathrm{id}_V$. Every non-trivial reflection has order 2. The identity id_V is both a 1-rotation and a reflection. Every 2-rotation is a reflection, but the converse is not true, as reflections may fix more than one node.

There are two different methods of writing down a permutation. In the so-called *two-row table representation*, the elements of V are placed above their images. In the *cyclic representation*, the orbits of the permutation are displayed one after another, enclosed by brackets. Trivial orbits are omitted. For example, the two following representations correspond to the same permutation:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 5 & 8 & 3 & 2 & 1 & 7 & 4 & 6 \end{pmatrix} \qquad (1\ 5)(2\ 8\ 6\ 7\ 4)$$

More facts about finite permutation groups can be found in Wielandt [73] or, with a view towards combinatorial structures, Biggs and White [10].

1.3.2 Automorphisms

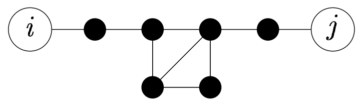
Let $G = (V, c)$ be a graph. As defined in Sect. 1.1, an automorphism of G is an isomorphism $G \to G$, i.e., a permutation π of V such that $c(i, j) = c(\pi(i), \pi(j))$ for $i, j \in V$. The set of automorphisms of G is denoted by $\operatorname{Aut} G$. Obviously, the set $\operatorname{Aut} G$ forms a group with respect to composition. More precisely, $\operatorname{Aut} G$ is a subgroup of S_n and hence a permutation group with domain V. The automorphism group of the edgeless graph E_n and the complete graph K_n is S_n. In fact, every finite group is isomorphic to the automorphism group of some graph; this is even true for strongly regular graphs; see Mendelsohn [61].

Some classes of graphs are defined by means of their automorphism group. To give an example, a graph G is called *transitive* if the group $\operatorname{Aut} G$ is transitive, i.e., if for every $i, j \in V$ there is an automorphism π of G mapping i to j. For another example, a *Cayley graph* over some finite group H is a graph $G = (H, c)$ such that H is a subgroup of $\operatorname{Aut} G$. More background on these classes of graphs is given by Godsil and Royle [32].

In the following, we state complexity results related to automorphisms. Since most of these results refer to simple graphs in the literature, we need the following

Lemma 1.1
For a given graph $G = (V, c)$, one can construct a simple undirected graph G' without loops in quadratic time such that $\operatorname{Aut} G = \operatorname{Aut} G'$.

Proof: As always, assume $V = \{1, \ldots, n\}$. We may assume $c(V^2) \subseteq \{1, \ldots, n^2\}$, otherwise we can recolor the graph without touching automorphisms. We start constructing G' by taking over the nodes from V. For all $(i, j) \in V^2$, we add the following graph $c(i, j)$ times between i and j:



Clearly, every automorphism of G induces an automorphism of G'. It is easy to verify that we do not get additional automorphisms by this construction. □

This construction is even linear if G is considered a multigraph, i.e., if we assume that $c(i, j)$ edges between i and j are part of the input. Since simple graphs are included in the general definition, we conclude that the problems discussed in the following are polynomial time equivalent for every class of graphs between simple undirected graphs and general graphs according to our definition. In particular, complexity results given in the literature remain valid in our setup.

Now consider the graph automorphism problem:

(Aut)	Given a graph G. Decide whether there exists a non-trivial automorphism of G.

As for (GI), the exact complexity of (Aut) is unresolved. No polynomial time algorithm is known, but (Aut) is at most as hard as the graph isomorphism problem; however, (Aut) is not known to be isomorphism-complete. Problems polynomially equivalent to (Aut) are called *automorphism-complete*. So we have the following hierarchy of complexity classes:

$$\text{P} \subseteq \text{automorphism-complete} \subseteq \text{isomorphism-complete} \subseteq \text{NP-complete} .$$

Trying to compute the complete automorphism group of a graph in polynomial time is a hopeless task, since it may contain up to $n!$ elements. But $\operatorname{Aut} G$ can be generated by $\lfloor n \log_2 n \rfloor$ elements, so a natural question is whether one can compute a set of generators in polynomial time. Unfortunately, the following two problems are isomorphism-complete, as shown by Hoffmann [38]:

(AutG)	Given a graph G. Compute a set of generators for $\operatorname{Aut} G$.

(#Aut)	Given a graph G. Compute the number of automorphisms of G.

The following problems are also related to (Aut):

(Aut1R)	Given a graph G and a node i of G. Decide whether there exists an automorphism π of G with $\pi(i) \neq i$.

(Aut-R)	Given a graph $G = (V, c)$ and a set $R \subseteq V^2$. Decide whether there is an automorphism π of G with $\pi(i) \neq j$ for all $(i, j) \in R$.

(Aut$_0$)	Given a graph G. Decide whether there exists an automorphism of G without fixed nodes.

(2-Aut$_0$)	Given a graph G. Decide whether there exists an automorphism of G of order two without fixed nodes.

Lubiw [51] showed that the problem (Aut1R) is isomorphism-complete, whereas the problems (Aut-R), (Aut$_0$), and (2-Aut$_0$) are NP-complete.

For a lot of restricted graph classes, the problem (GI) and hence (Aut) and (AutG) can be solved in polynomial time, e.g., for planar graphs [43], graphs of bounded degree [38], graphs of bounded eigenvalue multiplicity [3], partial k-trees [13], and graphs of bounded average genus [24].

1.3.3 Automorphism Partitionings

When dealing with automorphisms of graphs or other combinatorial structures, many algorithms can be improved significantly by applying labeling algorithms in a preprocessing step. Fine labelings decrease the number of permutations that have to be considered as automorphism candidates. For $t \in \mathbb{N}$, a *t-labeling* of $G = (V, c)$ is defined as a coloring $c' \colon V^t \to \mathbb{N}$ such that for every $\pi \in \operatorname{Aut} G$ and $i_1, \ldots, i_t \in V$ we have $c'(i_1, \ldots, i_t) = c'(\pi(i_1), \ldots, \pi(i_t))$. In other words, a t-labeling is a coloring that does not decrease the automorphism group.

The finest possible t-labeling c_t of G is called the *t-automorphism partitioning* of G. For $t = 1$, the t-automorphism partitioning is the usual *automorphism partitioning* of G. Unfortunately, the problem of computing the automorphism partitioning of a graph is isomorphism-complete:

(PRT)	Given a graph G. Compute the automorphism partitioning of G.

A simple proof of the isomorphism-completeness is given by Read and Corneil [68]. Nevertheless, there are polynomial time heuristics computing very fine labelings in general; see for example Bastert [7]. Furthermore, there are exact algorithms running fast in practice like `nauty`.

Next, let $t = 2$. By definition, the given coloring c of the graph G is a 2-labeling of G. At many points of our algorithm, it is favorable to have a fine coloring c, since this reduces the number of a priori possible automorphisms. Hence it is useful to replace c by a 2-labeling of G as fine as possible, if not the 2-automorphism partitioning. By definition of 2-labelings, this does not change the set of automorphisms or symmetries of G. In our algorithm, we do this in a preprocessing step.

The t-automorphism partitioning can be computed from a given set of generators of $\operatorname{Aut} G$ in the following way: we use disjoint dynamic sets. First, we consider all subsets of V^t containing exactly one element. Next, for each of the generators π of $\operatorname{Aut} G$ and for all $(i_1, \ldots, i_t) \in V^t$, we merge the subset of V^t containing $(i_1, \ldots, i_t)$ with the subset containing $(\pi(i_1), \ldots, \pi(i_t))$. It is easy to see that the resulting partitioning of V^t is the t-automorphism partitioning of G. For fixed t, the runtime of this algorithm is polynomial in the number of generators of the automorphism group of G. In particular, we derive that (PRT) and the corresponding problems for fixed $t \geq 2$ can be reduced to (AUTG) in polynomial time and are hence all isomorphism-complete.

For the branch & cut-algorithms presented in the following chapters, we use `nauty` to compute a set of generators of the group $\operatorname{Aut} G$. From these generators, we determine the 2-automorphism partitioning of G. This is all done in a preprocessing step.

1.3.4 The Coloring Graph

For a graph $G = (V, c)$, we define its *coloring graph* as the simple undirected graph $G' = (V^2, E')$ where $((i_1, j_1), (i_2, j_2)) \in E'$ if and only if $c(i_1, i_2) = c(j_1, j_2)$. Observe that G' may have loops. We will use this graph at several points in our algorithm. Its importance is emphasized by the following results:

Lemma 1.2
The maximal cardinality of a clique in G' is n. There is a one-to-one correspondence between the cliques in G' of maximal cardinality and the set $\operatorname{Aut} G$.

Proof: First observe that $\{(i,i) \mid i \in V\}$ is a clique in G' with cardinality n. On the other hand, we may assume that $c(i,i) \neq c(j,k)$ for $j \neq k$ (otherwise the coloring c may be replaced by another one satisfying this condition), hence for every clique Q and every $i \in V$ we have $|\{j \in V \mid (i,j) \in Q\}| \leq 1$, so that

$$|Q| = \sum_{i \in V} |\{j \in V \mid (i,j) \in Q\}| \leq n \ .$$

For $\pi \in \operatorname{Aut} G$, define $Q_\pi = \{(i, \pi(i)) \mid i \in V\}$. The set $Q_\pi \subseteq V^2$ induces a clique of size n in G'. Conversely, for every clique Q in G' with cardinality n and every $i \in V$, there is exactly one node $j \in V$ with $(i,j) \in Q$. Analogously, for every $j \in V$, there is exactly one node $i \in V$ with $(i,j) \in Q$. Hence Q defines a node permutation π_Q of G. For $i_1, i_2 \in V$, we have $(i_1, \pi_Q(i_1)), (i_2, \pi_Q(i_2)) \in Q$, hence $c(i_1, i_2) = c(\pi_Q(i_1), \pi_Q(i_2))$. Thus $\pi_Q \in \operatorname{Aut} G$. Obviously, the maps $\pi \mapsto Q_\pi$ and $Q \mapsto \pi_Q$ are inverse, hence the result. □

Lemma 1.3
Let $d: V^2 \to \mathbb{R}$ and $M = \max\{d(i,j) \mid (i,j) \in V^2\}$. Define weights for the nodes of G' by $w_{(i,j)} = nM + d(i,j)$. Then there is a one-to-one correspondence between the cliques in G' of maximal weight and the automorphisms π of G maximizing the sum $\sum_{i \in V} d(i, \pi(i))$.

Proof: If Q has maximal weight, it has maximal cardinality n by definition of the weights, so every candidate clique Q corresponds to an automorphism π_Q by the last proof. The maximal weight clique Q is the one maximizing

$$\sum_{(i,j) \in Q} w_{(i,j)} = \sum_{i \in V} w_{(i, \pi_Q(i))} = n^2 M + \sum_{i \in V} d(i, \pi_Q(i)) \ ,$$

hence the result. □

Corollary 1.4
Let Q be an inclusion-maximal clique in G' with $|Q| < n$. Then for each $\pi \in \operatorname{Aut} G$, there is a node $(i,j) \in Q$ with $\pi(i) \neq j$.

1.3.5 The Orbit Graphs

Let $G = (V, c)$ be a graph and $d \in \mathbb{N}$. Let G_d be the simple directed graph (V, E_d), such that $(i, j) \in E_d$ if and only if $c(i, j) = d$. For two nodes $i, j \in V$, we call $G_{c(i,j)}$ the *orbit graph* of (i, j). This is motivated by

Lemma 1.5
For $\pi \in \operatorname{Aut} G$, each orbit of π is a directed cycle in exactly one of the graphs G_d.

Proof: Let $i \in V$ and $d = c(i, \pi(i))$. We have $c(i, \pi(i)) = c(\pi^e(i), \pi^{e+1}(i))$ for all $e \in \mathbb{Z}$, hence $\operatorname{orb}_\pi(i)$ is a directed cycle in G_d. Since all orbit graphs are pairwise edge-disjoint, the orbit graph containing $\operatorname{orb}_\pi(i)$ is unique. □

1.4 Symmetries of Graphs

In this section, we define a class of automorphisms that plays an important role in automatic graph drawing: we consider symmetries of a graph, i.e., automorphisms that can be displayed in a two-dimensional drawing. A more precise definition is given in Sect. 1.4.1. The problem of symmetry detection is considered in Sect. 1.4.2, whereas the problem of drawing a given symmetry is discussed shortly in Sect. 1.4.3.

1.4.1 Symmetries

Let $\mathbb{R}^2$ be the Euclidean plane. An *isometry* of $\mathbb{R}^2$ is a bijective map $\varphi\colon \mathbb{R}^2 \to \mathbb{R}^2$ preserving Euclidean distances; it is necessarily either a *rotation*, a *reflection*, or a *translation* of the plane, or a composition of them; see Martin [59]. A rotation is given by a single fixed point, its *center*, and a degree by which all points rotate around the center. A reflection is given by a line of fixed points, its *axis*; all points are reflected at this line. A translation is given by a vector $v \in \mathbb{R}^2$, all points are shifted by this vector.

A *(straight line) drawing* of a graph $G = (V, c)$ in the Euclidean plane is given by an injective map $D\colon V \to \mathbb{R}^2$ assigning nodes to positions on the plane. A *symmetry* of a drawing D is an isometry φ of $\mathbb{R}^2$ such that $\varphi(D(V)) = D(V)$ and

$$c\left(D^{-1}(p_1), D^{-1}(p_2)\right) = c\left(D^{-1}(\varphi(p_1)), D^{-1}(\varphi(p_2))\right) \text{ for all } p_1, p_2 \in D(V)\ ,$$

i.e., the set of node positions is fixed under φ and every edge is mapped to an edge of the same color. By the characterization of planar isometries mentioned above, each symmetry of D must be a combination of reflections and rotations, since a translation cannot map the finite set $D(V)$ to itself. In two dimensions, any such combination is a reflection or a rotation again. Hence every symmetry of D is either a reflection or a rotation. The set $\operatorname{Sym} D$ of all symmetries of D forms a group.

If D is any drawing of G, we have a group homomorphism $R_D\colon \operatorname{Sym} D \to \operatorname{Aut} G$, given by $\varphi \mapsto D^{-1}\varphi D$; recall that $\varphi D(V) \subseteq D(V)$. If $R_D(\varphi) = \pi$, we say that φ *induces* π and that D *displays* π. The image of R_D, i.e., the set of automorphisms displayed by D, is a subgroup of $\operatorname{Aut} G$ and hence a permutation group. In general, R_D is not surjective. There are even automorphisms of G that cannot be displayed by any drawing of G. An example is given in Fig. 1.1. An automorphism π of G that can be displayed by some drawing of G is called a *geometric automorphism* or a *symmetry* of the graph G. Automorphisms are sometimes called *combinatorial symmetries* to distinguish them from geometric symmetries.

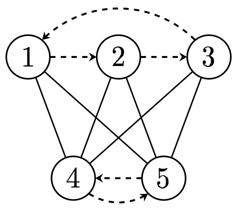


Figure 1.1: The automorphism given by the dashed arrows cannot be displayed in the plane, because it has non-trivial orbits of different lengths. Thus it is neither a rotation nor a reflection and hence no symmetry by Lemma 1.6 below

The set $\operatorname{Sym} G$ of symmetries of G does not even form a group under composition; see Fig. 1.2 for a counterexample. The same is true for the set $\operatorname{Rot}_k G$ of k-rotations of G and the set $\operatorname{Ref} G$ of reflections of G—the composition of two reflections may have any order between one and n.

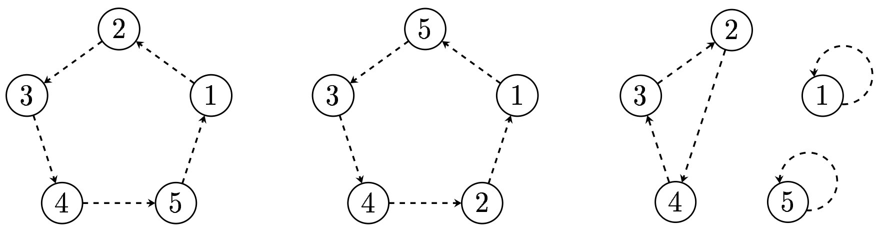


Figure 1.2: The symmetries of E_5 do not form a group: the composition of the 5-rotations displayed in the first two drawings is neither a rotation nor a reflection and hence no symmetry by Lemma 1.6 below

The following results provide abstract criteria for deciding whether some given automorphisms of G can be displayed by a symmetric drawing of G:

Lemma 1.6 (Eades and Lin [29])
An automorphism is a symmetry if and only if it is a rotation or a reflection.

Lemma 1.7 (Eades and Lin [29])
A set of automorphisms can be displayed by a single drawing of G if and only if it generates a subgroup of $\operatorname{Aut} G$ that is generated by a single symmetry or by a rotation π_1 and a reflection π_2 satisfying $\pi_2\pi_1 = \pi_1^{-1}\pi_2$.

In general, a set of two or more symmetries of the same graph G cannot be displayed by a single drawing; see Fig. 1.3 for an example.

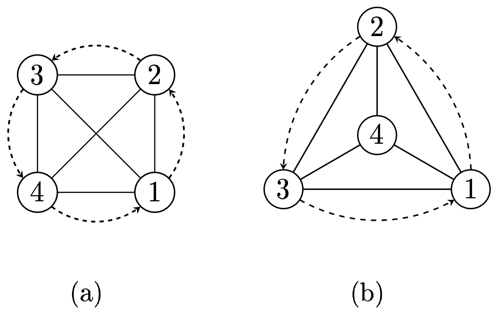


Figure 1.3: Two symmetries of K_4 that cannot be displayed simultaneously

1.4.2 Detecting Symmetries

As pointed out in the introduction, displaying symmetries is an important aim in automatic graph drawing. In general, there are two different ways to tackle this problem. Some algorithms have an implicit tendency to display symmetries but fail to find them in general. A well-known example is the spring embedder; see Eades and Lin [29] for a discussion of this method with respect to symmetry. Other heuristics are based on algebraic graph theory; see e.g. de Fraysseix [26].

The second way for creating symmetric drawings is to search for symmetries abstractly before computing a corresponding layout. In this approach, the problem of detecting a non-trivial symmetry in an abstract graph has to be solved. Manning [54] showed that the following problems concerning rotations are NP-complete:

(Rot)	Given a graph G. Decide whether there exists a non-trivial rotation of G.

(Rot$_0$)	Given a graph G. Decide whether there exists a rotation of G without fixed nodes.

(Rot$_1$)	Given a graph G. Decide whether there exists a rotation of G with one fixed node.

For reflections, we cannot use Manning's result, since his definition of an axial symmetry is not equivalent to our definition of a non-trivial reflection. However, finding a reflection of G with the minimum number of fixed nodes is NP-hard. This follows from the NP-completeness of problem (2-Aut$_0$), see Sect. 1.3.2, which is just another formulation of

(Ref$_0$)	Given a graph G. Decide whether there exists a reflection of G without fixed nodes.

If the number of fixed nodes is restricted only by requiring non-triviality, the problem is at most isomorphism-complete:

(Ref)	Given a graph G. Decide whether there exists a non-trivial reflection of G.

This follows from the isomorphism-completeness of the problem (#Aut) defined in Sect. 1.3.2, since a non-trivial reflection is the same as an automorphism of order two: by group-theoretic arguments, the group $\operatorname{Aut} G$ contains an element of order two if and only if the number of automorphisms of G is even. Hence (Ref) can be reduced to the problem (#Aut). We do not know whether the converse is true as well, i.e., whether the problem (Ref) is isomorphism-complete.

Finally, consider the problem

(RotA)	Given a graph G on n nodes. Decide whether there exists a rotation of G of order n.

A graph G for which (RotA) is answered positively is called *circulant*. Equivalently, a circulant graph is a Cayley graph over $\mathbb{Z}_n$. Hence (RotA) is the problem of recognizing circulant graphs. The complexity of this problem is unknown. However, if n is prime, the problem (RotA) is solvable in polynomial time by a result of Muzychuk and Tinhofer [63, 64]. We conclude that (Rot$_0$) is polynomial as well if n is prime. In fact, the only candidate rotation order is n in this case.

In summary, detecting symmetries in general graphs is NP-hard. Nevertheless, for planar graphs these symmetry problems can be solved in linear time if only planar symmetries are allowed, i.e., symmetries that can be displayed by planar drawings.

Manning [53] showed this for outerplanar and embedded graphs. The algorithm for embedded graphs was generalized to triconnected planar graphs by Hong et al. [42]. Using SPQR-trees and block-cut-trees, the result was extended to biconnected and one-connected planar graphs and finally to general planar graphs by Hong and Eades [39, 40, 41]. In Chapter 5, we present a new approach for detecting symmetries based on integer programming. This algorithm is exact and admits general graphs. Because of the NP-hardness of the problem one cannot expect polynomial runtime. However, the practical runtimes are reasonable for graphs on up to 50 nodes.

Recently, another method for symmetry detection in general graphs was presented by Abelson et al. [2]. On one hand, the runtime results of this approach are significantly better than ours. On the other hand, our approach is more flexible: we can use arbitrary linear objective functions, see Chapters 2 to 5, and we can generalize our approach to find fuzzy symmetries, see Chapter 6.

1.4.3 Drawing Symmetries

Once we have computed a symmetry π of G—abstractly given as a permutation of nodes—we want to compute a nice drawing of G displaying π. On one hand, the restriction to drawings displaying π is strong. Most existing algorithms in automatic graph drawing—see Kaufmann and Wagner [47] or Di Battista et al. [28] for an overview—cannot deal with this problem. The only well-known algorithm that can be adjusted in a trivial way is the spring embedder: theoretically, it suffices to start with any drawing displaying π, since every iteration works symmetrically and hence preserves π. However, fast implementations usually do not work exactly and hence not always symmetrically. This can easily be repaired after each iteration. Algorithms computing orthogonal or nearly orthogonal drawings are inapt to draw symmetries in general, since a symmetry of order k with $k \neq 2, 4$ cannot be displayed in this style. The same is true for Sugiyama style layouts.

On the other hand, the restriction to drawings displaying π still allows to take other aesthetics into account. The most important aesthetic criterion according to Purchase [67] is crossing minimization. Buchheim and Hong [17] consider the problem of finding a drawing of a fixed symmetry π of a simple graph with a minimal number of edge crossings. They show that this problem is NP-hard even under strong restriction. Furthermore, they give an $O(m \log m)$ algorithm that finds a crossing minimal drawing of a symmetry with an orbit graph that is a path, where m is the number of edges of G and the orbit graph of π arises from G by merging nodes belonging to the same orbit under π and deleting multiple edges and loops. Another algorithm for drawing symmetries was presented by Carr and Kocay [22]; this algorithm does not consider any aesthetic criterion explicitly. It searches for a long cycle in G that can be drawn as a cycle in a symmetric drawing of π. However, for the nodes not belonging to the cycle, no algorithm is given.

1.5 Branch and Cut

Concluding the preliminaries, we shortly describe the branch & cut-technique and introduce the corresponding terminology. For more details and other applications, see e.g. Jünger and Naddef [44] or Jünger and Thienel [45].

The first task in a branch & cut-approach is to model the problem being tackled as an *integer linear program* (ILP). An ILP is a linear program (LP) with the additional constraint that all variables be integer. Solving ILPs is NP-hard in general. In fact, for many well-known NP-hard problems an ILP-model is found easily. On the other hand, LPs without integrality constraints can be solved in polynomial time by Karmakar [46] and very fast in practice, e.g., by CPLEX [25]. This leads to the idea of using LP-techniques for solving ILPs. For this, we consider the *LP-relaxation* of the ILP, obtained by dropping all integrality constraints. If we are lucky, the optimal solution of this relaxation is integer and thus also an optimal solution of the ILP.

In general, however, the optimal solution of the LP-relaxation will be fractional. For ease of exposition, we only consider minimization problems in the following. Then a fractional solution yields a lower bound on the optimal objective value of the ILP, but does not yield a feasible solution. On the other hand, any feasible solution of the ILP corresponds to an upper bound; the problem is solved to optimality as soon as upper and lower bound coincide.

If the optimal solution of the LP-relaxation is fractional, we have to start the *cutting phase*: a *cutting plane* is a linear inequality that is valid for all feasible solutions of the ILP but not for the currently optimal fractional solution. In the cutting phase, we try to find such inequalities by so-called *separation algorithms* in order to add them to the LP-relaxation. The resulting LP has a new optimal solution; if this solution is fractional again, we start another separation step. By Grötschel et al. [35], the separation problem for a given ILP is polynomially equivalent to the problem of minimizing a general linear objective function over this ILP. In practice, only special classes of cutting planes are considered and separated.

The theoretical background for cutting planes and separation is as follows: we consider the polytope P defined as the convex hull of all integer solutions of the original ILP; we assume that this polytope is bounded. If we had a complete polynomial description of P in terms of linear constraints, the problem could be solved in polynomial time. Indeed, we could ignore the integrality constraints then.

In general, however, no such description exists. In the cutting phase, the polytope P is enclosed more and more by new inequalities. By adding only constraints violated by the current fractional solution, we hope that a small number of new constraints suffices to get an integer solution.

Usually, the best cutting planes are inequalities inducing *facets* of the polytope P, i.e., maximal faces of P with respect to inclusion. For this reason, branch & cut-algorithms afford a good deal of problem-specific polyhedral investigation. Profound knowledge of the polyhedral structure of P allows to determine useful classes of cutting planes. After such a class has been found, the corresponding separation problem has to be solved, at least heuristically. Notice that the cutting procedure also allows to solve ILPs with an exponential number of constraints in the original formulation: for non-feasible but integer solutions of the LP-relaxation, we can separate cutting planes as well.

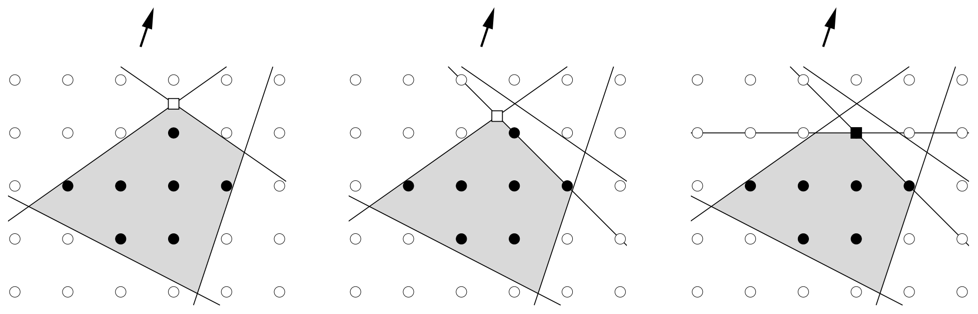


Figure 1.4: An example for the cutting phase: the linear objective function is represented by the vector on top, the polytope defined by the LP-relaxation is given by the shaded area, and the feasible solutions of the ILP are drawn as filled dots. At first, the optimal LP-solution, marked by a square, is fractional. After adding a facet-inducing inequality, the new LP-solution is still fractional. Adding another linear inequality finally yields an integer optimal solution

In summary, the cutting phase consists of solving LPs and adding new constraints alternately; see Fig. 1.4 for an illustration. However, we may end up with a fractional solution for which we do not find any cutting plane. In this case, the *branching phase* is invoked. After choosing a *branching variable* x with LP-value $\overline{x}$, the problem is divided into two subproblems: in the first, we add the constraint $x \leq \lfloor \overline{x} \rfloor$; in the second, we add $x \geq \lfloor \overline{x} \rfloor + 1$. The optimal solution of the original problem is the better one of the optimal solutions of the two subproblems. Repeating the branching process yields a *branching tree* of subproblems. The *enumeration strategy* determines in which order the tree is traversed as long as no optimal solution for the original ILP is found.

If all variables in the ILP are binary, branching means *setting* a variable to zero in the first subproblem and to one in the second subproblem. As soon as one or more variables are set in some subproblem, it may be possible to set further variables by logical implications. Sometimes it is possible to *fix* variables for the original ILP, i.e., for every subproblem in the branching tree, for example by reduced costs. This may also allow fixing other variables by logical implications.

An important feature of the branch & cut-method is that usually only a small part of the branching tree has to be traversed, because some subtrees can be *pruned*. This is the case if the local lower bound in a subproblem is greater than the global upper bound, i.e., if we already know some feasible solution that is better than all solutions in this subproblem. We can skip the corresponding subtree then.

To find feasible solutions of the global ILP, we do not have to wait for integer LP-solutions in general. Often a fractional LP-solution helps to find good feasible solutions heuristically. This fractional solution may be close to an optimal feasible solution of the ILP. Algorithms trying to derive feasible solutions of the ILP from fractional LP-solutions are called *primal heuristics*.

Chapter 2

Detecting Automorphisms

Our first target is to investigate the problem of detecting not necessarily geometric automorphisms. Note that there are practically fast algorithms finding not only a single non-trivial automorphism but even a set of generators of $\operatorname{Aut} G$ for a given graph G. The new approach presented in this chapter is not meant to be an algorithm for finding any automorphism, but to find automorphisms that meet additional requirements: first, by choosing an adequate linear objective function. For example, we can minimize the number of fixed nodes and hence solve the problem (AUT_0). More generally, for any $R \subseteq V^2$, we can minimize the number of pairs $(i, j) \in R$ such that i is mapped to j, thus solving the problems (AUT1R) and (AUT-R). Most generally, we can assign a weight to each pair of nodes (i, j) determining how much we would like to have node i mapped to node j. We consider these problems in a common integer programming approach designed to find automorphisms minimizing an arbitrary linear objective function. The second way to restrict the set of desired automorphisms is by adding new linear constraints to the ILP; we do this in the following chapters in order to detect geometric automorphisms.

For designing the desired branch & cut-algorithm, the first step is to set up an integer linear program describing the automorphisms of a given graph; see Sect. 2.1. Different objective functions are proposed in Sect. 2.2. An important theoretical task is to find out as much as possible about the structure of the corresponding automorphism polytope. Basic properties of this polytope are examined in Sect. 2.3, while cutting planes are discussed in Sect. 2.4. In Sect. 2.5, we deal with the separation problem for automorphism polytopes. In Sect. 2.6, we present primal heuristics for finding automorphisms. We discuss the branching and enumeration strategy in Sect. 2.7, explain a fixing and setting rule in Sect. 2.8, and examine the practical performance of the algorithm in Sect. 2.9.

2.1 The Integer Linear Program

Let $G = (V, c)$ be a graph as defined in Sect. 1.1. Assume $V = \{1, \ldots, n\}$ and let π be a permutation of V. Then we define a real $n \times n$-matrix $M(\pi)$ by

$$M(\pi)_{ij} = \begin{cases} 1 & \text{if } \pi(i) = j \\ 0 & \text{otherwise,} \end{cases}$$

yielding a monomorphism M of the group S_n of permutations of V into the general linear group $\mathrm{GL}_n(\mathbb{R})$. In particular, we get an isomorphism between S_n and its image $M(S_n)$. The matrices in $M(S_n)$ are called *permutation matrices* and can be characterized as the set of $n \times n$-matrices $X = (x_{ij})$ with $x_{ij} \in \{0, 1\}$ and

$$\sum_{j \in V} x_{ij} = 1 \text{ for all } i \in V \quad \text{and} \quad \sum_{i \in V} x_{ij} = 1 \text{ for all } j \in V\ . \tag{2.1}$$

For the desired ILP modeling automorphisms, we use a binary *mapping variable* x_{ij} for each pair $(i, j) \in V^2$ and add the constraints (2.1). A value of one for the mapping variable x_{ij} is interpreted as mapping node i to node j. The equations on the left hand side of (2.1) make sure that each node i is mapped to exactly one node j, so that the variables induce a function $V \to V$. By the right hand side, this function is bijective. In summary, we have modeled the permutations of V up to now.

Next we have to translate the condition of $\pi \in S_n$ being an automorphism of G into a condition on the corresponding matrix $M(\pi)$. For this, consider the adjacency matrix A_G of G. We use the following Lemma:

Lemma 2.1
The permutation π of V is an automorphism of G if and only if the matrix $M(\pi)$ commutes with A_G, i.e., if $M(\pi)A_G = A_G M(\pi)$.

Proof: Let $M = M(\pi)$. We have $(A_G)_{ij} = c(i, j)$. Now $MA_G = A_G M$ if and only if $MA_G M^{-1} = A_G$ if and only if $c(\pi(i), \pi(j)) = c(i, j)$ for all $i, j \in V$. □

By Lemma 2.1, the automorphisms of G are singled out by adding the n^2 linear constraints $A_G X = XA_G$, where $X = (x_{ij})$. Hence they correspond to the solutions of the following ILP via M:

$$\begin{array}{rcll} x_{ij} & \in & \{0,1\} & \text{for all } i,j \in V \\ \sum_{j\in V} x_{ij} & = & 1 & \text{for all } i \in V \\ \sum_{i\in V} x_{ij} & = & 1 & \text{for all } j \in V \\ A_G X & = & XA_G\ . & \end{array} \tag{2.2}$$

The number of variables in (2.2) is n^2. If the nodes of the graph have different colors, this number can often be reduced sharply: suppose that for two nodes $i, j \in V$ we have $c(i) \neq c(j)$. No automorphism can map i to j then, hence x_{ij} is zero for any solution of (2.2). Consequently, we can omit this variable from the beginning. Observe that a finer coloring c of the graph allows to leave out more mapping variables. Hence fine labelings or even the automorphism partitioning can improve performance significantly; see Sect. 1.3.3.

2.2 Objective Functions

In Sect. 2.1, we did not introduce any objective function for the automorphism ILP. In general, any linear function can be used. In the following, we propose some objective functions for solving problems described in Sect. 1.3.2.

Assume that our aim is to minimize the objective function. In the general case, we have a coefficient $w_{ij} \in \mathbb{R}$ for each mapping variable x_{ij} and hence for each pair of nodes (i, j). The larger w_{ij} is, the less preferred is mapping i to j. More precisely, we get an automorphism $\pi \in \operatorname{Aut} G$ minimizing

$$\sum_{\pi(i)=j} w_{ij} = \sum_{i \in V} w_{i\pi(i)} \; .$$

If all coefficients w_{ij} are binary, we thus find an automorphism π such that the set

$$\{i \in V \mid w_{i\pi(i)} = 1\}$$

has minimal cardinality. In other words, we can solve the NP-complete problem (Aut-R) by defining

$$w_{ij} = \begin{cases} 1 & \text{if } (i,j) \in R \\ 0 & \text{otherwise.} \end{cases}$$

As a consequence, the general optimization problem for the automorphism ILP is NP-hard. The problem (Aut1R) is a special case of (Aut-R). Another special case of (Aut1R) is (Aut_0): for this problem, we can set

$$w_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

By this, we find an automorphism with the minimum number of fixed nodes. In particular, the group $\operatorname{Aut} G$ is trivial if and only if the resulting automorphism is id_V. Thus we can solve (Aut) as well.

2.3 The Automorphism Polytope

We next examine the polytope corresponding to the automorphism ILP presented in Sect. 2.1. The *assignment polytope* $P(S_n) \subseteq \mathbb{R}^{n\times n}$ is defined as the convex hull of all permutation matrices, i.e., as the convex hull of $M(S_n)$, where M is defined as in Sect. 2.1. This polytope is well studied. The matrices in $P(S_n)$ are called *doubly stochastic matrices*. Birkhoff [11] showed that $P(S_n)$ is just the set of matrices $X = (x_{ij})$ satisfying the equations (2.1) and

$$x_{ij} \geq 0 \text{ for all } i,j \in V \ . \tag{2.3}$$

Hence we have a small and simple linear description of the polytope $P(S_n)$. The dimension of $P(S_n)$ is $(n-1)^2$ and the number of its facets is n^2. These and other properties of $P(S_n)$ are demonstrated by Balinski and Russakoff [6] and by Brualdi and Gibson [14, 15, 16].

Now we define the *automorphism polytope* $P(\operatorname{Aut} G) \subseteq \mathbb{R}^{n\times n}$ of a graph $G = (V, c)$ as the subpolytope of $P(S_n)$ given as the convex hull of $M(\operatorname{Aut} G)$. In particular, we have $P(\operatorname{Aut} K_n) = P(S_n)$. In general, the automorphism polytope is much less examined and understood than the assignment polytope. The properties of $P(\operatorname{Aut} G)$ strongly depend on the structure of the graph, not only on the number of nodes. Even computing the dimension is at least as hard as the problem (AUT) defined in Sect. 1.3.2. It ranges from $(n-1)^2$ for a complete or edgeless graph to zero for a graph without non-trivial automorphisms. Thus, proving results about facets of $P(\operatorname{Aut} G)$ in the classical way is hardly possible. Instead, we will describe this polytope in a rather unusual way in Sect. 2.4.

2.4 Cutting Planes

After setting up the ILP, the most important task for designing a branch & cut-algorithm is to find good cutting planes. More precisely, we have to find classes of inequalities that are valid for each feasible solution of the ILP and that are as tight as possible. Furthermore, we need separation algorithms to find out, at least heuristically, whether some of these inequalities are violated by a fractional solution of the LP-relaxation; these are examined in Sect. 2.5.

In Sect. 2.4.1, we present a class of constraints based on automorphism partitionings of G. This class induces a complete description of the automorphism polytope. Some subclasses are examined in Sect. 2.4.2.

2.4.1 Homomorphism Constraints

As noted in Sect. 2.3, the structure of the automorphism polytope of G strongly depends on the structure of G, making if difficult to prove results about facets in the usual way. Nevertheless, using automorphism partitionings, we can describe the automorphism polytope completely in the following way: fix $t \in \mathbb{N}$ and consider the t-automorphism partitioning c_t of G defined in Sect. 1.3.3. Let I be a multiset of node-pairs of G, i.e., a finite set of node-pairs that may contain multiple elements. Suppose that

$$c_t(i_1, \ldots, i_t) \neq c_t(j_1, \ldots, j_t) \text{ for all subsets } \{(i_1, j_1), \ldots, (i_t, j_t)\} \text{ of } I \;, \tag{2.4}$$

where the subsets of a multiset are defined in the obvious way. Then, by definition of c_t, the *homomorphism constraint* given by

$$H_{I,t}\colon \sum_{(i,j)\in I} x_{ij} \leq t-1 \tag{2.5}$$

is a valid inequality for the automorphism ILP (2.2). Observe that $H_{I,t}$ is still valid if we replace c_t by an arbitrary t-labeling of G in (2.4), but we do not necessarily get all homomorphism constraints in this case.

For $t = 1$, the constraint $H_{I,t}$ is equivalent to $x_{ij} \leq 0$ for all $(i,j) \in I$. Hence all inequalities $H_{I,1}$ are satisfied as soon as $x_{ij} = 0$ for all $i, j \in V$ with $c_1(i) \neq c_1(j)$. These equations have been used in Sect. 2.1 to reduce the number of variables.

Theorem 2.2
In the affine subspace of $n \times n$-matrices (x_{ij}) given by $\sum_{j\in V} x_{ij} = 1$ for all $i \in V$, each rational inequality valid for $P(\operatorname{Aut} G)$ is induced by a homomorphism constraint.

Proof: Let

$$H\colon \sum_{(i,j)\in V^2} a_{ij} x_{ij} \leq t-1$$

be any rational inequality valid for $P(\operatorname{Aut} G)$. We may assume $t \in \mathbb{Z}$ and $a_{ij} \in \mathbb{Z}$ for $i, j \in V$. For all $(i,j) \in V^2$ with $a_{ij} < 0$, we use $\sum_{j'\in V} x_{ij'} = 1$ to replace x_{ij} by

$$1 - \sum_{j'\in V\setminus\{j\}} x_{ij'} \;,$$

increasing the coefficient of each $x_{ij'}$ by $-a_{ij} > 0$. After these replacements, all coefficients on the left hand side are non-negative, hence we may assume $a_{ij} \geq 0$ for all $i, j \in V$. Since $M(\mathrm{id}_V) \in P(\operatorname{Aut} G)$, we derive $\sum_{i\in V} a_{ii} \leq t-1$ and hence $t \geq 1$. Let I be the multiset of node-pairs containing the pair (i,j) exactly a_{ij} times, for all $i, j \in V$. Since H is valid for $P(\operatorname{Aut} G)$, condition (2.4) holds for I and t. Obviously, we have $H = H_{I,t}$. □

Corollary 2.3
The polytope $P(\operatorname{Aut} G)$ is completely described by the constraints (2.1) and (2.5).

2.4.2 Special Homomorphism Constraints

In practice, homomorphism constraints perform well even if t is restricted to two. To explain this, we give two examples of special constraints contained in the class of inequalities $H_{I,2}$.

First observe that the constraints $A_G X = X A_G$ in (2.2) are constraints of type $H_{I,2}$ if G is a simple graph. Indeed, the inequality

$$\sum_{k=1}^{n} a_{ik} x_{kj} \leq \sum_{k=1}^{n} x_{ik} a_{kj}$$

is equivalent to

$$\sum_{k=1}^{n} a_{ik} x_{kj} + \sum_{k=1}^{n} (1 - a_{kj}) x_{ik} \leq 1$$

using the constraints (2.1). In the latter inequality, all coefficients are binary for simple graphs.

Next, let $i \in V$ and $d \in \mathbb{N}$. In the orbit graph G_d defined in Sect. 1.3.5, the in-degree of i has to equal its out-degree by Lemma 1.5, i.e., we have the valid equation

$$\sum_{c(j,i)=d} x_{ji} = \sum_{c(i,k)=d} x_{ik} \,. \tag{2.6}$$

Using (2.1), the two inequalities given by (2.6) are equivalent to

$$\sum_{c(j,i)=d} x_{ji} + \sum_{c(i,k)\neq d} x_{ik} \leq 1 \quad \text{and} \quad \sum_{c(j,i)\neq d} x_{ji} + \sum_{c(i,k)=d} x_{ik} \leq 1 \,.$$

These inequalities are easily seen to be homomorphism constraints.

2.5 Separation

Since the linear optimization problem over the automorphism polytope $P(\operatorname{Aut} G)$ is NP-hard in general, the same is true for the corresponding general separation problem by Grötschel et al. [35]. As all constraints are induced by homomorphism constraints by Theorem 2.2, the separation problem for homomorphism constraints is also NP-hard. In fact, the size of the t-automorphism partitioning of G is exponential in t, hence we cannot even compute the automorphism partitionings in polynomial time for unbounded t.

However, in practice, we noticed that for most test instances the restriction to $t = 1, 2$ sufficed to prevent branching when searching for automorphisms; see the runtime results in Sect. 2.9. Hence in our branch & cut-approach, we only use homomorphism constraints of type $H_{I,1}$ and $H_{I,2}$. As explained in Sect. 2.4, the constraint $H_{I,1}$ is equivalent to $x_{ij} = 0$ for all $(i, j) \in I$, so all inequalities $H_{I,1}$ are satisfied as soon as $x_{ij} = 0$ for all $i, j \in V$ with $c_1(i) \neq c_1(j)$. For $t = 2$, the separation is performed heuristically: consider the coloring graph G' defined in Sect. 1.3.4. By (2.4), the constraint $H_{I,2}$ is valid for $P(\operatorname{Aut} G)$ if and only if I is an independent set in G'. We do not know whether finding maximum weight independent sets in G' is possible in polynomial time. However, we can use any heuristic for the weighted independent set problem to separate the constraints $H_{I,2}$ heuristically. The straightforward greedy strategies we use in our implementation yield good results quickly in general.

2.6 Primal Heuristics

Finding automorphisms heuristically is difficult, since a local variation of the graph usually changes the set of its automorphisms completely, so that straightforward greedy heuristics do not work. Nevertheless, Lemma 1.2 of Sect. 1.3.4 helps to develop a primal heuristic for finding automorphisms of G by providing a link between automorphisms of G and maximal cardinality cliques in the coloring graph G'. This Lemma suggests to search for automorphisms of G by searching for maximal cardinality cliques in G'. In a branch & cut-approach, any primal heuristic should be guided by the last LP-solution $\overline{x}_{ij}$. This solution can be taken into account easily: we define weights for the nodes of G' by $w_{(i,j)} = n + \overline{x}_{ij}$ and search for maximum weight cliques.

Doing this heuristically, we may end up with an inclusion-maximal clique Q in G' that does not contain n nodes. This occurs in particular if we use greedy heuristics. In this case, we can use Corollary 1.4 to derive the valid inequality

$$\sum_{(i,j) \in Q} x_{ij} \leq |Q| - 1 \; .$$

If this inequality is violated, we add it as a cutting plane.

We also use the following very simple primal heuristic: we start with an undefined function π and traverse the mapping variables by descending LP-value $\overline{x}_{ij}$. For each variable x_{ij} visited, we check whether both $\pi(i)$ and $\pi^{-1}(j)$ are undefined. If so, we set $\pi(i) = j$. By this, we finally get a permutation π of V. If we are lucky, this permutation is an automorphism of G. In practice, this method often yields good primal solutions while requiring very little runtime.

2.7 Branching and Enumeration

Our branching and enumeration strategy is very simple: we always branch at the fractional variable with greatest value in the last LP-solution. The subproblem tackled next is the one containing the largest number of variables fixed or set to one. Notice that setting a variable to one allows setting $2(n-1)$ other variables to zero, whereas setting a variable to zero does not determine other variables in general. So our strategy prefers to set as many variables as possible in the hope of finding feasible solutions quickly.

We also make use of the isomorphism pruning technique devised by Margot [57]. This method is able to decrease the number of subproblems to be solved in a branch & cut-algorithm in case that the ILP model has a non-trivial symmetry group. The symmetry group consists of all permutations of the set of variables that do not change the ILP including the objective function. In general, the larger the symmetry group of an ILP is, the more subproblems in the corresponding branching tree are isomorphic. Since isomorphic subproblems have equal optimal solutions, we have to consider only one of the isomorphic subproblems and can prune the others.

Coming back to our application, we assume that we use an objective function that is fixed under the symmetry group of (2.2). For example, this is true for the number of fixed nodes. Then it is easy to see that

Lemma 2.4
The symmetry group of (2.2) is isomorphic to $\operatorname{Aut} G$.

Here, an automorphism π of G acts on the set of variables by mapping x_{ij} to $x_{\pi(i)\pi(j)}$. By Lemma 2.4, applying isomorphism pruning is the more profitable for a graph G the more automorphisms G admits. However, the drawback of this technique is that a lot of group operations have to be performed in order to decide whether a subproblem can be pruned. These are hard to implement and do not run in polynomial time in general. For this reason, we use a weaker pruning criterion than the one given in [57]. Our criterion can be checked quickly if the 2-automorphism partitioning c_2 is given. To explain it, assume that we use a ranked branching rule and a rank vector R as in [57]. Then we have

Theorem 2.5
Let W be the set of variables currently set to one by branching decisions. Assume that there exist $i, j \in V$ with $R[x_{ij}] \leq n^2$ and $c_2(i,j) = c_2(i',j')$ for some $x_{i'j'} \in W$. Choose such i and j with $R[x_{ij}]$ minimal. Then the current subproblem can be pruned if $x_{ij} \notin W$.

Proof: By definition of c_2, there is an automorphism of G mapping i to i' and j to j'. By Lemma 2.4, this automorphism induces an automorphism π of (2.2) mapping x_{ij}

to $x_{i'j'}$. Hence x_{ij} is contained in $\pi^{-1}(W) \setminus W$. By the minimality of $R[x_{ij}]$, it follows that $R(\pi^{-1}(W))$ is lexicographically strictly smaller than $R(W)$. Hence the current subproblem can be pruned by [57]. $\square$

Observe that the criterion of Theorem 2.5 can be verified easily in a runtime linear in the number of variables, if the time needed for computing c_2 is ignored. In fact, we only have to compute c_2 once for each instance and we compute it anyway for separating homomorphism constraints; see Sect. 2.5.

2.8 Fixing and Setting Variables

Fixing or setting variables in a branch & cut-algorithm sometimes allows to fix or set other variables by logical implications. Whenever a variable x_{ij} is fixed (set) to one in our application, we know that some other variable $x_{i'j'}$ may be one only if $c(i,i') = c(j,j')$ and $c(i',i) = c(j',j)$. Otherwise, we can fix (set) $x_{i'j'}$ to zero. Clearly, this remains true—and is more efficient in general—if we replace c by the possibly finer 2-automorphism partitioning c_2.

2.9 Experimental Results

In the following, we present runtime results for our branch & cut-implementation of automorphism detection for general graphs. In the experiments, we searched for an automorphism with a minimal number of fixed nodes; this problem is NP-hard as a generalization of the problem ($\textsc{Aut}_0$) defined in Sect. 1.3.2. When using other objective functions, we obtained similar results in general.

For all our evaluations, we used an AMD 1400 MHz Athlon processor. Runtime results are always given in CPU-seconds; we imposed a CPU-time limit of one hour. Our implementation is based on ABACUS [1, 45] in combination with CPLEX [25]. The general branch & cut-parameters were set as follows: we neither restricted the size of the branching tree nor the number of cutting phase iterations per subproblem. The number of cutting planes added in a single iteration was bounded by 100.

2.9.1 The Algorithm

In a preprocessing step, we apply `nauty` [60] to replace the original coloring c of the input graph by its 2-automorphism partitioning c_2. The initial ILP in the implemented branch & cut-algorithm models all node permutations, i.e., it consists of mapping variables and the permutation constraints (2.1); variables are deleted as described in Sect. 2.1.

Notice that we do not use the constraints $A_G X = X A_G$ of (2.2). Instead, we make use of the homomorphism constraints (2.5) for $t = 2$ in the cutting phases; see Sect. 2.4. No other class of cutting planes is included in our implementation. During separation, we add as many violated homomorphism constraints as we can find by a simple greedy independent set heuristic; see Sect. 2.5. As mentioned above, the general limit is 100 per cutting phase iteration, but this limit is rarely reached.

If branching is necessary, we use the branching and enumeration strategy described in Sect. 2.7 and fix and set variables by the logical implications explained in Sect. 2.8. Finally, we implemented both a primal heuristic based on a greedy maximal clique heuristic and the simple algorithm given in Sect. 2.6.

2.9.2 Test Sets

For lack of benchmark data sets containing graphs with non-trivial automorphism groups, we created random instances for evaluating our algorithm. This is delicate: testing random graphs is not useful, since automorphisms are rare. Most graphs do not have any non-trivial automorphism at all. In this case, the automorphism partitioning computed by `nauty` is trivial, so there is nothing left to do. But if the user of our algorithm would not expect her graphs to have symmetries or automorphisms with a reasonable probability, she probably would not use it.

Instead of using random graphs, we therefore created all instances in the following way: for a given number n of nodes, we first chose some permutations $\pi_0, \pi_1, \ldots, \pi_r$ of $V = \{1, \ldots, n\}$ according to a rule depending on the test set. Then we computed the partition of V^2 induced by these permutations, i.e., the orbitals of the permutation group generated by $\pi_0, \pi_1, \ldots, \pi_r$. We joined the parts containing (i, j) and (j, i) for all $i, j \in V$ in order to get an undirected graph. Finally, we flipped a coin for each part: either we added an edge between all node-pairs in the part or between none of them.

Observe that we did not explicitly forbid isomorphic pairs of graphs in our test sets. However, the probability of creating such pairs is low except for very small instances. Furthermore, we did not require connectedness.

For automorphism detection, we created two different test sets. Instances of the first set `aut` were created using a single automorphism: given the number n of nodes, we chose a permutation π_0 of V randomly. Using this permutation, we produced a test graph in the way just explained. For each $n = 1, \ldots, 100$, we created 100 instances.

We also created another test set `aut+` containing instances with more automorphisms than `aut`: we used three random permutations with $\lfloor n/2 \rfloor$ or more fixed nodes each to create a test graph as explained above. Complete or edgeless graphs were rejected. Again, we created 100 instances for $n = 1, \dots, 100$. The resulting graphs have large automorphism groups in general: we have $|\mathrm{Aut}\, G| \geq \lfloor n/2 \rfloor!$ for 99.9% and $|\mathrm{Aut}\, G| \geq (n-1)!$ for 6.8% of the graphs in `aut+`; the corresponding numbers for `aut` are 10.6% and 4.6%.

2.9.3 Results

The results of applying our automorphism detection algorithm to the graphs in `aut` are displayed in Table 2.1. Sorted by the number of nodes, we give the average and maximal values for the following data: the runtime, the number of variables used in the ILP, the number of subproblems created during the branch & cut-process (not counting the root problem), and the number of LPs that had to be solved. Where not all instances could be solved to optimality within the time limit of one CPU-hour, we give the percentage of non-solved instances instead of the maximum runtime. Observe that all other results given in the corresponding lines refer only to the instances that could be solved within one hour.

Table 2.1: Results for automorphism detection in `aut`

	runtime		#variables		#subprobs		#LPs	
n	avg	max	avg	max	avg	max	avg	max
1–10	0.00	0.02	24.1	100	0.0	0	1.2	9
11–20	0.01	1.10	135.2	400	0.0	2	1.8	66
21–30	0.03	0.48	343.6	900	0.0	10	2.3	66
31–40	0.06	0.75	646.4	1600	0.0	6	2.7	34
41–50	0.12	1.11	1075.6	2500	0.0	6	3.1	32
51–60	0.23	15.39	1587.4	3600	0.2	122	3.5	159
61–70	0.43	28.85	2229.1	4900	0.0	6	3.8	89
71–80	0.59	28.48	2879.8	6400	0.0	16	3.9	83
81–90	3.15	1299.44	3686.3	7922	0.1	28	5.1	326
91–100	5.35	0.1 %	4586.9	9802	0.2	116	5.3	388

Table 2.1 shows that, on average, runtime is very short for up to 80 nodes and reasonable for 81 to 100 nodes. However, the variance is high: while 999 instances on 91 to 100 nodes needed 5.35 CPU-seconds on average, a single instance could not be solved within an hour of CPU time. This wide range of performance is typical for most of our experimental results.

We also investigated the effect of computing the 2-automorphism partitioning c_2 in a preprocessing step and found that this is crucial for the performance of our algorithm: when switching off 2-automorphism partitioning, i.e., when applying our algorithm to the original graph coloring c instead of c_2, runtimes for `aut` grew significantly. This is not surprising, since the performance of our algorithm depends on fine labelings at many points, in particular for obtaining tight homomorphism constraints. Nevertheless, when using the polynomial time labeling algorithm `qweil` by Bastert [7], results were the same as with `nauty`, even though `qweil` does not always find the 2-automorphism partitioning.

Most algorithms dealing with automorphisms of a graph need the more runtime the more automorphisms the graph admits; see e.g. [2]. Therefore, we also applied our algorithm to the test set `aut+` containing graphs with large automorphism groups. Runtimes for `aut+` are displayed in Table 2.2.

Table 2.2: Results for automorphism detection in `aut+`

	runtime		#variables		#subprobs		#LPs	
n	avg	max	avg	max	avg	max	avg	max
1–10	0.00	0.02	22.7	82	0.0	0	1.0	1
11–20	0.01	0.03	171.2	362	0.0	0	1.0	1
21–30	0.02	0.04	473.7	842	0.0	0	1.0	1
31–40	0.04	0.09	938.4	1522	0.0	0	1.0	2
41–50	0.09	0.16	1540.9	2214	0.0	0	1.0	2
51–60	0.18	0.30	2301.0	3142	0.0	0	1.0	1
61–70	0.30	0.74	3209.6	4494	0.0	2	1.0	4
71–80	0.47	0.79	4282.5	5782	0.0	0	1.0	2
81–90	0.66	0.87	5502.9	6900	0.0	0	1.0	1
91–100	0.95	1.26	6879.2	9033	0.0	0	1.0	1

In spite of their large automorphism groups, the graphs in `aut+` require less time than those in `aut`; especially the maximum runtimes are much shorter. Here we see the result of two counteracting effects: on one hand, the average number of variables needed for graphs with a lot of automorphisms is larger, as a smaller automorphism group implies a finer 2-automorphism partitioning which in turn allows to omit more variables. On the other hand, the presence of more automorphisms implies that solutions of the ILP are found more easily. The second effect seems to prevail.

In summary, our branch & cut-algorithm for detecting optimal automorphisms in general graphs runs very fast in practice for most instances on up to 100 nodes, in particular for graphs with a large automorphism group.

Chapter 3

Detecting Rotations

In the previous chapter, we explained how to detect general graph automorphisms by a branch & cut-approach. In the following, we specialize in rotations of order k, where $k \in \{2, \ldots, n\}$ is fixed throughout this chapter. When proceeding from automorphisms to rotations, the tools change significantly: as the automorphisms of a graph form a group, dealing with automorphisms involves a lot of group-theory. The group structure can often be exploited to improve algorithms dealing with automorphisms. Furthermore, the automorphism group of a graph is a well-studied object. On the other hand, the rotations of a graph do not form a group in general. Research on symmetries of abstract graphs is rare. The properties that distinguish rotations from general automorphisms concern the lengths of the orbits, involving divisibility arguments and hence elementary number theory.

In Sect. 3.1, we model the k-rotations of a given graph by extending the automorphism ILP presented in Sect. 2.1. We can use any linear objective function in combination with the rotation ILP. In Sect. 3.2, we consider the subpolytope of the automorphism polytope induced by the k-rotations. Cutting planes of this polytope are described in Sect. 3.3 and separation algorithms are proposed in Sect. 3.4. In Sect. 3.5, we describe rules for fixing and setting variables. For experimental runtime results, see Sect. 5.3.

3.1 The Integer Linear Program

We adjust the automorphism ILP (2.2) presented in Sect. 2.1 to the problem of detecting k-rotations instead of arbitrary automorphisms. Let π be an automorphism of G given by variables x_{ij}, i.e., let $M(\pi) = (x_{ij})$.

The first condition for π to be a k-rotation is $|\mathrm{orb}_\pi(i)| \in \{1, k\}$ for all $i \in V$. All other orbit lengths are forbidden. In order to set up linear constraints to enforce this property of π, we define a *forbidden orbit* of G as a circular list $C = (i_1, \ldots, i_p)$ of pairwise distinct nodes in V with $k \neq p \geq 2$. Let $|C| = p$ and

$$x(C) = \sum_{t=1}^{p-1} x_{i_t i_{t+1}} + x_{i_p i_1} \ .$$

Then the condition $|\mathrm{orb}_\pi(i)| \in \{1, k\}$ for all $i \in V$ is equivalent to

$$x(C) \leq |C| - 1 \ \text{ for all forbidden orbits } C \ . \tag{3.1}$$

Indeed, the constraint (3.1) implies that for every forbidden orbit C at least one variable in $x(C)$ is zero, so that C is not an orbit of π.

To this point, we have ensured that all orbits of π have length k or 1. Any k-rotation has to meet the additional condition $|\mathrm{Fix}(\pi)| \leq 1$. We can express this by

$$\sum_{i \in V} x_{ii} \leq 1 \ , \tag{3.2}$$

as the sum on the left hand side equals the number of fixed nodes.

Combining (2.2), (3.1), and (3.2), an ILP describing k-rotations of G is

$$\begin{array}{rcll} x_{ij} & \in & \{0,1\} & \text{for all } i, j \in V \\ \sum_{j \in V} x_{ij} & = & 1 & \text{for all } i \in V \\ \sum_{i \in V} x_{ij} & = & 1 & \text{for all } j \in V \\ A_G X & = & X A_G & \\ \sum_{i \in V} x_{ii} & \leq & 1 & \\ x(C) & \leq & |C| - 1 & \text{for all forbidden orbits } C \ . \end{array} \tag{3.3}$$

The number of variables in (3.3) is n^2 as in (2.2); see Sect. 2.1. The same methods for reducing this number apply here. In the remainder of this section, we develop criteria for leaving out further variables that are valid only if the algorithm is restricted to k-rotations.

First, we replace the inequality (3.2) by tighter constraints. If k divides n, we may set $x_{ii} = 0$ for all $i \in V$, i.e., we may omit the variables x_{ii}, since each non-trivial orbit of a k-rotation has length k and every fixed node of a k-rotation π would imply $|\mathrm{Fix}(\pi)| \geq k \geq 2$, which is not allowed for k-rotations. Now assume that k does not divide n. Then k divides $n-1$, since otherwise the problem is infeasible. In this case, we can derive that exactly one node is fixed by any k-rotation π of G. Hence we have $\sum_{i\in V} x_{ii} = 1$. Thus, we can always replace (3.2) by equations.

If k does not divide n, we can do more: for $d \in \mathbb{N}$, consider $P_d = \{i \in V \mid c(i) = d\}$. Clearly, each set P_d is fixed by any automorphism of G. Hence k divides $|P_d|$ for all but one d, and for all these d we can omit the variable x_{ii} for all $i \in P_d$.

The following technique of deleting variables is more complicated, but very effective in general. Let $i, j \in V$ and $d = c(i,j)$. Consider the orbit graph G_d of (i,j); see Sect. 1.3.5. By Lemma 1.5, the orbit of i under each automorphism $\pi \in \mathrm{Aut}\, G$ with $\pi(i) = j$ is a directed cycle in G_d. In particular, if there is no directed path from j to i in G_d of length $k-1$, we have $x_{ij} = 0$. Unfortunately, the problem of deciding whether in a given directed graph there is a path of given length from one given node to another is NP-complete, as the longest path problem is NP-hard. Nevertheless, we can try to find necessary conditions for such a path to exist, and leave out x_{ij} if they do not hold. For example, if we relax the problem by allowing multiple nodes in the paths, i.e., if we replace paths by walks, the problem becomes polynomial time solvable. If we do not even find a walk of length $k-1$ from j to i, we surely have $x_{ij} = 0$.

If $k \geq 3$ and all objective function coefficients of the variables x_{ij} with $i \neq j$ are equal, we can also delete variables because of symmetry. Under these conditions, assume that there is some feasible solution of (3.3) corresponding to a rotation π. Then we have equivalent solutions corresponding to π^e for each e with $1 \leq e \leq k$ such that $\gcd(e,k) = 1$. Thus, we can delete $\varphi(k) - 1$ arbitrary variables x_{ij} with $i \neq j$ without losing all optimal solutions, where

$$\varphi(k) = |\{e \in \{1, \ldots, k\} \mid \gcd(e,k) = 1\}|$$

is the Euler φ-function: indeed, at least one of the $\varphi(k)$ rotations π^e remains feasible, since no two of them share any variables with value one. This strategy for variable deletion can be regarded as a way of destroying symmetry in the ILP. Decreasing the number of isomorphic optimal solutions can decrease runtime significantly.

Using the techniques described above, we can often reduce the number of variables in the rotation ILP even more than in the automorphism ILP (2.2). However, the number of constraints increases significantly: we have an exponential number of forbidden orbit constraints. Branch & cut can deal with this problem by not adding all these constraints from the beginning but one after another in the cutting phase; see Sect. 1.5. We need a separation algorithm in this case; see Sect. 3.4.

3.2 The Rotation Polytope

All rotations are automorphisms, hence the k-rotation polytope $P(\mathrm{Rot}_k G)$ is defined in a natural way as the subpolytope of $P(\mathrm{Aut}\, G)$ spanned by all matrices $M(\pi)$ for $\pi \in \mathrm{Rot}_k G$. In the following, we will concentrate on the special case where G is the complete graph K_n. In fact, since

$$\mathrm{Rot}_k G = \mathrm{Aut}\, G \cap \mathrm{Rot}_k K_n \ , \tag{3.4}$$

we have $P(\mathrm{Rot}_k G) \subseteq P(\mathrm{Aut}\, G) \cap P(\mathrm{Rot}_k K_n)$. If k divides n, the latter polytope models all decompositions of the complete graph K_n into node-disjoint directed cycles of length k. If k divides $n-1$, we get all decompositions of K_n into $(n-1)/k$ node-disjoint directed cycles of length k and one isolated node. In all other cases, the polytope $P(\mathrm{Rot}_k K_n)$ is empty. An important special case is $k = n$, where $P(\mathrm{Rot}_k K_n)$ coincides with the asymmetric traveling salesman polytope $\mathrm{ATSP}(n)$ on n nodes, which is well-studied; see e.g. Grötschel and Padberg [36]. Some results for $\mathrm{ATSP}(n)$ will be generalized to $P(\mathrm{Rot}_k K_n)$ in the following. For ease of exposition, we will identify a permutation π with the corresponding matrix $M(\pi)$.

Lemma 3.1
Let $k \geq 3$ and let π be a permutation of V. If the length of each orbit of π is a multiple of k except for at most one trivial orbit, then π is an affine combination of k-rotations of K_n.

Proof: We may assume that $(1, \ldots, mk)$ is an orbit of π with $m > 1$. We show that π is an affine combination of permutations that agree with π on all nodes $mk+1, \ldots, n$ but have the first orbit split into orbits of length k and $(m-1)k$. Repeating this process, we get the desired result.

For $i \in \{1,2,3\}$, let π_i equal π except that i is mapped to $(m-1)k+i+1$ and $(m-1)k+i$ is mapped to $i+1$. Furthermore, let π' equal π except that i is mapped to $(m-1)k+i+1$ and $(m-1)k+i$ is mapped to $i+1$ for all $i \in \{1,2,3\}$. By construction, we have

$$M(\pi) = \frac{1}{2} \sum_{i \in \{1,2,3\}} M(\pi_i) - \frac{1}{2} M(\pi') \ ,$$

and the sum of coefficients on the right hand side is one. It is easily checked that the permutations π_1, π_2, π_3, and π' each split up the set $\{1, \ldots, mk\}$ into one orbit of length k, namely the one containing node 1, and one orbit of length $(m-1)k$. $\square$

Theorem 3.2
Let $k \geq 3$ and $k \mid n$. Then $P(\mathrm{Rot}_k K_n)$ and $P(\mathrm{Rot}_n K_n) = ATSP(n)$ span the same affine space. The dimension of $P(\mathrm{Rot}_k K_n)$ is $n^2 - 3n + 1$.

Proof: The constraints (2.1) and $x_{ii} = 0$ for $i \in V$ form a complete system of equations for ATSP(n). Since these equations are satisfied by all k-rotations of K_n as well, we get

$$\operatorname{aff} P(\mathrm{Rot}_k K_n) \subseteq \operatorname{aff} \mathrm{ATSP}(n) .$$

The converse follows from Lemma 3.1. Finally, the statement on the dimension of $P(\mathrm{Rot}_k K_n)$ is a consequence of the respective result for the asymmetric traveling salesman polytope [36]. □

Theorem 3.3
Let $k \geq 3$ and $k \mid (n-1)$. Then $P(\mathrm{Rot}_k K_n)$ and $P(\mathrm{Rot}_{n-1} K_n)$ span the same affine space. For $n \geq 5$, the dimension of $P(\mathrm{Rot}_k K_n)$ is $n^2 - 2n$.

Proof: Let $\pi \in \mathrm{Rot}_k K_n$. Let π' be the restriction of π to $V \setminus \mathrm{Fix}(\pi)$. By Theorem 3.2, the polytopes $P(\mathrm{Rot}_k K_{n-1})$ and $P(\mathrm{Rot}_{n-1} K_{n-1})$ span the same affine space. Thus π' is an affine combination of $(n-1)$-rotations of $V \setminus \mathrm{Fix}(\pi)$. Adding the trivial orbit of π to each of these, we get π as an affine combination of $(n-1)$-rotations of K_n. Hence $M(\pi) \in \operatorname{aff} P(\mathrm{Rot}_{n-1} K_n)$ and thus

$$\operatorname{aff} P(\mathrm{Rot}_k K_n) \subseteq \operatorname{aff} P(\mathrm{Rot}_{n-1} K_n) .$$

The converse follows from Lemma 3.1.

For the second statement, we may hence assume $k = n-1$. The $2n$ constraints (2.1) and $\sum_{i \in V} x_{ii} = 1$ form a system of independent equations for $P(\mathrm{Rot}_{n-1} K_n)$ after one of the constraints in (2.1) is removed, hence

$$\dim P(\mathrm{Rot}_{n-1} K_n) \leq n^2 - 2n .$$

For $n \geq 5$, this system is complete: we show this by constructing $n^2 - 2n + 1$ affinely independent $(n-1)$-rotations of K_n. First, we have $(n-1)^2 - 3(n-1) + 2$ affinely independent $(n-1)$-rotations of $V \setminus \{1\}$ by Theorem 3.2, giving rise to the same number of affinely independent $(n-1)$-rotations of V satisfying

$$x_{1i} = x_{i1} = x_{ii} = 0 \text{ for all } i \in V \setminus \{1\} .$$

Using the dimensions corresponding to x_{1i} and x_{i1} for $i \in V \setminus \{1, 2\}$, we can choose $2(n-2) - 1$ affinely independent $(n-1)$-rotations of K_n fixing 2. Then we still have

$$x_{12} = x_{21} = 0 \quad \text{and} \quad x_{ii} = 0 \text{ for all } i \in V \setminus \{1, 2\}$$

for all chosen rotations. Next, we choose a rotation fixing i and neither mapping 1 to 2 nor 2 to 1, for each $i \in V \setminus \{1, 2\}$. Finally, we add an arbitrary rotation mapping 1 to 2 and an arbitrary rotation mapping 2 to 1. In summary, we have gathered

$$(n-1)^2 - 3(n-1) + 2 + 2(n-2) - 1 + n - 2 + 2 = n^2 - 2n + 1$$

affinely independent $(n-1)$-rotations of K_n as promised. □

3.3 Cutting Planes

All equations and inequalities valid for $P(\operatorname{Aut} G)$ are still valid for $P(\mathrm{Rot}_k G)$. In particular, we can use homomorphism constraints (2.5) for rotation detection as well; see Sect. 2.4. However, we need more cutting planes now to cut off the automorphisms of G that do not form k-rotations. It is hard to find facet-inducing inequalities for the rotation polytope for general graphs, since even deciding whether this polytope is empty is an NP-complete problem at least for some k; see problem (ROT) in Sect. 1.4. Instead, we will evaluate our cutting planes with respect to the polytope $P(\mathrm{Rot}_k K_n)$. For the remainder of this section, we assume that k divides either n or $n-1$, since $P(\mathrm{Rot}_k K_n)$ is empty otherwise.

Theorem 3.4
Let $k \geq 4$ and $n \geq 5$. Then for all $i, j \in V$ with $i \neq j$, the constraint $x_{ij} \geq 0$ induces a facet of $P(\mathrm{Rot}_k K_n)$.

Proof: First we show that we may assume $k = n$ or $k = n-1$. For this, it suffices to modify the proof of Lemma 3.1 in such a way that, if $\pi(i) \neq j$ for some permutation π the orbit lengths of which are all multiples of k except for at most one trivial orbit, then the constructed k-rotations neither map i to j. Again, repeating this process yields a collection of k-rotations that do not map i to j and that span the same affine space as the rotations of order n or $n-1$ not mapping i to j.

If there is an orbit of length greater than k that does not contain both i and j, we can split up this orbit as in the proof of Lemma 3.1. Otherwise, we may assume that $(1, \ldots, mk)$ is an orbit of π with $m > 1$ containing i and j. Furthermore, we may assume $i = 4$. Since $k \geq 4$, the permutations π_1, π_2, π_3, and π' constructed in the Proof of Lemma 3.1 all agree with π on i.

For $k = n$, we have $P(\mathrm{Rot}_k K_n) = \mathrm{ATSP}(n)$, hence the result follows from [36] as $n \geq 5$. So let $k = n-1$ and assume $i, j \in V \setminus \{1, 2\}$. If $n \geq 6$, we can use the same technique as in the proof of Theorem 3.3: in this case, we can apply the result for the case $k = n$ to $n-1$. We derive that the inequality $x_{ij} \geq 0$ induces a facet of $P(\mathrm{Rot}_{n-1} K_{n-1})$. Thus we can choose $(n-1)^2 - 3(n-1) + 1$ affinely independent $(n-1)$-rotations of $V \setminus \{1\}$ not mapping i to j. Additionally, we get $3n-5$ rotations as in the proof of Theorem 3.3; as $n \geq 6$, we may assume that none of them maps i to j. Hence we have constructed

$$(n-1)^2 - 3(n-1) + 1 + 3n - 5 = n^2 - 2n$$

affinely independent $(n-1)$-rotations of K_n not mapping i to j, so that $x_{ij} \geq 0$ induces a facet of $P(\mathrm{Rot}_{n-1} K_n)$. The case $n = 5$ can be checked explicitly. □

In the following, we examine the forbidden cycle constraint (3.1) more closely. For this, let $C = (i_1, \ldots, i_p)$ be a circular list of distinct nodes with $k \neq p \geq 2$ again. If k does not divide p, we can improve (3.1) to

$$\sum_{i,j \in C,\ i \neq j} x_{ij} \leq p - 1\ . \tag{3.5}$$

Indeed, the sum on the left hand side is at most p by (2.1). If it is exactly p for some permutation π, then C is fixed under π, but no single node of C is fixed. Since k does not divide p, some non-trivial orbit of π must have a length not equal to k, so π is no k-rotation.

If additionally k divides n or k does not divide $p - 1$, we even have

$$\sum_{i,j \in C} x_{ij} \leq p - 1 \tag{3.6}$$

by similar reasoning. Observe that for $k = n$ the constraint (3.6) is the subtour elimination constraint for the cut defined by C.

Theorem 3.5
Let $n \geq 5$ and $k \mid n$. Then the constraint (3.6) induces a facet of $P(\mathrm{Rot}_k K_n)$ if k neither divides p nor $p \pm 1$.

Proof: The idea is the same as in the last proof: we have $2 \leq p \leq n-2$, hence (3.6) induces a facet of ATSP(n) by [36]. Consider a permutation π of V with all orbit lengths a multiple of k that satisfies (3.6) with equality. Then it suffices to show that π is an affine combination of k-rotations satisfying (3.6) with equality.

Since π satisfies (3.6) with equality, there is exactly one pair $(i, j) \in C \times (V \setminus C)$ with $\pi(i) = j$ and exactly one pair $(i, j) \in (V \setminus C) \times C$ with $\pi(i) = j$. In particular, there is exactly one orbit $(1, \ldots, mk)$ of π that is neither contained in C nor contained in $V \setminus C$. All other orbits can be split up as in the proof of Lemma 3.1 without losing equality in (3.6). If $m = 1$, we are ready, so it remains to split up $(1, \ldots, mk)$ for $m > 1$.

We may assume $p < (m-1)k$, otherwise we can exchange C with $V \setminus C$ and argue analogously. We have $p \leq (m-1)k - 2$, since k neither divides p nor $p \pm 1$. Thus at least $k+2$ nodes of $(1, \ldots, mk)$ belong to $V \setminus C$. Since π satisfies (3.6) with equality, the nodes of $V \setminus C$ appear consecutively in $(1, \ldots, mk)$, hence we may assume that the first three and the last $k-1$ nodes in this sequence belong to $V \setminus C$. Using this it is easy to verify that splitting up $(1, \ldots, mk)$ as in Lemma 3.1 again preserves equality in (3.6). □

Observe that Theorem 3.5 is not true in general if k divides $p-1$ or $p+1$. As an example, let $k=3$, $n=6$, and $p=2$ or $p=4$. We have $\dim P(\mathrm{Rot}_3 K_6) = 19$ by Theorem 3.2, whereas it is readily checked that the number of 3-rotations of K_6 satisfying (3.6) with equality is only 16.

As a special case of (3.6), we have a valid inequality

$$x_{ij} + x_{ji} \leq 1 \tag{3.7}$$

if $k \geq 3$ and $i \neq j$. In particular, the constraints $x_{ij} \leq 1$ do not induce facets of $P(\mathrm{Rot}_k K_n)$, in contrary to the constraints $x_{ij} \geq 0$; see Theorem 3.4.

Corollary 3.6
Let $n \geq 5$ and $k \geq 4$ with $k \mid n$. Let $i, j \in V$ with $i \neq j$. Then the constraint (3.7) induces a facet of $P(\mathrm{Rot}_k K_n)$.

If k divides n, it is easy to see that any valid inequality for ATSP(n) with a support graph containing no more than $k-1$ nodes is also valid for $P(\mathrm{Rot}_k K_n)$. If additionally we have $k \geq 3$, then every k-rotation is an asymmetric assignment, hence every odd CAT inequality [4] and every primitive source-destination inequality [5] is valid for $P(\mathrm{Rot}_k K_n)$ as well.

To conclude this section, observe that we have $P(\mathrm{Rot}_k G) \neq P(\mathrm{Aut}\, G) \cap P(\mathrm{Rot}_k K_n)$ in general. In fact, even the dimensions of these polytopes may disagree, as the following example shows: let C_4 denote the undirected cycle on four nodes. Then we have a valid equation $x_{13} = 0$ for $P(\mathrm{Rot}_4 C_4)$, whereas the matrix

$$\frac{1}{2}\left(M((13)(24)) + M((12)(34))\right) = \frac{1}{2}\left(M((1243)) + M((1342))\right)$$

is contained in both $P(\mathrm{Aut}\, C_4)$ and $P(\mathrm{Rot}_4 K_4)$ but does not satisfy $x_{13} = 0$.

3.4 Separation

In this section, we consider the separation problems corresponding to the classes of cutting planes introduced in Sect. 3.1 and Sect. 3.3: the generalized subtour elimination constraints (3.5) are examined in Sect. 3.4.1, the forbidden orbit constraints (3.1) are examined in Sect. 3.4.2.

3.4.1 Subtour Elimination Constraints

Using (2.1), it is easy to see that the subtour elimination constraint (3.5) examined in Sect. 3.3 is equivalent to the inequality

$$\sum_{i \in C,\ j \notin C} x_{ij} + \sum_{i \in C} x_{ii} \geq 1 \ .$$

To separate constraints of this type, we have to minimize the function f defined by

$$f(C) = \sum_{i \in C,\ j \notin C} x_{ij} + \sum_{i \in C} x_{ii}$$

over all subsets C of V with a cardinality not divisible by k. Recall that (3.5) is not valid if k divides $|C|$. It is easy to see that the function f is submodular and that the feasible sets C form a triple family. Hence we have a polynomial time separation algorithm by Grötschel et al. [35]; faster algorithms are given by Goemans and Ramakrishnan [33] and Benczúr and Fülöp [8].

3.4.2 Forbidden Orbit Constraints

Since (3.5) is not valid if k divides $|C|$, we need a separation algorithm for the forbidden orbit constraints (3.1). These are separated heuristically. We first compute a permutation π of V that is close to the current fractional solution. For this, the following very simple heuristic yields good results in general: as in Sect. 2.6, we traverse the pairs $(i, j) \in V^2$ in descending order according to the current value of x_{ij} and set $\pi(i) = j$ if and only if i has no image and j has no preimage under π yet. Now we visit each orbit C of π such that $k \neq |C| \geq 2$ and check whether the corresponding orbit length constraint (3.1) is violated.

3.5 Fixing and Setting Variables

When searching for rotations of order k, we can extend our strategy for fixing or setting variables: consider the simple directed graph given by the mapping variables fixed (set) to one, i.e., the graph $G_1 = (V, E_1)$ with

$$E_1 = \{(i, j) \in V^2 \mid i \neq j \text{ and } x_{ij} \text{ is fixed (set) to one}\} \ .$$

We may assume that G_1 is a disjoint union of directed cycles of length k and directed paths of length at most $k - 1$, since otherwise the fixed (set) variables imply infeasibility. Let P_1 and P_2 be two path components in G_1; let i be the last node of P_1 and j the first node of P_2. Let p_1 be the number of edges in P_1 and p_2 the number of edges in P_2.

If $P_1 = P_2$, we can fix (set) the variable x_{ij} to either zero or one: if $p_1 = k - 1$, the node i must be mapped to j, since otherwise any orbit containing P_1 would have to contain at least $k + 1$ nodes, which is not allowed for rotations of order k. Hence we can fix (set) x_{ij} to one. Otherwise, if $p_1 \neq k - 1$, we may not have $x_{ij} = 1$, since this would determine an orbit consisting of $p_1 + 1$ nodes. In this case, we can fix (set) x_{ij} to zero. Finally, if $P_1 \neq P_2$ and $p_1 + p_2 \geq k - 1$, we can fix (set) x_{ij} to zero. Indeed, if i was mapped to j, we would get a path containing at least k edges, which is infeasible as argued above. In general, we observed that these fixing and setting rules are very effective in practice.

Chapter 4

Detecting Reflections

In the previous chapter, we explained how to detect rotations of fixed order k using the branch & cut-technique. For $k \geq 3$, these rotations are the only symmetries of order k. For $k = 2$, we also have to consider reflections. Each 2-rotation is a reflection, but the converse is not true since the number of fixed nodes is not bounded for reflections. In this chapter, we present an integer programming approach for reflection detection. In principle, we could use the same ILP as for rotations after removing the constraint on the number of fixed nodes. However, the rotation ILP can be simplified in the reflectional case, mainly because we can model the constraints on the orbit lengths much easier now; we do not need the forbidden orbit constraints for reflections.

The reflection ILP is presented in Sect. 4.1. The reflection polytope is examined in Sect. 4.2; cutting planes for this polytope are considered in Sect. 4.3. Sect. 4.4 deals with the corresponding separation problem. Primal heuristics are considered in Sect. 4.5. For experimental runtime results, see Sect. 5.3.

4.1 The Integer Linear Program

Reflections of G are much easier to model than k-rotations for $k \geq 3$. Instead of using the forbidden orbit constraints (3.1) presented in Sect. 3.1, we just have to ensure that the automorphism π to be represented satisfies $\pi^2 = \mathrm{id}_V$, i.e., that in our model we have

$$x_{ij} = x_{ji} \tag{4.1}$$

for all nodes $i, j \in V$. In other words, we only need a single mapping variable for each pair of nodes. Hence an ILP modeling reflections is given by

$$\begin{array}{rcll} x_{ij} = x_{ji} & \in & \{0,1\} & \text{for all } i,j \in V,\ i \leq j \\ \sum_{j \in V} x_{ij} & = & 1 & \text{for all } i \in V \\ \sum_{i \in V} x_{ij} & = & 1 & \text{for all } j \in V \\ A_G X & = & X A_G\,. & \end{array} \tag{4.2}$$

Observe that all constraints in (4.2) are equations. The number of constraints in (4.2) is quadratic, whereas the rotation ILP (3.3) contains an exponential number of constraints.

4.2 The Reflection Polytope

The reflection polytope $P(\operatorname{Ref} G)$ of G is defined as the convex hull of all $M(\pi)$ with $\pi \in \operatorname{Ref} G$. Analogously to (3.4), we have

$$\operatorname{Ref} G = \operatorname{Aut} G \cap \operatorname{Ref} K_n\,. \tag{4.3}$$

We will examine the polytope $P(\operatorname{Ref} K_n)$ in the following. The reflections of K_n correspond bijectively to the permutations of K_n with orbits of length one or two. These in turn correspond bijectively to the matchings in K_n, where fixed nodes correspond to unmatched nodes. Hence we have

Theorem 4.1
The polytope $P(\operatorname{Ref} K_n)$ is isomorphic to the matching polytope. The dimension of $P(\operatorname{Ref} K_n)$ is $(n^2 - n)/2$.

The matching polytope is well examined. The first and ground-breaking investigation was presented by Edmonds [30]. For more information, see Lovász and Plummer [50] or Schrijver [71].

4.3 Cutting Planes

The matching polytope is described completely by trivial constraints and blossom constraints [30]. The trivial constraints correspond to constraints included in (2.1); the blossom constraints translate to

$$\sum_{i,j \in C,\ i \neq j} x_{ij} \leq |C| - 1 \tag{4.4}$$

for all odd subsets C of V. The constraint (4.4) agrees with (3.5) for the case $k = 2$. On the left hand side of (4.4) all variables appear pairwise: along with x_{ij} we have x_{ji}. As $x_{ij} = x_{ji}$ for $i \neq j$, an equivalent constraint is

$$\sum_{i,j \in C,\ i<j} x_{ij} \leq (|C| - 1)/2 \; .$$

Together, the constraints (2.1) and (4.4) describe $P(\operatorname{Ref} K_n)$ completely. In fact, the blossom constraints (4.4) induce facets of $P(\operatorname{Ref} K_n)$ if $n \geq 4$ [70].

However, observe that as for the rotation polytope we have

$$P(\operatorname{Ref} G) \neq P(\operatorname{Aut} G) \cap P(\operatorname{Ref} K_n)$$

in general, so that not all valid inequalities for the reflection polytope of G are given by valid inequalities for the automorphism polytope of G or valid inequalities for the matching polytope of K_n. For example, if G is the directed cycle $(1, 2, 3, 4)$, we have $\operatorname{Ref} G = \{\mathrm{id}_V\}$, but the matrix

$$\frac{1}{2}\left(M((1234)) + M((1432))\right) = \frac{1}{2}\left(M((12)(34)) + M((14)(23))\right)$$

is contained in both $P(\operatorname{Aut} G)$ and $P(\operatorname{Ref} K_4)$.

4.4 Separation

The blossom constraints (4.4) are special constraints of type (3.5) for $k = 2$. Hence they can be separated in polynomial time; see Sect. 3.4. In this case, separation is equivalent to odd cut minimization. The first polynomial time algorithm for this problem was given by Padberg and Rao [66].

4.5 Primal Heuristics

We apply a maximum weight matching algorithm to the complete graph K_n with weights given by the current LP-solution. More precisely, the weight for $(i, j) \in V^2$ is defined as $w_{ij} = \overline{x}_{ij} + \overline{x}_{ji}$ for $i, j \in V$, where $\overline{x}_{ij}$ is the current LP-value of the variable x_{ij}. The exact algorithm runs in $O(n^3 \log n)$ time. Additionally, we use a simple heuristic: we traverse all node-pairs $(i, j) \in V^2$ in descending order according to their weight w_{ij}, and match i with j if and only if both nodes have not been matched to other nodes before.

Both methods yield matchings of K_n that do not necessarily induce reflections of G. However, by the construction of the weights, the heuristic is guided by the current LP-solution and hence tends to find matchings that correspond to reflections.

Chapter 5

Detecting Symmetries

Combining the rotation and reflection detection algorithms presented in the previous chapters, we are able to detect optimal symmetries now. The optimal symmetry π of a graph G is the one maximizing $\operatorname{ord}(\pi)$ and, within the symmetries of maximal order, the one minimizing $|\operatorname{Fix}(\pi)|$. To find the optimal symmetry of G, we traverse all potential orders $k = n, \ldots, 3$ first, trying to find a rotation of order k by the algorithm presented in Chapter 3. Observe that the number of fixed nodes is the same for each k-rotation of G, so that the first rotation we find is already optimal. If we did not find any rotation for $k \geq 3$, we proceed with reflection detection as described in Chapter 4. Here we minimize the number of fixed nodes by an appropriate objective function. The best possible result is a rotation of order two, the worst possible result is the trivial symmetry id_V.

This algorithm could also be packed into a single ILP modeling all symmetries. We did this earlier; see Buchheim and Jünger [19]. For several reasons, we now prefer the approach presented here: the separate ILPs for each order k are much simpler than the one for all orders. Furthermore, the polytopes for special orders are easier to understand and closer to the automorphism polytope, since we do not need additional variables here as we did in the other approach. Finally, experimental results are much better for the new approach. These results show that in most cases we only have to solve a single ILP anyway, since by theoretical considerations we can omit many orders from the outset.

In Sect. 5.1, we explain how to reduce the number of potential orders, i.e., the number of ILPs we have to solve. In Sect. 5.2, we consider the problem of finding a second symmetry that can be displayed simultaneously with the first one. Finally, we include an experimental evaluation of our approach for detecting optimal symmetries in general graphs; see Sect. 5.3.

5.1 Reducing the Number of Potential Orders

The number of potential symmetry orders k and hence the number of ILPs to be solved can often be reduced by using labelings again: let $V^2 = P_1 \oplus \ldots \oplus P_r$ be the partitioning of V^2 according to c or some finer 2-labeling, e.g., the 2-automorphism partitioning c_2. Assume that the pairs (i, i) for $i \in V$ are contained in $P_{s+1} \oplus \ldots \oplus P_r$. If $k \geq 3$, we know that at most one node may be fixed. Hence the size of every single part must be divisible by k, except for at most one part P_l with $l > s$ and size one greater than a number divisible by k. If k does not meet this condition, we can omit the corresponding ILP. In summary, Algorithm 1 finds an optimal symmetry of G.

Algorithm 1: Optimal symmetry detection in a graph G on n nodes

```
foreach $k = n, \ldots, 3$ do
    let feasibleOrder = true;
    let oneFixed = false;
    foreach $l = 1, \ldots, r$ do
        if $k$ does not divide $|P_l|$ then
            if $l \leq s$ or $k$ does not divide $(|P_l| - 1)$ then
                let feasibleOrder = false;
            else
                if oneFixed then
                    let feasibleOrder = false;
                else
                    let oneFixed = true;
    if feasibleOrder then
        detect any rotation of order $k$ in $G$;
        if rotation detected then
            stop;
detect a reflection in $G$ with a minimal number of fixed nodes;
```

Observe that these feasibility criteria include the condition that G can only admit a k-rotation if k either divides n or $n-1$, so that the number of ILPs to solve is bounded by the number of divisors of either n or $n-1$. Let $d(n)$ be the number of divisors of n. By Hardy and Wright [37], we know that for each $\varepsilon > 0$ there is an integer n_0 such that

$$d(n) < 2^{(1+\varepsilon)\ln n/\ln\ln n} \quad \text{for all } n \geq n_0 \, .$$

For $\varepsilon = \log_2 e - 1$, we get $d(n) + d(n-1) \in O(n^{1/\ln\ln n})$. By Dirichlet, the average number of divisors of all numbers from one to n is asymptotic to $\ln n$ up to a small constant. Thus we have to solve $O(\ln n)$ ILPs on average.

5.2 Detecting the Second Symmetry

If the optimal symmetry of G is a rotation, there may be a second symmetry that can be displayed simultaneously; see Lemma 1.7. To find it, we use

Lemma 5.1
Let π_1 be a rotation of G maximizing $k = \operatorname{ord}(\pi_1)$. Then a symmetry π_2 can be displayed simultaneously with π_1 without being contained in (π_1) if and only if π_2 is a non-trivial reflection of G such that $\pi_2\pi_1 = \pi_1^{-1}\pi_2$ and either k is odd or $\pi_2 \neq \pi_1^{k/2}$.

Proof: We prove necessity first. By Lemma 1.7, we must have $\pi_2\pi_1 = \pi_1^{-1}\pi_2$. By the same Lemma, π_2 is a reflection, since π_1 has maximal order. It must be non-trivial to avoid $\pi_2 \in (\pi_1)$. If $\pi_2 \notin (\pi_1)$ and k is even, we surely know $\pi_2 \neq \pi_1^{k/2}$. To prove sufficiency, we know that π_1 and π_2 can be displayed simultaneously by Lemma 1.7. Now assume that $\pi_2 \in (\pi_1)$. Let $\pi_2 = \pi_1^e$ with $e < k$. Since $\pi_1^{2e} = \pi_2^2 = \mathrm{id}_V$, we have $k \mid 2e$ and hence $k = 2e$. Thus k is even and $\pi_2 = \pi_1^{k/2}$. □

To find the second symmetry π_2, we apply the detection algorithm for reflections, see Chapter 4, with additional restrictions $\pi_2\pi_1 = \pi_1^{-1}\pi_2$ and, if $k = \operatorname{ord}(\pi_1)$ is even, $\pi_2 \neq \pi_1^{k/2}$. The first condition is equivalent to

$$x_{\pi_1(i),j} = x_{i,\pi_1(j)} \quad \text{for all } i,j \in V \;, \tag{5.1}$$

the second condition is modeled by

$$\sum_{i\in V} x_{i,\pi_1^{k/2}(i)} \leq n-2 \;. \tag{5.2}$$

So we have the following ILP for finding the second symmetry:

$$\begin{array}{rcll} x_{ij} = x_{ji} & \in & \{0,1\} & \text{for all } i,j \in V,\ i \leq j \\ \sum_{j\in V} x_{ij} & = & 1 & \text{for all } i \in V \\ \sum_{i\in V} x_{ij} & = & 1 & \text{for all } j \in V \\ A_G X & = & X A_G & \\ x_{\pi_1(i),j} & = & x_{i,\pi_1(j)} & \text{for all } i,j \in V \\ \sum_{i\in V} x_{i,\pi_1^{k/2}(i)} & \leq & n-2 & \text{if } k \text{ is even .} \end{array} \tag{5.3}$$

Observe that this ILP may be infeasible; in this case, there is no second symmetry that can be displayed along with π_1 without being contained in (π_1).

If we find a second symmetry, the group generated by both symmetries is not necessarily a symmetry group of G of maximal size. The algorithm of Abelson et al. [2] is designed to find such a group. However, our main objective is to maximize the order of the first symmetry instead of the size of the symmetry group.

5.3 Experimental Results

In this section, we report practical runtime results for symmetry detection in general graphs. More precisely, we solve the NP-complete problem of finding the optimal symmetry as described in Sect. 5.1. In particular, we do not use any specific objective function for rotation detection, i.e., all objective function coefficients are zero. For reflection detection, we minimize the number of fixed nodes.

Our branch & cut-implementation for rotation and reflection detection is based on the implementation for automorphism detection evaluated in Sect. 2.9. The general experimental framework is the same here.

5.3.1 The Algorithm

Again, we first compute the 2-automorphism partitioning of G using `nauty`. As for automorphism detection, the branch & cut-algorithm starts with an ILP containing mapping variables and the permutation constraints (2.1); variables are deleted as described in Sect. 2.1. For rotation detection, we delete further variables and add the fixed nodes constraint (3.2); for both, see Sect. 3.1. For reflection detection, the variables x_{ij} and x_{ji} are identified for all $i, j \in V$; see Sect. 4.1.

In every cutting phase, we add as many violated homomorphism constraints (2.5) for $t = 2$ as possible. Only if no such constraints are found, we proceed to the separation of the different kinds of orbit length constraints: for rotation detection, we separate subtour elimination constraints (3.5) and, if not successful, forbidden orbit constraints (3.1); see Sect. 3.4. For reflection detection, we separate blossom constraints (4.4) as explained in Sect. 4.4. We do not use any other class of cutting planes in our implementation.

The branching and enumeration strategy is the same as described in Sect. 2.7. For rotations, the fixing and setting of variables is extended by the rules given in Sect. 3.5. For primal heuristics, see Sect. 2.6 and Sect. 4.5. For the sake of comparability, we search for a single optimal symmetry without looking for a second symmetry as proposed in Sect. 5.2.

5.3.2 Test Sets

We created a new test set `sym` containing symmetric graphs. For a given number n of nodes, each instance in `sym` was determined as follows: first, we randomly chose a feasible order, i.e., a non-negative integer k that either divides n or $n - 1$. Then we picked a feasible number of fixed nodes randomly, i.e., a non-negative integer f such that k divides $n - f$ and $f \leq 1$ for $k \geq 3$. Finally, we randomly chose a permutation of V with f fixed elements and all other orbits of length k. Using this permutation, we computed an instance as explained in Sect. 2.9.2. We created 100 instances for each $n = 1, \ldots, 50$ by this method. The collection `sym` is also used in [2].

To get harder instances, we also produced a set of highly symmetric graphs called `sym+` in the following way: we started with a random symmetry π_0 chosen as for `sym` and added transpositions π_i of V as long as the partition of V^2 induced by $(\pi_0, \ldots, \pi_i)$ was non-trivial. Again, we created 100 instances for $n = 1, \ldots, 50$. The resulting graphs have many automorphisms in general: we have $|\mathrm{Aut}\, G| \geq \lfloor n/2 \rfloor!$ for 86.0% and $|\mathrm{Aut}\, G| \geq (n-1)!$ for 20.2% of the graphs in `sym+`. For `sym`, the corresponding figures are 14.4% and 8.8%.

By construction, many graphs in `sym` and `sym+` admit symmetries of high order. For comparison purposes, we also experimented with the test set `aut` introduced in Sect. 2.9. In this set, the optimal symmetry order is one or two for most instances.

5.3.3 Results

The runtime results for `sym` are displayed in Table 5.1. Here we also state the number of ILPs that had to be solved; the numbers of variables, subproblems, and LPs are total numbers for all ILPs.

Table 5.1: Results for optimal symmetry detection in `sym`

	runtime		#ILPs		#variables		#subprobs		#LPs	
n	avg	max	avg	max	avg	max	avg	max	avg	max
1–10	0.01	0.90	0.9	2	17.9	180	0.2	76	1.8	159
11–20	0.02	3.14	1.0	2	102.0	380	0.3	92	3.0	216
21–30	0.36	142.86	1.0	2	250.2	926	0.2	46	2.6	134
31–40	0.06	1.81	1.0	1	458.8	1560	0.0	0	1.9	20
41–50	0.12	4.75	1.0	1	675.6	2450	0.0	0	2.1	32

Observe that on average both the number of subproblems and the number of LPs is very low. Table 5.1 also shows that the number of generated subproblems as well as the number of LPs to be solved surprisingly decrease for large graphs in `sym`. This is due to the structure of these graphs: since only one symmetry was created explicitly, it is likely—especially for large graphs—that the automorphism group of the graph is generated by this symmetry alone. In this case, the automorphism partitioning consists of the orbits of this symmetry and many variables can be deleted.

Nevertheless, there is a graph on 24 nodes that required 142.86 CPU-seconds, 46 subproblems, and 134 LPs. This graph is displayed in Fig. 5.1. Its number of automorphisms is 768. The order of the optimal rotation is 24, i.e., the graph is circulant and hence transitive. The total number of constraints added during the branch & cut-process was 594; solving LPs consumed 99.1% of the total CPU-time.

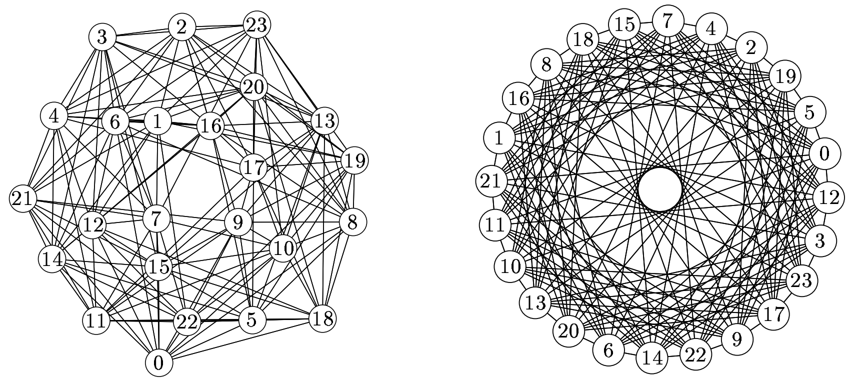


Figure 5.1: The hardest instance in `sym`. The drawing on the left hand side was computed by a spring embedder without symmetry detection; for the drawing on the right hand side, we used our algorithm in a preprocessing step

Recall that our algorithm for rotation detection terminates as soon as any feasible solution is found. Hence the runtime for some instance graph strongly depends on the performance of the primal heuristics on this instance. As the success of primal heuristics is volatile in general, this may explain the wide range of runtimes we observed for symmetry detection. We believe that also the hardness of symmetry detection for the graph examined above is due to incidental failure of primal heuristics rather than deep structural reasons.

We experimented a lot with separation order in the cutting phase, i.e., with different rules whether or when to separate homomorphism constraints or orbit length constraints. However, the runtime increased whenever we tried to separate the latter in every iteration of the cutting phase. One reason may be that the separation algorithms for orbit length constraints require more time in practice than the fast separation heuristics for homomorphism constraints. Moreover, they yield at most one violated constraint per iteration. It turned out that the best strategy is to separate orbit length constraints only if necessary to prevent branching.

Furthermore, we investigated whether we could improve the runtime results by bounding the number of cutting planes added in a cutting phase iteration or the number of iterations per subproblem. We could not.

Next, we applied the branch & cut-algorithm for optimal symmetry detection to the instances in `sym+`. The results are displayed in Table 5.2.

Table 5.2: Results for optimal symmetry detection in `sym+`

	runtime		#ILPs		#variables		#subprobs		#LPs	
n	avg	max	avg	max	avg	max	avg	max	avg	max
1–10	0.01	0.06	0.9	1	26.0	90	0.3	8	2.2	23
11–20	0.64	345.41	1.0	1	193.5	380	17.8	10086	90.5	49731
21–30	5.74	0.7 %	1.0	3	551.5	2268	41.4	8340	287.3	90825
31–40	15.21	1.3 %	1.0	1	1103.6	1560	60.8	10822	379.8	71862
41–50	40.75	11.9 %	1.0	1	1802.0	2450	87.4	10542	564.6	52881

Detecting symmetries for instances in `sym+` needs much more time than for instances in `sym`. In particular, the maximal numbers of subproblems and LPs are huge. This may be due to the large automorphism groups of the graphs. Since automorphisms do not form symmetries in general, the situation is different from automorphism detection, where large automorphism groups leaded to better results; see Sect. 2.9. To examine the impact of the number of automorphisms more closely, we split up the runtime results for $40 < n \leq 50$ by the size of $\operatorname{Aut} G$ in Table 5.3.

Table 5.3: Results for `sym+` by the number of automorphisms for $40 < n \leq 50$

range of	runtime		#variables		#subprobs		#LPs	
$\lvert\operatorname{Aut} G\rvert$	avg	max	avg	max	avg	max	avg	max
$[1, 10!)$	0.37	5.76	2030.1	2450	0.0	0	3.5	32
$[10!, 20!)$	3.48	83.32	1925.7	2450	0.9	24	13.4	97
$[20!, 30!)$	4.11	1.9 %	1295.7	2352	18.9	68	51.5	151
$[30!, 40!)$	63.66	13.0 %	1455.9	2450	211.6	10542	1258.6	52881
$[40!, 50!)$	45.75	15.2 %	1952.7	2450	64.2	10432	468.9	47562

These figures reinforce our general impression that the hardest instances are those with a large but not too large group of automorphisms: the average runtime, number of subproblems, and number of LPs is higher for $\lvert\operatorname{Aut} G\rvert \in [30!, 40!)$ than for any other size of $\operatorname{Aut} G$. For larger automorphism groups, runtime is slightly shorter, but still much longer than for instances with less than 30! automorphisms. Notice that all instances with less than 20! automorphisms could be solved quickly.

We would like to emphasize the fact that we did not manage to create harder instances than those in `sym+`. Whenever we produced instances in a different way or with other parameters, they turned out to be easier to solve.

Finally, we applied the optimal symmetry detection algorithm to the graphs in `aut`; see Table 5.4. Like those in `sym`, the graphs in `aut` were created using a single automorphism, but since this automorphism is no symmetry in general, our algorithm cannot benefit from this fact. Comparing these results with the corresponding results for automorphism detection given in Table 2.1 of Sect. 2.9, we observe that the restriction to geometric automorphisms increases runtime significantly. Therefore, one way to future improvement will be to find more and better constraints cutting off non-geometric automorphisms.

Table 5.4: Results for optimal symmetry detection in `aut`

	runtime		#ILPs		#variables		#subprobs		#LPs	
n	avg	max	avg	max	avg	max	avg	max	avg	max
1–10	0.01	0.06	0.9	1	21.9	90	0.1	6	1.6	22
11–20	0.05	9.06	1.0	2	134.0	524	0.6	30	4.1	116
21–30	1.71	613.03	1.0	3	349.0	1446	2.1	336	9.1	495
31–40	19.36	2734.61	1.0	4	664.4	3184	4.5	166	16.0	485
41–50	22.49	0.7 %	1.0	3	1084.5	2616	11.5	1952	31.3	2817

In summary, our branch & cut-algorithm for detecting symmetries in general graphs works reasonably fast for random symmetric graphs on up to 50 nodes. However, instances needing much more time than acceptable could be constructed. Notice that the group-theoretic approach of Abelson et al. [2] runs significantly faster than ours, but the gap is closing and there is still much room for future improvement.

Chapter 6

Detecting Fuzzy Symmetries

In the previous chapters, we have considered the problem of exact symmetry detection. However, most graphs do not admit any symmetry. In order to make our approach more flexible, we next consider the following more general problem: given a simple graph $G = (V, E)$ on n nodes and a positive integer k that either divides n or $n-1$, find a simple graph $G' = (V, E')$ with a symmetry π of order k such that G' differs minimally from G, i.e., such that the symmetric difference $E \bigtriangleup E'$ has minimal cardinality. In other words, we allow to delete and create a minimal number of edges in G to obtain a graph admitting a symmetry of order k. More generally, we can assign a weight w_{ij} to each pair $(i, j) \in V^2$ and minimize the total weight of all node-pairs with different adjacency in G and G'. In particular, by assigning large enough weights, we can forbid to delete or create certain edges. On the other hand, zero weights can be assigned to node pairs the adjacency of which is irrelevant.

We restrict ourselves to undirected graphs without loops in this chapter. In Sect. 6.1, we explain how to adjust the ILPs for rotation and reflection detection to allow fuzziness. In Sect. 6.2, we define the corresponding polytopes. Cutting planes are considered in Sect. 6.3, the corresponding separation problems are examined in Sect. 6.4. Primal heuristics are presented in Sect. 6.5, branching and enumeration are explained in Sect. 6.6, and a rule for fixing and setting variables is given in Sect. 6.7. We discuss experimental results in Sect. 6.8.

6.1 The Integer Linear Programs

In the following, we adjust the ILPs for rotation and reflection detection presented in Sect. 3.1 and Sect. 4.1 to allow fuzziness. More precisely, we will allow to delete or create edges to find a symmetry. The new ILPs do not only model the symmetry but also its underlying graph. For this, we introduce a new binary variable y_{ij} called *edge variable* for each unordered pair of nodes $(i, j) \in V^2$ with $i \neq j$. The represented graph contains an undirected edge between i and j if and only if $y_{ij} = 1$.

Since the graph G is not fixed any more, the constraints $A_G X = X A_G$ have to be replaced; using the matrix (y_{ij}) instead of A_G would lead to quadratic constraints. The constraints $A_G X = X A_G$ were used to force the permutation represented by the mapping variables to be an automorphism of G. Instead, we use the constraints

$$x_{i_1 j_1} + x_{i_2 j_2} \leq 2 \pm y_{i_1 i_2} \mp y_{j_1 j_2} \tag{6.1}$$

for all $i_1, i_2, j_1, j_2 \in V$ with $i_1 \neq i_2$ and $j_1 \neq j_2$. In both constraints, the left hand side is at most two and the right hand side is at least one. If the left hand side equals two, then i_1 is mapped to j_1 and i_2 is mapped to j_2. But if the right hand side of one of the two constraints is one, then $y_{i_1 i_2} \neq y_{j_1 j_2}$, so that the edge (i_1, i_2) may not be mapped to (j_1, j_2). Consequently, the constraints (6.1) make sure that the permutation represented by the mapping variables is an automorphism of the graph given by the edge variables. Finally, we replace the permutation constraints (2.1) by the weaker constraints

$$\sum_{j \in V} x_{ij} \leq 1 \text{ for all } i \in V \quad \text{and} \quad \sum_{i \in V} x_{ij} \leq 1 \text{ for all } j \in V . \tag{6.2}$$

Thus the permutation corresponding to the mapping variables may be partially or totally undefined. However, by adding a large negative value to all objective function coefficients of mapping variables, every optimal solution will still correspond to a well-defined rotation or reflection. We apply this modification of (2.1) mainly for technical reasons; the resulting polytopes are much easier to investigate. Nevertheless, for reflections, the possibility of not mapping a node i to any node $j \in V$ also has a useful interpretation as deleting node i in G'. This does not work for rotations unless we additionally require $\sum_{j \in V} x_{ij} = \sum_{j \in V} x_{ji}$ for all $i \in V$.

In summary, we get the following ILP for fuzzy rotation detection:

$$\begin{array}{rcll}
x_{ij} & \in & \{0,1\} & \text{for all } i, j \in V \\
y_{ij} = y_{ji} & \in & \{0,1\} & \text{for all } i, j \in V,\ i < j \\
\sum_{j \in V} x_{ij} & \leq & 1 & \text{for all } i \in V \\
\sum_{i \in V} x_{ij} & \leq & 1 & \text{for all } j \in V \\
x_{i_1 j_1} + x_{i_2 j_2} & \leq & 2 \pm y_{i_1 i_2} \mp y_{j_1 j_2} & \text{for all } i_1, i_2, j_1, j_2 \in V,\ i_1 \neq i_2,\ j_1 \neq j_2 \\
\sum_{i \in V} x_{ii} & \leq & 1 & \\
x(C) & \leq & |C| - 1 & \text{for all forbidden orbits } C .
\end{array} \tag{6.3}$$

In the reflection case, we have

$$\begin{array}{rcll} x_{ij} = x_{ji} & \in & \{0,1\} & \text{for all } i,j \in V,\ i \leq j \\ y_{ij} = y_{ji} & \in & \{0,1\} & \text{for all } i,j \in V,\ i < j \\ \sum_{j \in V} x_{ij} & \leq & 1 & \text{for all } i \in V \\ \sum_{i \in V} x_{ij} & \leq & 1 & \text{for all } j \in V \\ x_{i_1 j_1} + x_{i_2 j_2} & \leq & 2 \pm y_{i_1 i_2} \mp y_{j_1 j_2} & \text{for all } i_1, i_2, j_1, j_2 \in V,\ i_1 \neq i_2,\ j_1 \neq j_2\ . \end{array} \tag{6.4}$$

Observe that both ILPs do not depend on the structure of G any more; they are determined by the number of nodes of G. The information about edges of G is stored in the objective function: we minimize the number of manipulations in the graph that are necessary to get a symmetry of the required order, i.e., we minimize

$$\sum_{(i,j) \in V^2 \setminus E} y_{ij} + \sum_{(i,j) \in E} (1 - y_{ij})\ .$$

This objective function is readily adjusted to the problem of finding a modification of G of minimal weight that is necessary to get a symmetry of the required order, where every pair of nodes may have any weight. In particular, we can forbid deletion or creation for every single pair of nodes by assigning a large enough weight to the corresponding edge variable.

Independently, we can still assign arbitrary weights to the mapping variables. For $k = 2$, we have to punish fixed nodes as for exact reflections, since otherwise the graph G together with id_V would form an optimal solution of (6.4). However, the penalty for fixed nodes has to be attuned to the penalty for edge deletion or creation in this case.

6.2 The Fuzzy Symmetry Polytopes

The polytopes corresponding to (6.3) and (6.4) differ significantly from the rotation and reflection polytopes examined in Sect. 3.2 and Sect. 4.2. On one hand, they are complicated by additional variables. On the other hand, they only depend on the number n of nodes and the desired order k, but not on the structure of G. Thus we may define the *fuzzy symmetry polytope* $\mathrm{FSP}(k,n)$ to be the polytope corresponding to (6.3), if $k \geq 3$, or the polytope corresponding to (6.4), if $k = 2$.

Theorem 6.1
The polytope FSP(k,n) is full-dimensional. Its dimension is $\frac{3}{2}n^2 - \frac{1}{2}n$ for $k \geq 3$ and n^2 for $k = 2$.

Proof: For every mapping variable, the corresponding unit vector is a feasible solution of (6.3) or (6.4). The same is true for unit vectors corresponding to edge variables and for the zero vector. □

6.3 Cutting Planes

In the following, we investigate the polyhedral structure of the fuzzy symmetry polytope FSP(k, n). First notice that every valid or facet-inducing constraint yields three other valid or facet-inducing constraints by the following observations:

Lemma 6.2
If a constraint H is valid or facet-inducing for FSP(k, n), then the same is true after replacing each variable y_{ij} in H by $1 - y_{ij}$.

Lemma 6.3
If a constraint H is valid or facet-inducing for FSP(k, n), then the same is true after replacing each variable x_{ij} in H by x_{ji}.

Both results follow from symmetry immediately. In the remainder of this section, we will usually consider only one of the four constraints without explicitly mentioning the other three.

For the following, denote the unit vector corresponding to x_{ij} by e_{ij} and the one corresponding to y_{ij} by e'_{ij}, for all $i, j \in V$.

Theorem 6.4
All constraints $x_{ij} \geq 0$, $y_{ij} \geq 0$, and $y_{ij} \leq 1$ induce facets of FSP(k, n).

Proof: The equation $x_{ij} = 0$ is satisfied by each unit vector except for e_{ij} and by the zero vector. All these vectors belong to FSP(k, n). Hence $x_{ij} \geq 0$ induces a facet of FSP(k, n). For $y_{ij} \geq 0$, we can argue analogously. The last statement follows from Lemma 6.2. □

Whenever showing in the following that some inequality H induces a facet of the polytope FSP(k, n), we show that every vector e_{ij} and e'_{ij} is a linear combination of vectors in FSP(k, n) that satisfy H with equality. For ease of exposition, we call a vector in FSP(k, n) *feasible* if it satisfies H with equality. If a vector is a linear combination of feasible vectors, we shortly call it *combinable*. Thus H induces a facet of FSP(k, n) if all unit vectors e_{ij} and e'_{ij} are combinable; the converse is true if the zero vector does not satisfy H with equality.

The polytopes FSP$(2, n)$ and FSP(k, n) for $k \geq 3$ differ significantly. Therefore, we examine both cases separately: we deal with fuzzy rotations in Sect. 6.3.1 and fuzzy reflections in Sect. 6.3.2.

6.3.1 Fuzzy Rotations

Throughout this section we assume $k \geq 3$. We start with investigating the fixed nodes constraint (3.2):

Theorem 6.5
The constraint (3.2) induces a facet of FSP(k,n).

Proof: For $i \in V$, the unit vector e_{ii} is feasible. Let $i, j \in V$ with $i \neq j$. Choose some $i' \in V \setminus \{i, j\}$. Then both $e_{ij} + e_{i'i'}$ and $e'_{ij} + e_{i'i'}$ are feasible, so that both e_{ij} and e'_{ij} are combinable. □

Theorem 6.6
The constraints (6.2) induce facets of FSP(k,n).

Proof: By Lemma 6.3 and symmetry, it suffices to show the result for the single inequality

$$\sum_{j \in V} x_{1j} \leq 1 \; . \tag{6.5}$$

For all $j \in V$, the vector e_{1j} is feasible. For $i \in V \setminus \{1\}$ and $j \in V$, define

$$v_{ij} = \begin{cases} e_{ij} + e_{1i} & \text{if } j \notin \{1, i\} \\ e_{ij} + e_{1j'} & \text{if } j \in \{1, i\}, \text{ where } j' \notin \{1, i\} \; . \end{cases}$$

In both cases, the vector v_{ij} is feasible, showing that e_{ij} is combinable. Hence all unit vectors corresponding to mapping variables are combinable. Finally, if $i, j \in V$ with $i \neq j$, the vector $e'_{ij} + e_{11}$ is feasible, so that e'_{ij} is combinable, too. □

We continue with a review of the subtour elimination constraints (3.5) and (3.6). Let C be a subset of V containing p nodes.

Theorem 6.7
Let $2 \leq p \leq n-1$ and assume that k does not divide p. If k divides $p-1$, then (3.5) induces a facet of FSP(k,n). Otherwise, (3.6) induces a facet of FSP(k,n).

Proof: For any $F \subseteq V^2$, let $e_F = \sum_{(i,j) \in F} e_{ij}$. First notice that for every directed path P we have $e_P \in \mathrm{FSP}(k,n)$. If P is a directed path on p nodes within C, then e_P satisfies both (3.5) and (3.6) with equality, hence e_P is feasible. Furthermore, if F is a cycle on p nodes within C, then e_F is a linear combination of p vectors e_P, where each set P is a directed path on p nodes within C, resulting from the deletion of a single edge in F. Hence e_F is combinable.

Let $i, j \in C$. If $i \neq j$, choose a directed cycle F on p nodes within C containing (i, j) and let $P = F \setminus \{(i, j)\}$. Then $e_{ij} = e_F - e_P$ is combinable. Next, assume $i = j$. If $k \mid (p-1)$, let F be a node-disjoint union of $(p-1)/k$ directed cycles within $C \setminus \{i\}$ on k nodes each. Otherwise, let F be a directed path on $p - 1$ nodes within $C \setminus \{i\}$. In the first case, the vector $e_F + e_{ii}$ is feasible for (3.5); in the second case, it is feasible for (3.6). Since e_F is a sum of unit vectors e_{ij} with $i \neq j$, the vector e_F and hence e_{ii} is combinable as well.

Now let $i \in V \setminus C$ or $j \in V \setminus C$. Let P be a path on p nodes within C, starting at j if $j \in C$ and ending at i if $i \in C$. Then both e_P and $e_P + e_{ij}$ are feasible, hence the unit vector e_{ij} is combinable. Up to now, we have combined all unit vectors corresponding to mapping variables.

Finally, let $i, j \in V$ with $i \neq j$. Let P be a path on p nodes within C, ending at i if $i \in C$. Then both e_P and $e_P + e'_{ij}$ are feasible, hence e'_{ij} is combinable. □

Next, let W_1 and W_2 be any two subsets of V. Let $i_1, i_2 \in V \setminus (W_1 \cup W_2)$ with $i_1 \neq i_2$. Then we have the following valid inequality for FSP(k, n):

$$\sum_{j_1 \in W_1} x_{i_1 j_1} + \sum_{j_2 \in W_2} x_{i_2 j_2} \leq 2 - y_{i_1 i_2} + \sum_{j_1 \in W_1,\ j_2 \in W_2} y_{j_1 j_2} \,. \tag{6.6}$$

Indeed, the left hand side is at most two by (6.2), while the right hand side is at least one. If the left hand side is exactly two, we have $x_{i_1 j_1} = 1$ for some $j_1 \in W_1$ and $x_{i_2 j_2} = 1$ for some $j_2 \in W_2$. Thus $y_{i_1 i_2} = 1$ implies $y_{j_1 j_2} = 1$, so that the right hand side is at least two.

Theorem 6.8
The constraint (6.6) induces a facet of FSP(k, n) if and only if

- *(a)* $|W_1| \neq 0$ *and* $|W_2| \neq 0$,
- *(b)* $|W_2 \setminus W_1| \neq 1$ *and* $|W_1 \setminus W_2| \neq 1$, *and*
- *(c)* $|W_1 \cup W_2| \geq 2$.

Proof: We assume $i_1 = 1$ and $i_2 = 2$. First we show sufficiency. By (a) and (c), we can choose nodes $w_1 \in W_1$ and $w_2 \in W_2$ with $w_1 \neq w_2$. Furthermore, for each $j \in W_2 \setminus W_1$, we can choose some $j' \in (W_2 \setminus W_1) \setminus \{j\}$ by (b). For $j \in W_1 \setminus W_2$ we define j' analogously.

Now let $i, j \in V$ with $i \neq j$. We define

$$w_{ij} = \begin{cases} e_{1w_1} + e_{2w_2} & \text{if } i = 1,\ j = 2 \\ e'_{ij} + e_{1i} + e_{2j} + e'_{12} & \text{if } i \in W_1,\ j \in W_2 \\ e'_{ij} + e_{1w_1} + e'_{12} & \text{otherwise.} \end{cases}$$

Furthermore, for any $i, j \in V$, let

$$v_{ij} = \begin{cases} e_{1j} + e'_{12} & \text{if } i = 1,\ j \in W_1 \\ e_{1j} + e_{2j'} + e'_{12} + e'_{jj'} & \text{if } i = 1,\ j \in W_2 \setminus W_1 \\ e_{1j} + e_{2w_2} + e'_{12} + e'_{jw_2} & \text{if } i = 1,\ j \notin W_1 \cup W_2 \\ e_{2j} + e'_{12} & \text{if } i = 2,\ j \in W_2 \\ e_{2j} + e_{1j'} + e'_{12} + e'_{jj'} & \text{if } i = 2,\ j \in W_1 \setminus W_2 \\ e_{2j} + e_{1w_1} + e'_{12} + e'_{jw_1} & \text{if } i = 2,\ j \notin W_1 \cup W_2 \\ e_{ij} + e_{1w_1} + e'_{12} & \text{if } i \notin \{1,2\},\ j \notin \{1, w_1\} \\ e_{ij} + e_{2w_2} + e'_{12} & \text{if } i \notin \{1,2\},\ j \in \{1, w_1\}\ . \end{cases}$$

It is verified easily that all vectors w_{ij} and v_{ij} are feasible. So it suffices to show that all unit vectors are linear combinations of these vectors. First observe that

$$e'_{12} = -(w_{12} - v_{1w_1} - v_{2w_2})/2\ .$$

Moreover, we have

$$\begin{aligned} e_{1j} &= v_{1j} - e'_{12} && \text{if } j \in W_1 \text{ and} \\ e_{2j} &= v_{2j} - e'_{12} && \text{if } j \in W_2\ . \end{aligned}$$

Hence we get the remaining unit vectors corresponding to edge variables as follows:

$$\begin{aligned} e'_{ij} &= w_{ij} - e_{1i} - e_{2j} - e'_{12} && \text{if } i \in W_1,\ j \in W_2 \text{ and} \\ e'_{ij} &= w_{ij} - e_{1w_1} - e'_{12} && \text{otherwise.} \end{aligned}$$

Finally, we have

$$\begin{aligned} e_{1j} &= v_{1j} - e_{2j'} - e'_{12} - e'_{jj'} && \text{if } j \in W_2 \setminus W_1\ , \\ e_{2j} &= v_{2j} - e_{1j'} - e'_{12} - e'_{jj'} && \text{if } j \in W_1 \setminus W_2\ , \\ e_{1j} &= v_{1j} - e_{2w_2} - e'_{12} - e'_{jw_2} && \text{if } j \notin W_1 \cup W_2\ , \\ e_{2j} &= v_{2j} - e_{1w_1} - e'_{12} - e'_{jw_1} && \text{if } j \notin W_1 \cup W_2\ , \\ e_{ij} &= v_{ij} - e_{1w_1} - e'_{12} && \text{if } i \notin \{1,2\},\ j \notin \{1, w_1\}, \text{ and} \\ e_{ij} &= v_{ij} - e_{2w_2} - e'_{12} && \text{if } i \notin \{1,2\},\ j \in \{1, w_1\}\ . \end{aligned}$$

It remains to show the necessity of the conditions (a) to (c). First, let $W_1 = \emptyset$. Then (6.6) is improved by (6.2), i.e., the constraint (6.2) is strictly more powerful than (6.6), so that (6.6) cannot induce a facet of $\mathrm{FSP}(k, n)$. Hence (a) is necessary.

Next, let $W_1 \setminus W_2 = \{i\}$. Adding i to W_2, the left hand side of (6.6) gets a new addend x_{2i}, but the right hand side is not changed: the only possible new addend on the right hand side is y_{ij} for some $j \in W_1 \setminus \{i\}$. Since $W_1 \setminus W_2 = \{i\}$, we have $j \in W_2$, hence y_{ij} has already been part of the right hand side before adding i to W_2. Thus adding i improves (6.6), so that (b) is necessary.

Finally, assume $|W_1 \cup W_2| \leq 1$. If $W_1 = \emptyset$ or $W_2 = \emptyset$, we have a violation of condition (a). Otherwise, we have $W_1 = W_2 = \{i\}$. Then (6.6) is improved by (6.2) again, so that (c) is necessary, too. □

In the special case $W_1 = W_2 = \{j_1, j_2\}$, the constraint (6.6) and its counterpart according to Lemma 6.2 emerge as

$$x_{i_1j_1} + x_{i_1j_2} + x_{i_2j_1} + x_{i_2j_2} \leq 2 \pm y_{i_1i_2} \mp y_{j_1j_2} .$$

These constraints improve (6.1); both induce facets of FSP(k, n) by Theorem 6.8.

Next, let $W \subseteq V$ and $w \in V \setminus W$. Let $i_1, i_2 \in V \setminus (W \cup \{w\})$ with $i_1 \neq i_2$. Then

$$\sum_{j \in W} (x_{i_1j} + x_{i_2j}) + 2x_{i_1w} + 2x_{i_2w} \leq 3 - y_{i_1i_2} + \sum_{j \in W} y_{jw} \tag{6.7}$$

is a valid inequality for FSP(k, n). Indeed, the left hand side is at most three by (6.2), while the right hand side is at least two. If the left hand side is equal to three, one of the nodes i_1 and i_2 is mapped to w, while the other one is mapped to some $j \in W$. Hence $y_{i_1i_2} = y_{jw}$, so that the right hand side is at least three.

Theorem 6.9
The constraint (6.7) induces a facet of FSP(k, n) if and only if $|W| \geq 2$.

Proof: We assume $i_1 = 1$ and $i_2 = 2$ again. Let $w_1, w_2 \in W$ with $w_1 \neq w_2$. For $i, j \in V$ with $i \neq j$ define

$$w_{ij} = \begin{cases} e_{1w_1} + e_{2w_2} + e'_{12} + e'_{w_1w_2} & \text{if } i = 1,\ j = 2 \\ e'_{wj} + e_{1w} + e_{2j} + e'_{12} & \text{if } i = w,\ j \in W \\ e'_{ij} + e_{1w} + e'_{12} & \text{otherwise.} \end{cases}$$

For $i, j \in V$ define

$$v_{ij} = \begin{cases} e_{1w} + e'_{12} & \text{if } i = 1,\ j = w \\ e_{1j} + e_{2w} & \text{if } i = 1,\ j \in W \\ e_{1j} + e_{2w} + e'_{12} + e'_{jw} & \text{if } i = 1,\ j \notin W \cup \{w\} \\ e_{2w} + e'_{12} & \text{if } i = 2,\ j = w \\ e_{2j} + e_{1w} & \text{if } i = 2,\ j \in W \\ e_{2j} + e_{1w} + e'_{12} + e'_{jw} & \text{if } i = 2,\ j \notin W \cup \{w\} \\ e_{ij} + e_{1w} + e'_{12} & \text{if } i \notin \{1,2\},\ j \notin \{1, w\} \\ e_{i1} + e_{2w} + e'_{12} & \text{if } i \notin \{1,2\},\ j = 1 \\ e_{iw} + e_{1w_1} + e_{2w_2} + e'_{12} + e'_{w_1w_2} & \text{if } i \notin \{1,2\},\ j = w\ . \end{cases}$$

All vectors w_{ij} and v_{ij} are feasible, hence again it suffices to show that they span all dimensions. We have

$$e'_{12} = (w_{12} - w_{w_1w_2} + 2v_{1w} + v_{2w} - v_{1w_1} - v_{2w_2})/3\ ,$$

hence we can combine

$$\begin{aligned} e_{1w} &= v_{1w} - e'_{12}\ , \\ e_{2w} &= v_{2w} - e'_{12}\ , \\ e_{1j} &= v_{1j} - e_{2w} \quad \text{if } j \in W, \text{ and} \\ e_{2j} &= v_{2j} - e_{1w} \quad \text{if } j \in W\ . \end{aligned}$$

For the unit vectors corresponding to edge variables, we have

$$\begin{array}{rcll} e'_{wj} & = & w_{wj} - e_{1w} - e_{2j} - e'_{12} & \text{if } j \in W, \text{ and} \\ e'_{ij} & = & w_{ij} - e_{1w} - e'_{12} & \text{otherwise.} \end{array}$$

Finally,

$$\begin{array}{rcll} e_{1j} & = & v_{1j} - e_{2w} - e'_{12} - e'_{jw} & \text{if } j \notin W \cup \{w\}\ , \\ e_{2j} & = & v_{2j} - e_{1w} - e'_{12} - e'_{jw} & \text{if } j \notin W \cup \{w\}\ , \\ e_{ij} & = & v_{ij} - e_{1w} - e'_{12} & \text{if } i \notin \{1,2\},\ j \notin \{1,w\}\ , \\ e_{i1} & = & v_{i1} - e_{2w} - e'_{12} & \text{if } i \notin \{1,2\},\ j = 1, \text{ and} \\ e_{iw} & = & v_{iw} - e_{1w_1} - e_{2w_2} - e'_{12} - e'_{w_1 w_2} & \text{if } i \notin \{1,2\},\ j = w\ . \end{array}$$

To show the necessity of $|W| \geq 2$, assume $W = \{i\}$. Then (6.7) can be improved to $x_{1w} + x_{1i} + x_{2w} + x_{2i} \leq 2 - y_{12} + y_{iw}$, since $x_{1w} + x_{2w} \leq 1$ by (6.2). If $W = \emptyset$, we can improve (6.7) to (6.2). □

6.3.2 Fuzzy Reflections

Next, we examine the polytope FSP$(2, n)$ more closely. We start with the weak permutation constraints (6.2):

Theorem 6.10
The constraints (6.2) induce facets of FSP$(2,n)$.

Proof: It suffices to show the result for

$$\sum_{j \in V} x_{1j} \leq 1\ . \tag{6.8}$$

For all $j \in V$, the vector e_{1j} is feasible. For $i, j \in V \backslash \{1\}$, the vector $e_{ij} + e_{11}$ is feasible, showing that e_{ij} is combinable. Finally, if $i, j \in V$ with $i \neq j$, the vector $e'_{ij} + e_{11}$ is feasible, so that e'_{ij} is combinable, too. □

Theorem 6.11
Let C be an odd subset of V with $|C| \geq 3$. Then the blossom constraint (4.4) induces a facet of FSP$(2,n)$.

Proof: This constraint induces a facet of the matching polytope by Schrijver [70]. By Theorem 4.1, all unit vectors e_{ij} for $i, j \in V$ with $i \neq j$ are combinable. For $i = j$, choose pairwise distinct nodes $i_1, j_1, \ldots, i_p, j_p \in C \backslash \{i\}$, where $p = (|C| - 1)/2$. Then the vector $e_{ii} + \sum_{t=1}^{p} e_{i_t j_t}$ is feasible, so that e_{ii} is combinable.

Finally, let $i, j \in V$ with $i \neq j$. Whether i or j belong to C or not, we can find a reflection of K_n that maps i to j and that satisfies (4.4) with equality; let v be the corresponding feasible vector in FSP$(2, n)$. Then $e'_{ij} + v$ is feasible, too, so that e'_{ij} is combinable. □

The constraint (6.6) is valid for fuzzy reflections as well. However, we can improve it by adding $x_{i_1 i_2}$ to the left hand side: if $x_{i_1 i_2} = 1$, all other variables on the left hand side must be zero. Hence we get

$$x_{i_1 i_2} + \sum_{j_1 \in W_1} x_{i_1 j_1} + \sum_{j_2 \in W_2} x_{i_2 j_2} \leq 2 - y_{i_1 i_2} + \sum_{j_1 \in W_1,\ j_2 \in W_2} y_{j_1 j_2} \ . \tag{6.9}$$

Theorem 6.12
The constraint (6.9) induces a facet of FSP$(2, n)$ if and only if

(a) $|W_1| \neq 0$ *and* $|W_2| \neq 0$,

(b) $|W_2 \setminus W_1| \neq 1$ *and* $|W_1 \setminus W_2| \neq 1$, *and*

(c) $|W_1 \cup W_2| \geq 2$.

Proof: The proof is similar to the one of Theorem 6.8. However, in the reflection case we only have one unit vector $e_{ij} = e_{ji}$ for each unordered pair $(i, j) \in V^2$. The following cases in the definition of v_{ij} have to be changed:

$$v_{ij} = \begin{cases} e_{12} + e'_{12} & \text{if } \{i,j\} = \{1,2\} \\ e_{ij} + e_{12} + e'_{12} & \text{if } i, j \notin \{1,2\} \\ v_{ji} & \text{if } i \notin \{1,2\},\ j \in \{1,2\} \ . \end{cases}$$

The remainder of the proof is completely analogous to the one of Theorem 6.8. □

By Theorem 6.12, the constraint (6.1) is improved by the facet-inducing constraints

$$x_{i_1 i_2} + x_{i_1 j_1} + x_{i_1 j_2} + x_{i_2 j_1} + x_{i_2 j_2} \leq 2 \pm y_{i_1 i_2} \mp y_{j_1 j_2} \ .$$

Next, observe that (6.7) is true for fuzzy reflections as well. Again, we can improve this constraint to

$$2x_{i_1 i_2} + \sum_{j \in W} (x_{i_1 j} + x_{i_2 j}) + 2x_{i_1 w} + 2x_{i_2 w} \leq 3 - y_{i_1 i_2} + \sum_{j \in W} y_{jw} \ . \tag{6.10}$$

Theorem 6.13
The constraint (6.10) induces a facet of FSP$(2, n)$ if and only if $|W| \geq 2$.

Proof: Exactly the same redefinition as in the proof of Theorem 6.12 allows to adjust the proof of Theorem 6.9. □

6.4 Separation

In the following, we consider the separation problems for the cutting planes presented in Sect. 6.3. For the subtour elimination constraints (3.5) and (3.6), see Sect. 3.4. For the blossom constraints (4.4), see Sect. 4.4. Constraints of type (6.6) and (6.9) are examined in Sect. 6.4.1, constraints of type (6.7) and (6.10) in Sect. 6.4.2.

6.4.1 Constraints of Type (6.6) and (6.9)

We do not know whether the separation problem for constraints of type (6.6) or (6.9) is NP-complete, but we conjecture that it is. We separate these constraints heuristically. The algorithm works greedily: we traverse all edge variables $y_{i_1 i_2}$ in descending order according to their current LP-value $\overline{y}_{i_1 i_2}$. For two given nodes i_1 and i_2, we start with $W_1 = W_2 = \emptyset$ and traverse all mapping variables x_{ij} with $i \in \{i_1, i_2\}$ in descending order. We add j to W_1 if $i = i_1$ and if this increases the difference between the left and the right hand side of (6.6) or (6.9), i.e., if

$$\overline{x}_{i_1 j} > \sum_{j_2 \in W_2,\ (j_2, j) \notin W_1 \times W_2} \overline{y}_{j j_2} \ .$$

Analogously, we add j to W_2 if $i = i_2$ and

$$\overline{x}_{i_2 j} > \sum_{j_2 \in W_1,\ (j, j_2) \notin W_1 \times W_2} \overline{y}_{j j_2} \ .$$

For the resulting sets W_1 and W_2, we check whether (6.6) or (6.9) is violated by the current LP-solution. The runtime of this separation heuristic is $O(n^4)$. In practice, violated constraints are usually found very quickly.

6.4.2 Constraints of Type (6.7) and (6.10)

Separation for constraints of type (6.7) or (6.10) is straightforward:

Theorem 6.14
The separation problem for constraints of type (6.7) or (6.10) can be solved in a runtime of $O(n^4)$.

Proof: Traverse all $O(n^3)$ combinations of $i_1, i_2, w \in V$ with $i_1 \neq i_2$ and $w \notin \{i_1, i_2\}$. For a fixed combination, (6.7) or (6.10) is violated for some subset $W \subseteq V \setminus \{i_1, i_2, w\}$ if and only if it is violated for

$$W = \{j \in V \setminus \{i_1, i_2, w\} \mid \overline{x}_{i_1 j} + \overline{x}_{i_2 j} - \overline{y}_{jw} \geq 0\} \ .$$

This set can be determined in $O(n)$ time. □

6.5 Primal Heuristics

The first type of primal heuristics used for fuzzy symmetry detection is based on the primal heuristics for exact symmetry detection: in the first step, we use the latter to find a symmetry π of K_n that is close to the current LP-values of the mapping variables. We use the methods presented in Sect. 2.6.

In the second step, we determine a graph that admits the symmetry π and that differs from the current LP-values of the edge variables minimally. More precisely, we determine a set $E' \subseteq V^2$ such that π is a symmetry of the graph $G' = (V, E')$ and such that

$$\sum_{(i,j)\in V^2\setminus E'} \overline{y}_{ij} + \sum_{(i,j)\in E'} (1 - \overline{y}_{ij}) \tag{6.11}$$

is minimal, where $\overline{y}_{ij}$ denotes the current LP-value of y_{ij}. This can be done easily: as the permutation of nodes is determined by π, the same is true for the orbitals of π, i.e., the orbits of node-pairs. Since all orbitals are disjoint, the minimization of (6.11) can be performed separately for each orbital of π. For a single orbital, either all edges or none have to be present in E'. The minimization is obtained by adding all edges if and only if the average of all edge variable LP-values in this orbital is at least 1/2.

We also use a primal heuristic working the other way around: we consider the graph $G = (V, E')$ that is as close as possible to the current edge variable LP-values, i.e., we set

$$E' = \{(i,j) \in V^2 \mid \overline{y}_{ij} > 1/2\} \ ,$$

and compute a partially defined symmetry π as follows: we traverse all $(i,j) \in V^2$ by descending value of $\overline{x}_{ij}$. We set $\pi(i) = j$ if and only if the partial definition of π obtained so far does not violate (6.3) or (6.4). The result is not a well-defined symmetry in general, but it yields an upper bound on the objective function value of all feasible solutions and may hence be useful for pruning subproblems. This heuristic works particularly well for reflection detection.

6.6 Branching and Enumeration

The branching and enumeration strategy for fuzzy symmetry detection is the same as for automorphism detection; see Sect. 2.7. This is possible in spite of the newly introduced edge variables, as we can always branch on mapping variables: if all mapping variables assume integer values, the same follows for all edge variables, since the only constraints bounding edge variables are the constraints of type (6.1).

Notice that isomorphism pruning for fuzzy symmetry detection is useless in general: if edge variables corresponding to adjacent node pairs in G have different objective function coefficients than those corresponding to non-adjacent node-pairs, then the symmetry groups of the fuzzy symmetry ILPs are isomorphic to subgroups of $\operatorname{Aut} G$, which in turn will be trivial in general.

6.7 Fixing and Setting Variables

If enough mapping variables are fixed or set to one, it is possible to fix (set) edge variables by logical implications. More precisely, as soon as the node-pairs $(i, j) \in V^2$ with x_{ij} fixed (set) to one form one or more cycles of K_n, we can also fix (set) every edge variable $y_{i'j'}$ such that both i' and j' belong to any of these cycles. Indeed, for each such i' and j', the orbital of (i', j') is fixed (set) completely. Hence we can determine the optimal value of all edge variables in this orbital by a similar reasoning as in Sect. 6.5: we fix (set) all edge variables to one if the number of node-pairs in the orbital that are adjacent in G is larger than the number of non-adjacent node-pairs; otherwise, we fix (set) all edge variables to zero.

6.8 Experimental Results

The fuzzy symmetry detection approach we presented in this chapter is not fully developed yet. It is necessary and certainly possible to improve it significantly by future work. Nevertheless, we implemented a branch & cut-algorithm based on the theoretic results obtained so far. In the evaluation, we searched for fuzzy reflections, fuzzy rotations of order n, and fuzzy automorphisms. Every modification of the graph was punished by an objective function coefficient of one. In the reflection case, the same was true for fixed nodes, so that one edge modification was allowed to prevent one fixed node. We also experimented with allowing to delete edges but not to create them. The general evaluation framework is the same as in Sect. 2.9 and Sect. 5.3, except that we now restrict the number of cutting phase iterations per subproblem to five. This restriction yielded the best results.

6.8.1 The Algorithm

To detect fuzzy symmetries, we cannot use automorphism partitionings any more, so the preprocessing step using `nauty` is dropped. Our initial ILP contains all mapping and edge variables as well as all weak permutation constraints (6.2); see Sect. 6.1.

In every iteration of the cutting phases, we add at most ten violated constraints. Analogously to exact symmetry detection, we first separate fuzzy homomorphism constraints of type (6.6) or (6.9) and, if not successful, of type (6.7) or (6.10). For both separation problems, see Sect. 6.4. If no violated constraints are found, we try to separate an orbit length constraint as for exact symmetry detection; see Sect. 5.3.

The branching and enumeration strategy is not changed; see Sect. 6.6. For fixing and setting variables, we implemented the extension presented in Sect. 6.7. Primal heuristics are used as explained in Sect. 6.5.

6.8.2 Test Sets

As we are not searching for exact symmetries any more, we can now use random simple undirected graphs for our experiments. Each pair of different nodes in our test instances is adjacent by a probability of 1/4. Observe however that random graphs are hard instances for fuzzy symmetry detection, since in general a lot of modifications are necessary to make them symmetric. For every number n of nodes, we tested 100 graphs in each evaluation.

6.8.3 Results

Compared with our results for exact symmetry or automorphism detection, runtimes for fuzzy symmetry detection are rather disappointing. The results for reflection detection are displayed in Table 6.1. This time, we only have to solve a single ILP per instance. The number of variables in this ILP is determined by n. Since we can use our algorithm heuristically by stopping it at any time, we also list the CPU-time needed to find an optimal solution—without knowing its optimality at this point.

Table 6.1: Results for fuzzy reflection detection

	runtime		opttime		#subprobs		#LPs	
n	avg	max	avg	max	avg	max	avg	max
8	0.04	0.32	0.00	0	4.0	25	13.2	79
9	0.13	1.58	0.01	1	8.8	57	28.9	201
10	0.53	3.15	0.04	2	20.7	95	68.8	317
11	2.01	14.46	0.43	5	48.4	263	160.9	894
12	11.92	75.08	1.86	49	167.5	931	567.8	3185

Table 6.1 shows that the runtime, the number of subproblems, and the number of LPs increase sharply already for small graphs. However, the time needed to find the

optimal solution is much shorter than the total runtime, i.e., we know the optimal solution much earlier than the fact that it is optimal. We conclude that the primal heuristics used in our algorithm work well, whereas the cutting planes and separation algorithms must be improved in order to get a practically useful algorithm.

In fact, a lot of violated cutting planes were usually found and added without changing the objective value of the optimal LP-solution, i.e., without increasing the local lower bound on the optimal solution of the ILP. This motivated the restriction of the number of cutting phase iterations per subproblem to five as mentioned above: allowing more iterations often led to a very large number of iterations with a lot of added constraints but without any improvement of the local lower bound. For the same reason, we restricted the number of constraints added in a single iteration to ten. On the other hand, using less than five iterations per subproblem or even switching off the cutting phases completely, i.e., running a pure branch & bound-algorithm, also increased runtime.

The results displayed in Table 6.1 are much more homogeneous than those for exact symmetry detection, i.e., the average and maximal figures do not differ as much. This is even more evident if we split up the results by the optimal objective function value, i.e., by the minimal number of modifications and fixed nodes; see Table 6.2.

Table 6.2: Results for fuzzy reflection detection, $n = 12$

obj	% of	runtime		opttime		#subprobs		#LPs	
val	insts	avg	max	avg	max	avg	max	avg	max
1	6	0.17	0.46	0.00	0	9.0	21	26.2	62
2	17	1.18	5.81	0.59	5	30.5	97	99.4	333
3	30	4.12	10.85	1.03	8	75.0	155	249.0	489
4	33	13.44	24.32	1.33	15	192.6	311	660.5	1031
5	13	40.72	60.22	7.77	49	511.0	791	1729.1	2526
6	1	75.08	75.08	0.00	0	931.0	931	3185.0	3185

These results reveal a strong connection between the hardness of an instance and its distance from being reflectional symmetric. This is good news, as the fuzzy symmetry detection algorithm is designed to draw nearly symmetric graphs.

Now consider the case $k = n$. Here, the problem is to modify the graph as little as possible to make it circulant. The runtime results are displayed in Table 6.3. We see that fuzzy rotation detection is even harder than fuzzy reflection detection: runtimes are longer by a factor of more than 25 for $n = 12$. One reason may be the smaller number of mapping variables in the fuzzy reflection ILP. When splitting up the results of Table 6.3 by the number of necessary modifications, we obtained the same general picture as in Table 6.2: runtime is the longer the more modifications we have to perform to make the graph circulant.

Table 6.3: Results for fuzzy n-rotation detection

	runtime		opttime		#subprobs		#LPs	
n	avg	max	avg	max	avg	max	avg	max
8	0.10	1.56	0.00	0	12.1	135	28.6	299
9	1.67	45.07	0.03	2	87.8	1239	232.6	3775
10	6.93	133.58	0.46	7	165.7	2251	541.4	8044
11	23.34	414.23	2.25	56	568.0	9867	1707.9	29759
12	305.64	15.0 %	35.60	711	2616.1	16785	8692.9	56342

To find out more about the reasons for the long runtimes, we also experimented with fuzzy automorphism detection. For this, we used the exact automorphism detection approach presented in Chapter 2 with the modifications introduced in Sect. 6.1. As for reflection detection, we assigned a penalty of one to each fixed node. The algorithm works like fuzzy symmetry detection without separating any orbit length constraints. The results are displayed in Table 6.4.

Table 6.4: Results for fuzzy automorphism detection

	runtime		opttime		#subprobs		#LPs	
n	avg	max	avg	max	avg	max	avg	max
8	0.14	2.22	0.01	1	9.7	85	30.9	298
9	1.75	10.97	0.38	10	55.6	299	184.4	1001
10	7.40	93.28	2.30	47	144.1	1397	489.0	5106
11	176.19	1.0 %	42.25	1129	1516.1	16613	5256.9	59074
12	600.12	15.0 %	89.15	1072	3461.8	16129	12283.6	57680

The runtimes for fuzzy automorphism detection are even worse than those for fuzzy rotation detection. Thus, the relatively poor performance of the fuzzy symmetry detection algorithm does not stem from the additional difficulty of symmetry detection compared with automorphism detection, but from allowing fuzziness. An important reason is the size of the LPs. We do not only have to deal with edge variables additionally, we also have much more mapping variables in general because we cannot use automorphism partitionings to delete some of them. Moreover, our classes of fuzzy homomorphism constraints seem to perform much worse than exact homomorphism constraints.

To investigate the effect of the number of variables more closely, we finally evaluated a variant of the fuzzy reflection detection algorithm: we only allowed to delete edges but not to create them. In the corresponding ILP, we only needed an edge variable for each adjacent node-pair in the original graph. By construction of our test set, we could thus save about 3/4 of the edge variables. Results are displayed in Table 6.5.

Table 6.5: Results for fuzzy reflection detection, deleting edges only

	runtime		opttime		#subprobs		#LPs	
n	avg	max	avg	max	avg	max	avg	max
8	0.02	0.14	0.00	0	3.1	15	9.0	38
9	0.05	0.24	0.00	0	5.9	29	17.5	59
10	0.10	0.35	0.00	0	8.6	33	24.0	70
11	0.35	1.53	0.05	1	16.7	53	48.7	156
12	0.84	4.39	0.06	2	28.8	111	85.0	327
13	4.68	20.13	1.23	11	85.5	273	260.3	846
14	8.19	26.94	1.22	8	122.7	371	380.1	1112
15	51.86	161.30	11.95	124	405.5	1149	1280.5	3623
16	79.16	219.41	10.05	107	554.9	1419	1791.2	4521

These figures show that a smaller number of edge variables can decrease runtime significantly. Compared with the results given in Table 6.1, we observed an improvement of average runtime by a factor of more than 14 for $n = 12$.

In summary, our branch & cut-algorithm for fuzzy symmetry detection only works for small graphs yet. Its performance is much better for nearly symmetric graphs than for random graphs. We are optimistic that the runtimes can be improved by future work, mainly by finding new classes of facets of the fuzzy symmetry polytopes.

Conclusion

We presented an integer programming approach for symmetry detection in general graphs and reported experimental results for a branch & cut-implementation of this approach. In terms of runtime, our approach cannot yet compete with the group-theoretic approach presented recently by Abelson et al. [2]. However, our approach is very flexible and allows arbitrary objective functions as well as restrictions by additional linear constraints.

Furthermore, our results show that integer programming methods can be applied successfully to problems related to group theory, in particular if group-theoretic arguments are exploited and integrated cleverly. Such a hybrid approach is most useful for problems on the borderline between group theory and combinatorial optimization, e.g., if the group structure is combined with linear constraints or objective functions.

As an example, we would like to point at a possible generalization of our algorithm presented in Chapter 2. In fact, all the results given there are valid for arbitrary permutation groups specified by a set of generators in place of the automorphism group of a graph: we can still use the assignment polytope together with some kind of homomorphism constraints then. Recall that these constraints are determined by the automorphism partitionings of the graph, which in turn can be computed from a set of generators of the automorphism group. It is easy to see that the same construction works for arbitrary permutation groups and that the crucial Theorem 2.2 remains valid. Separation and primal heuristics can be carried over without change. The only part of the branch & cut-algorithm that has to be changed is the feasibility check for integer solutions: if we do not have a problem-specific feasibility algorithm at our disposal, we need a general membership test for permutation groups given by generators. Such a test can be performed in polynomial time by Hoffmann [38]. In summary, our approach can be regarded as a general method for linear optimization over permutation groups given by group generators.

Unfortunately, the symmetries of a graph do not form a group with respect to composition in general, and the same is true for rotations or reflections. So we cannot use the permutation group approach for symmetry detection. Instead, we have to add further constraints to the integer linear program modeling automorphisms, leading to problems that are harder to solve than permutation group problems, as the runtime results in Chapter 5 show.

Since most graphs do not admit any non-trivial symmetry at all, fuzzy symmetry detection is a natural extension of exact symmetry detection. For Automatic Graph Drawing, this extension is very important, since nearly symmetric drawings can still reveal interesting structural properties of a graph. In Chapter 6, we presented an integer programming approach to fuzzy symmetry detection. However, even though we were able to discover large classes of cutting planes for the fuzzy symmetry polytopes, we saw that the practical performance of our branch & cut-algorithm is rather poor in general. In order to improve our approach, we have to identify more cutting planes or generalize the classes already known. This is the most important future challenge related to symmetry detection by branch & cut.

Finally, we come back to a problem that is closely related to reflection detection: the graph isomorphism problem. In fact, this problem is a special case of reflection detection in the following way: we can join the two graphs being tested for isomorphism and search for a reflection in the disjoint union. To avoid that nodes within the same original graph are matched, we just have to assign large weights to the corresponding mapping variables. By this, we are not only able to decide isomorphism but we can find an isomorphism minimizing any linear objective function. Proceeding from exact to fuzzy reflection detection, the graph isomorphism model becomes a model for the maximum common edge-induced subgraph problem: if we allow edge deletion but forbid edge creation, a reflection with a minimal number of deleted edges corresponds to a common subgraph with a maximal number of edges. By allowing node deletion and forbidding both edge deletion and edge creation, we can model the maximum node-induced common subgraph problem, too. Many well-known NP-hard problems like maximum clique can be regarded as special cases of the maximum edge-induced or node-induced common subgraph problem. Moreover, all weighted and fuzzy versions of these problems are special cases of the fuzzy reflection problem. In summary, fuzzy symmetry detection is a very general task. In view of this, its hardness is not surprising.

Bibliography

[1] ABACUS – A Branch-And-CUt System. `www.informatik.uni-koeln.de/abacus`.

[2] David Abelson, Seok-Hee Hong, and Donald E. Taylor. A group-theoretic method for drawing graphs symmetrically. In Goodrich and Kobourov [34], pages 86–97.

[3] László Babai, Dima Yu. Grigoryev, and David M. Mount. Isomorphism of graphs with bounded eigenvalue multiplicity. In *STOC 1982*, pages 310–324. ACM Press, 1982.

[4] Egon Balas. The asymmetric assignment problem and some new facets of the traveling salesman polytope on a directed graph. *SIAM Journal on Discrete Mathematics*, 2: 425–451, 1989.

[5] Egon Balas and Matteo Fischetti. A lifting procedure for the asymmetric traveling salesman polytope and a large class of new facets. *Mathematical Programming*, 58: 325–352, 1993.

[6] Michel L. Balinski and Andrew Russakoff. On the assignment polytope. *SIAM Review*, 16:516–525, 1974.

[7] Oliver Bastert. New ideas for canonically computing graph algebras. Technical Report TUM-M9803, Technische Universität München, Fakultät für Mathematik, 1998.

[8] András A. Benczúr and Ottilia Fülöp. Fast algorithms for even/odd minimum cuts and generalizations. In Mike Paterson, editor, *ESA 2000*, volume 1879 of *Lecture Notes in Computer Science*, pages 88–99. Springer-Verlag, 2000.

[9] Max Bense. *Einführung in die informationstheoretische Ästhetik*. Rowohlt, 1969.

[10] Norman L. Biggs and Arthur T. White. *Permutation Groups and Combinatorial Structures*, volume 33 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1979.

[11] Garrett Birkhoff. Tres observaciones sobre el algebra lineal. *Revista Facultad de Ciencias Exactas, Puras y Applicadas Universidad Nacional de Tucuman, Serie A (Matematicas y Fisica Teoretica)*, 5:147–151, 1946.

[12] George D. Birkhoff. *Aesthetic Measure*. Harvard University Press, 1933.

[13] Hans L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *Journal of Algorithms*, 11:631–643, 1990.

[14] Richard A. Brualdi and Peter M. Gibson. Convex polyhedra of doubly stochastic matrices I: Applications of the permanent function. *Journal of Combinatorial Theory*, 22(A):194–230, 1977.

[15] Richard A. Brualdi and Peter M. Gibson. Convex polyhedra of doubly stochastic matrices II: Graph of Ω_n. *Journal of Combinatorial Theory*, 22(B):175–198, 1977.

[16] Richard A. Brualdi and Peter M. Gibson. Convex polyhedra of doubly stochastic matrices III: Affine and combinatorial properties. *Journal of Combinatorial Theory*, 22(A):338–351, 1977.

[17] Christoph Buchheim and Seok-Hee Hong. Crossing minimization for symmetries. In Prosenjit Bose and Pat Morin, editors, *ISAAC 2002*, volume 2518 of *Lecture Notes in Computer Science*, pages 563–574. Springer-Verlag, 2002.

[18] Christoph Buchheim and Michael Jünger. Detecting symmetries by branch & cut. To appear in *Mathematical Programming.*

[19] Christoph Buchheim and Michael Jünger. Detecting symmetries by branch & cut. In Mutzel et al. [62], pages 178–188.

[20] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. A fast layout algorithm for k-level graphs. In Marks [58], pages 229–240.

[21] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. Improving Walker's algorithm to run in linear time. In Goodrich and Kobourov [34], pages 344–353.

[22] Hamish Carr and William Kocay. An algorithm for drawing a graph symmetrically. *Bulletin of the ICA*, 27:19–25, 1999.

[23] Ho-Lin Chen, Hsueh-I. Lu, and Hsu-Chun Yen. On maximum symmetric subgraphs. In Marks [58], pages 372–383.

[24] Jianer Chen. A linear-time algorithm for isomorphism of graphs of bounded average genus. *SIAM Journal of Discrete Mathematics*, 7:614–631, 1994.

[25] CPLEX 7.0. `www.ilog.com/products/cplex`.

[26] Hubert de Fraysseix. An heuristic for graph symmetry detection. In Kratochvíl [48], pages 276–285.

[27] Giuseppe Di Battista, editor. *Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, 1998. Springer-Verlag.

[28] Giuseppe Di Battista, Peter Eades, and Roberto Tamassia. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry Theory & Applications*, 4:235–282, 1994.

[29] Peter Eades and Xuemin Lin. Spring algorithms and symmetry. *Theoretical Computer Science*, 240(2):379–405, 2000.

[30] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17: 449–467, 1965.

[31] Max Fontet. A linear algorithm for testing isomorphism of planar graphs. In S. Michaelson and Robin Milner, editors, *ICALP 1976*, pages 411–424. Edinburgh University Press, 1976.

[32] Chris D. Godsil and Gordon Royle. *Algebraic Graph Theory.* Graduate Texts in Mathematics. Springer-Verlag, 2001.

[33] Michel X. Goemans and V. S. Ramakrishnan. Minimizing submodular functions over families of sets. *Combinatorica*, 15(4):499–513, 1995.

[34] Michael T. Goodrich and Stephen G. Kobourov, editors. *Graph Drawing 2002*, volume 2528 of *Lecture Notes in Computer Science*, 2002. Springer-Verlag.

[35] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization.* Springer-Verlag, 1988.

[36] Martin Grötschel and Manfred W. Padberg. Polyhedral theory. In Eugene L. Lawler, Jan K. Lenstra, A. H. G. Rinnooy Kan, and David B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, pages 251–305. John Wiley & Sons, 1985.

[37] Godfrey H. Hardy and Edward M. Wright. *An introduction to the theory of numbers.* Clarendon Press, 1938.

[38] Christoph M. Hoffmann. *Group-Theoretic Algorithms and Graph Isomorphism*, volume 136 of *Lecture Notes in Computer Science.* Springer-Verlag, 1982.

[39] Seok-Hee Hong and Peter Eades. Drawing planar graphs symmetrically II: Biconnected graphs. Technical Report CS-IVG-2001-01, University of Sydney, 2001.

[40] Seok-Hee Hong and Peter Eades. Drawing planar graphs symmetrically III: Oneconnected graphs. Technical Report CS-IVG-2001-02, University of Sydney, 2001.

[41] Seok-Hee Hong and Peter Eades. Drawing planar graphs symmetrically IV: Disconnected graphs. Technical Report CS-IVG-2001-03, University of Sydney, 2001.

[42] Seok-Hee Hong, Brendan McKay, and Peter Eades. Symmetric drawings of triconnected planar graphs. In *SODA 2002*, pages 356–365, 2002.

[43] John E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *STOC 1974*, pages 172–184. ACM Press, 1974.

[44] Michael Jünger and Denis Naddef, editors. *Computational Combinatorial Optimization – Optimal and Provably Near-Optimal Solutions*, volume 2241 of *Lecture Notes in Computer Science.* Springer-Verlag, 2001.

[45] Michael Jünger and Stefan Thienel. The ABACUS system for branch-and-cut-and-price-algorithms in integer programming and combinatorial optimization. *Software – Practice & Experience*, 30(11):1325–1352, 2000.

[46] Narendra Karmakar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[47] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs*, volume 2025 of *Lecture Notes in Computer Science.* Springer-Verlag, 2001.

[48] Jan Kratochvíl, editor. *Graph Drawing '99*, volume 1731 of *Lecture Notes in Computer Science*, 1999. Springer-Verlag.

[49] Richard E. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22:155–171, 1975.

[50] László Lovász and Michael D. Plummer. *Matching Theory*, volume 29 of *Annals of Discrete Mathematics*. North Holland, 1986.

[51] Anna Lubiw. Some NP-complete problems similar to graph isomorphism. *SIAM Journal on Computing*, 10(1):11–21, 1981.

[52] Magma Computational Algebra System. `magma.maths.usyd.edu.au/magma`.

[53] Joseph Manning. *Geometric Symmetry in Graphs*. PhD thesis, Purdue University, 1990.

[54] Joseph Manning. Computational complexity of geometric symmetry detection in graphs. In Naveed A. Sherwani, Elise de Doncker, and John A. Kapenga, editors, *The First Great Lakes Computer Science Conference*, volume 507 of *Lecture Notes in Computer Science*, pages 1–7. Springer-Verlag, 1991.

[55] Joseph Manning and Mikhail J. Atallah. Fast detection and display of symmetry in trees. *Congressus Numerantium*, 64:159–169, 1988.

[56] Joseph Manning and Mikhail J. Atallah. Fast detection and display of symmetry in outerplanar graphs. *Discrete Applied Mathematics*, 39(1):13–35, 1992.

[57] François Margot. Exploiting orbits in symmetric ILP. Preprint.

[58] Joe Marks, editor. *Graph Drawing 2000*, volume 1984 of *Lecture Notes in Computer Science*, 2001. Springer-Verlag.

[59] George E. Martin. *Transformation Geometry – An Introduction to Symmetry*. Springer-Verlag, 1982.

[60] Brendan D. McKay. `nauty` user's guide (version 1.5). Technical Report TR-CS-90-02, Computer Science Department, Australian National University, 1990.

[61] E. Mendelsohn. Every (finite) group is the group of automorphisms of a (finite) strongly regular graph. *Ars Combinatoria*, 6:75–86, 1978.

[62] Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors. *Graph Drawing 2001*, volume 2265 of *Lecture Notes in Computer Science*, 2001. Springer-Verlag.

[63] Mikhail E. Muzychuk and Gottfried Tinhofer. Recognizing circulant graphs of prime order in polynomial time. Technical Report TUM-M9703, Technische Universität München, Fakultät für Mathematik, 1997.

[64] Mikhail E. Muzychuk and Gottfried Tinhofer. Recognizing circulant graphs in polynomial time: An application of association schemes. *Electronic Journal of Combinatorics*, 8(1), 2001.

[65] Stephen North, editor. *Graph Drawing '96*, volume 1190 of *Lecture Notes in Computer Science*, 1996. Springer-Verlag.

[66] Manfred Padberg and M. Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7:67–80, 1982.

[67] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In Di Battista [27], pages 248–261.

[68] Ronald C. Read and Derek G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1977.

[69] Remko Scha and Rens Bod. Computationele esthetica. *Informatie en Informatiebeleid*, 11(1):54–63, 1993.

[70] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[71] Alexander Schrijver. *Combinatorial Optimization – Polyhedra and Efficiency.* Springer-Verlag, 2003.

[72] Sue H. Whitesides, editor. *Graph Drawing '98*, volume 1547 of *Lecture Notes in Computer Science*, 1998. Springer-Verlag.

[73] Helmut Wielandt. *Finite Permutation Groups.* Academic Press, 1964.

Erklärung

Ich versichere, dass ich die von mir vorgelegte Dissertation selbstständig angefertigt, die benutzten Quellen und Hilfsmittel vollständig angegeben und die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem Einzelfall als Entlehnung kenntlich gemacht habe; dass diese Dissertation noch keiner anderen Fakultät oder Universität zur Prüfung vorgelegen hat; dass sie – abgesehen von unten angegebenen Teilpublikationen – noch nicht veröffentlicht worden ist sowie, dass ich eine solche Veröffentlichung vor Abschluss des Promotionsverfahrens nicht vornehmen werde. Die Bestimmungen der Promotionsordnung sind mir bekannt. Die von mir vorgelegte Dissertation ist von Prof. Dr. Michael Jünger betreut worden.

Köln, 02.04.2003

Teilpublikationen

Christoph Buchheim and Michael Jünger. Detecting symmetries by branch & cut. To appear in *Mathematical Programming*.

Christoph Buchheim and Michael Jünger. Detecting symmetries by branch & cut. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *GD 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 178–188. Springer-Verlag, 2001.