Julius-Maximilians-Universität Würzburg
Institut für Mathematik
Lehrstuhl für Geometrie

Bachelor Thesis

Graph Isomorphism

Nils Wisiol

submitted on May 27th, 2015

supervisor:
Prof. Dr. Nils Rosehr

Abstract

While in general it is not known whether there is a polynomial time algorithm to decide whether two given graphs are isomorphic, there are polynomial-time algorithms for certain subsets of graphs, including but not limited to planar graphs and graphs with bounded valence.

In this thesis, we will give a brief introduction on the Graph Isomorphism Problem and its relations to complexity theory. We show that permutation groups can, despite their large sizes, stored in digital computers in a succinct way. This raises questions about our ability to answer important questions about these permutation groups with algorithms in polynomial time. We present some polynomial-time algorithms that can determine basic facts about succinctly stored groups. After this, we proof that graphs with valence bounded by 3 can be checked for isomorphism in polynomial time, following the proof given by Luks [Luk82].

Zusammenfassung

Es ist unbekannt, ob es einen Polynomialzeitalgorithmus gibt, der Isomorphie für zwei beliebige Graphen feststellen kann. Wir kennen jedoch Polynomialzeitalgorithmen für bestimmte Klassen von Graphen, beispielsweise planare Graphen und Graphen mit beschränkter Valenz.

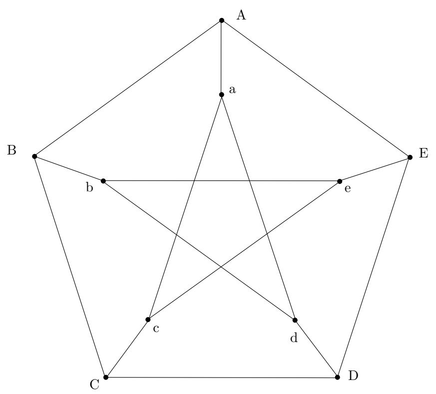
In der vorliegenden Arbeit geben wir eine kurze Einführung in das Graphen-Isomorphie-Problem und seine Verbindung zur Komplexitätstheorie. Wir zeigen dass Permutationsgruppen, obwohl von großer Ordnung, in kurzer Darstellung in digitalen Computern gespeichert werden können. Das wirft die Frage auf, ob wir wichtige Eigenschaften dieser Gruppen in Polynomialzeit algorithmisch festgestellt werden können. Wir führen einige Algorithmen auf, die einige dieser Fragen in Polynomialzeit beantworten können. Anschließend zeigen wir, basierend auf einem Beweis von Luks [Luk82], dass Graphen, deren Valenz durch 3 beschränkt ist, in Polynomialzeit auf Isomorphie überprüft werden können.

Contents

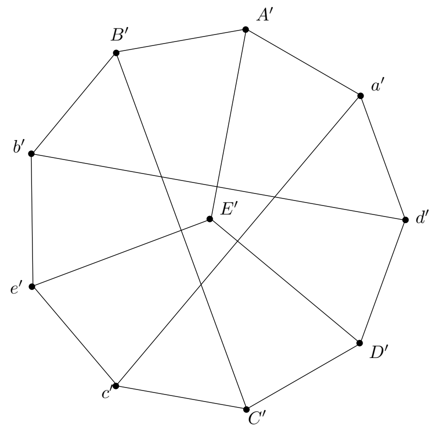
1 Introduction

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, deciding whether they are isomorphic is deciding whether these graphs are essentially the same. More precisely, it is deciding whether there is a bijection $\sigma : V_1 \to V_2$ such that $(v, w) \in E_1$ if and only if $(\sigma(v), \sigma(w)) \in E_2$. A bijection that satisfies this constraint is called a *graph isomorphism from G_1 to G_2*.

Example 1. To demonstrate graph isomorphism, we present two different drawings of the famous *Petersen Graph*.



(a) The Petersen Graph represented as an pentagon surrounded by another pentagon.



(b) A representation based on a polygon with nine edges.

Figure 1.1: Two different drawings of the Petersen Graph.

Although the two drawings look different, we will prove that they actually represent the same graph. Let $G_1 = (V_1, E_1)$ be the graph represented by Figure 1.1a and $G_2 = (V_2, E_2)$ be the graph represented by Figure 1.1b. To prove G_1 and G_2 are isomorphic we will define a one-to-one mapping σ and show that it is an isomorphism. Before having a close look at σ, notice that both G_1 and G_2 have the same number of edges and vertices. Moreover, both graphs have only vertices with degree exactly three. Thus, they meet some necessary but in general not sufficient criteria for being isomorphic.

We have $V_1 = \{a, b, c, d, e, A, B, C, D, E\}$ and $V_2 = \{x' \mid x \in V_1\}$. Let $\sigma : V_1 \to V_2$ be defined by $x \mapsto \sigma(x) := x'$, which is a bijection. Notice that σ preserves paths through the graph, which is another necessary condition for being an isomorphism. That is, the closed path (A, B, C, D, E) becomes (A', B', C', D', E'), which is still a circle. Similar, the closed path (a, c, e, b, d), the inner star in G_1, becomes the closed path (a', c', e', b', d').

So intuitively it is clear σ is an isomorphism. For a formal proof, we look at the adjacency matrix of G_1 and $\sigma(G_1)$. Since the graph is undirected, the adjacency matrix is symmetric. For convenience, we state only the upper half. One can think of it in three partitions: the path (A, B, C, D, E), the inner pentagon and the edges going from x to X. Comparing the adjacency matrix 1.2b with the drawing 1.1b it turns out that $\sigma(G_1) = G_2$, and thus σ being an isomorphism for G_1 and G_2. Therefore, G_1 and G_2 are isomorphic.

To decide whether or not two graphs are isomorphic is known as the *Graph Isomorphism Problem*. It belongs to the problems in NP, since one can guess and confirm mappings σ in polynomial time, but it is not known to be NP-complete. Moreover, it is known that the Graph Isomorphism Problem can only be NP-complete if the polynomial hierarchy collapses[1]

[1]The polynomial hierarchy generalizes the complexity classes P, NP and coNP to a hierarchy of increasingly complex classes. It is believed that higher classes of the hierarchy are honest supersets of their respective counterparts in lower levels.

	a	b	c	d	e	A	B	C	D	E
a			1	1		1				
b				1	1		1			
c					1			1		
d									1	
e										1
A							1			1
B								1		
C									1	
D										1
E										

(a) The adjacency matrix of G_1.

	a'	b'	c'	d'	e'	A'	B'	C'	D'	E'
a'			1	1		1				
b'				1	1		1			
c'					1			1		
d'									1	
e'										1
A'							1			1
B'								1		
C'									1	
D'										1
E'										

(b) The adjacency matrix of $\sigma(G_1)$.

Figure 1.2: Adjacency matrix of G_1 and $\sigma(G_1)$. Comparing to Figure 1.1b, it turns out that the adjacency matrix of $\sigma(G_1)$ represents the graph shown.

and it is thus believed not to be NP-complete. As no polynomial time algorithm is known for the general case as well, the Graph Isomorphism Problem is thought to be an intermediate problem in NP $-$ P [HS01].

Being thought to be in between P and NP-complete, the Graph Isomorphism Problem is related to the Integer Factorization Problem, which is thought to be an intermediate problem as well. In fact, it was shown that Graph Isomorphism and Integer Factorization can be reduced to the problem of counting automorphisms for rings. Kayal and Saxena used this to show that both problems cannot be NP-complete unless the polynomial hierarchy collapses [KS05].

As opposed to the general Graph Isomorphism Problem, it is known that the isomorphism problem is solvable in polynomial time for many classes of graphs, including planar graphs [HT74]. In this thesis, we will focus on graphs with bounded valence and demonstrate how the Graph Isomorphism Problem can be solved in polynomial time. We follow a proof due to Luks [Luk82].

2 Preliminaries

2.1 Algebra

Let G be a group, and $S \subseteq G$ be a subset of this group. Let $\langle S \rangle$ be the smallest subgroup of G that contains S. If $\langle S \rangle = G$, S is called a *generating set* for G. If $\langle S \rangle = G$ and for all $S' \subsetneq S$ we have$\langle S' \rangle \neq S$ then S is called a *minimal generating set*.

Let G be a group, H be a subgroup and $g \in G$. Then $gH = \{gh : h \in H\}$ is called the *left coset of H in G with respect to g*, and $Hg = \{hg : h \in H\}$ is called the *right coset of H in G with respect to g*. Cosets can also be defined as equivalence classes of the relation $\sim$ defined by $x \sim y$ if and only if $x^{-1}y \in H$ and $yx^{-1} \in H$ respectively. Therefore, the left (right) cosets of G form a partition of G [Bra].

Let G be a group and $N \subset G$ be a subgroup. We call N *normal*, if and only if $gN = Ng$ holds for all $g \in G$. For normal subgroups N of G, we write $N \lhd G$.

Let G be a group. G is called *simple* if and only if the only normal subgroups are the trivial group and G itself.

Let G be a group, and let H be a subgroup of G. In this case we also write $H \leq G$. We define the *quotient H modulo G* as the set of all left cosets of H in G, $G/H = \{gH : g \in G\}$. If H is a normal subgroup, we usually write $N = H$, and G/N along with the product of subsets forms an algebraic group.

Let G be a group, and let $H \leq G$. The cardinality of cosets of H in G is called the *index of the subgroup H in G*, written as $|G : H|$. That is, $|G : H| = |\{gH : g \in G\}| = |\{Hg : g \in G\}|$. Since the quotient group is the set of cosets, we obtain for a normal subgroup $N \lhd G$ that $|G : N| = |G/N|$.

We call a finite group G a *2-group*, if every element has a power of 2 as its order.

If there is a homomorphism $G \to \operatorname{Sym} B$, we define the action of G on as follows. The homomorphism yields a permutation of B for every element in G. The set $\{\sigma(b) \mid \sigma \in G\}$ for an element $b \in B$ is called the *G-orbit of b*. The group G acts *transitively on B* if B is a G-orbit. The homomorphism is called *action of G on B*.

For a group G transitively acting on a set A, we define a *G-block* as a non-empty subset B of A for which any action σ induced by G either stabilizes B, that is $\sigma(B) = B$, or moves B completely, that is $\sigma(B) \cap B = \emptyset$. We call the set $\{\sigma(B) \mid \sigma \in G\}$ a G-block system in A. For any $b \in B$, the set $\{b\}$ is a block. Therefore, we call a G-block system *minimal* and the action of G *primitive* if there are no G-blocks of size larger than one.

Lemma 2. *Let P be a transitive p-subgroup of $\operatorname{Sym} A$ with $|A| > 1$. Then any minimal p-block system consists of exactly p blocks. Furthermore, the subgroup P' which stabilizes all of the blocks has index p in P. [Luk82]*

Lemma 3. *Let G and H be groups, $I \subseteq G$, and $f : G \to H$ be a group homomorphism. If K is a generating set for $\operatorname{Ker} f$ and $f(I)$ is a generating set for $\operatorname{Im} f$, then $K \cup I$ generates G.*

Proof. Choose an arbitrary $g \in G$. The element $f(g)$ is a member of $\operatorname{Im} f$ and thus has a representation $f(g) = \prod f(i_k)^{\alpha_k}$ for some $i_k \in I$, $\alpha_k \in \{-1, 1\}$, $0 \leq k \leq m$. For the sake of simplicity, instead of $\prod_{k=0}^{m}$, we just write $\prod$. Let $h = g^{-1} \prod i_k^{\alpha_k}$. Then $f(h) = f(g^{-1}) f(\prod i_k^{\alpha_k}) = f(g)^{-1} \prod f(i_k)^{\alpha_k} = f(g)^{-1} f(g) = 1$ and therefore, we have $h, h^{-1} \in \operatorname{Ker} f$. From the definition of h we can derive $g = \prod i_k^{\alpha_k} \cdot h^{-1}$ and thus g can be generated from I and K. $\square$

2.2 Computational Complexity

For this thesis, we assume familiarity with the basic notions of Computational Complexity. For the reader's convenience, we review a couple of the most relevant definitions. The notions presented here are based on the text book of Homer and Selman [HS01].

2.2.1 Decision Problems

We define a *decision problem* to be a partitioning of the set of all strings into two sets, the set of yes-instances, and the set of no-instances. Usually, we write a decision problem just as the set of yes-instances. These sets are also called *language*.

Let T be a function defined on the natural numbers. We say a Turing machine M is $T(n)$ *time-bound*, if for every input of length n, it holds after at most $T(n)$ computational steps. We define $\mathrm{DTIME}(T(n))$ to be the collection of all languages that can be accepted by a Turing machine within a $T(n)$ time bound.

We define P to be the set of all languages that can be accepted by a Turing machine in polynomial time,

$$\mathrm{P} = \bigcup \{\mathrm{DTIME}(n^k) \mid k \geq 1\}.$$

A problem can be *decided in polynomial time* if its language (that is, the set of all words for which the answer to the problem is "yes") resides in P.

2.2.2 Function Problems

We define a *function problem* to compute a certain function f. This is a generalization of a decision problem: the latter can be modeled as a function problem that just computes the characteristic function χ_L of a language L defined by

$$\chi_L(x) = \begin{cases} 0 & (x \notin L), \\ 1 & (x \in L). \end{cases}$$

Following Krentel, we call a Turing machine *metric* if it writes a number on its output tape before it halts [Kre88]. A metric Turing machine *solves* the function problem of a function f if it writes $f(x)$ on its output tape for any input x. If a Turing machine that solves a function problem does so with a polynomial time-bound, we say the function problem can be *solved in polynomial time*.

2.2.3 Reductions

We define a *oracle Turing machine with oracle O* to be a Turing machine that has the additional capability to determine the truth value of $x \in O$ in just one computational step, given that x is written on one of the tapes of the machine.

In order to compare the complexity of different problems, we introduce reductions. For any two decision problems A and B we say, A is Turing-reducible to B if there is an oracle Turing machine with oracle B that can decide A in polynomial time, written as $A \leq B$. We can think of this notion as "B is at most polynomial-time more complex than A". Notice, since one query to the oracle takes one computational step as well, the number of oracle queries is limited by a polynomial.

A Turing machine that has the additional capability of computing $f(x)$ in $|f(x)|$ computational steps is called a *metric oracle Turing machine with oracle f*.

For comparison of functional problems f and g we define $f \leq g$ if there is a metric oracle Turing machine with oracle g that can compute f in polynomial time.

Using the characteristic function χ of a decision problem, we can write decision problems as function problems and apply the reduction of function problems to decision problems as well.

The reductions defined above are usually called Turing reductions. There exist many more reductions of different flavors, however for the purpose of this thesis the Turing reduction will suffice.

2.3 Graphs

An ordered pair of two sets $X = (V, E)$ is called an *(undirected simple) graph*, if E is a subset of the set of all 2-sets of V. (Notice that this definition does not include loops, that is edges connected a $v \in V$ with itself.) Elements of V are called *vertices* or *nodes* of X, members of E are called *edges* of X. By $V(X)$ and $E(X)$, we refer to the set of nodes of any graph X and the set of edges of X respectively. Although members of E are sets by definition, we often write vw or (v, w) instead of $\{v, w\}$ for the sake of simplicity. As opposed to directed graphs, we have $vw \in E$ if and only if $wv \in E$ for any graph in this thesis. Moreover, all graphs in this thesis are simple, that is, they have no edges vv for any $v \in V$. We only consider finite graphs in this thesis, as we use $|E|$ and $|V|$ to define the input length for algorithms.

A list of edges $(sv_1, v_1v_2, ..., v_{n-1}v_n, v_nt)$ of a graph X is called a *path from s to t*. A graph is called *connected*, if for any pair of nodes $v, w \in V(X)$ and $v \neq w$, there is a path in X from v to w. Otherwise, X is called *disconnected*.

For any node $v \in V(X)$ of a (simple) graph X, we define the *degree of v* to be the number of edges adjacent to v, $\deg v = |\{e \in E \,|\, v \in e\}|$. A graph X has *valence bounded by k*, if for any $v \in V(X)$, $\deg v \leq k$. Graphs with valence bounded by three are called *trivalent*.

For any two graphs $X_1 = (V_1, E_1)$ and $X_2 = (V_2, E_2)$, we call a bijective function $\sigma : V_1 \to V_2$ a graph *isomorphism*, if σ preserves edge relations, that is, for any $v, w \in V_1$ it holds that $vw \in E_1 \iff \sigma(v)\sigma(w) \in E_2$. For an isomorphism σ we can also write $\sigma : X_1 \to X_2$. If there is an isomorphism, X_1 and X_2 are called *isomorphic*.

Let X be a graph. Any graph isomorphism $\sigma : X \to X$ that maps X to itself is called a graph *automorphism*. The identity function is always an automorphism. The set of all automorphisms for a graph X together with composition of functions is called the *automorphism group* $\operatorname{Aut} X$. For any $e \in E(X)$, we define $\operatorname{Aut}_e(X)$ to be the set of automorphisms of X that fix e, that is, for $\sigma \in \operatorname{Aut}_e(X)$ with $e = vw$ we have $\sigma(\{v, w\}) = \{v, w\}$.

Definition 4. The *Graph Isomorphism Problem (for connected graphs)* is to decide whether two (connected) graphs are isomorphic.

Lemma 5. *The Graph Isomorphism Problem is polynomial time reducible to the Graph Isomorphism Problem for connected graphs.*

Proof. For any given graph, we can compute the number of connected components in polynomial time using a transitive hull algorithm similar to Algorithm 1.

Let $X_1 = (V_1, E_1)$, $X_2 = (V_2, E_2)$ be two possibly disconnected graphs. We assume X_1 and X_2 consist of an equal number of nodes, if they do not, they are not isomorphic and we are done. We compute all connected components for X_1 and X_2. If both graphs are connected, we have a Graph Isomorphism Problem for connected graphs and we are done. If the graphs have different number of connected components (for instance, one graph is connected, and the other is not), then they are not isomorphic and we are done.

Now assume both graphs are not connected and have an equal number of connected components.

For any given graph $X = (V, E)$, let $\widetilde{X} = (\widetilde{V}, \widetilde{E})$ be the graph with one additional node x_X that is connected to every node in X. Using this operation, $\widetilde{X}_1$ and $\widetilde{X}_2$ are both connected.

Computing $\widetilde{X}$ takes polynomial time, since only two nodes and $|V_1| + |V_2|$ edges have to be added. We will see that X_1 and X_2 are isomorphic if and only if $\widetilde{X}_1$ and $\widetilde{X}_2$ are isomorphic, hence we can use the algorithm for Graph Isomorphism for connected graphs to decide isomorphism for X_1 and X_2.

Notice that, since X is not connected, X does not have a node that is connected to every other node in X. Therefore, there is no node in X with degree $|V| - 1$. From construction we know that $|\widetilde{V}| = |V| + 1$ and the degree of the new node $x_X \in \widetilde{X}$ is $|\widetilde{V}| - 1 = |V|$. Hence x_X is the only node in $\widetilde{X}$ of degree $|\widetilde{V}| - 1$.

To see the equivalence, let $\widetilde{X}_1$ and $\widetilde{X}_2$ be isomorphic with $\sigma : \widetilde{X}_1 \to \widetilde{X}_2$ being an isomorphism. Since σ preserves node degree, σ maps x_{X_1} to x_{X_2}, both nodes being the only ones in their graph having node degree $|\widetilde{V}_1| - 1 = |\widetilde{V}_2| - 1$. Thus, the restriction of σ to X_1 is a graph isomorphism $X_1 \to X_2$.

Conversely, if X_1 and X_2 are isomorphic with an isomorphism σ we can compute $\widetilde{X}_1$ and $\widetilde{X}_2$ and extend σ to map x_{X_1} to x_{X_2} to get an isomorphism of $\widetilde{X}_1$ and $\widetilde{X}_2$. □

With justification given by Lemma 5, we assume from now on connected graphs.

2.4 On the Size of Group Representations

Lemma 6. *Any group G has a generating set of cardinality $\log_2 |G|$ or less.*

Proof. Let $\widetilde{G} = \{g_1, ..., g_m\}$ be a minimal generating set for $G = \langle \widetilde{G} \rangle$ and define $G_n = \langle g_1, ..., g_n \rangle$ for $n = 1, ..., m$. By minimality, $e \notin \widetilde{G}$. Assume $g_{n+1} \in G_n$, then $\widetilde{G} \setminus \{g_{n+1}\}$ is still a generating set for G. Therefore, $g_{n+1} \notin G_n$ and G_{n+1} has at least two disjoint cosets, eG_n and $g_{n+1}G_n$. Therefore, $|G_{n+1}| \geq 2|G_n|$. By induction, we obtain $|G| = |G_m| \geq 2^m$. Hence, $m \leq \log_2 |G|$. □

3 Basic Polynomial-Time Graph Operations

To tackle the Graph Automorphism Problem for graphs with valence bounded by three, we need some basic graph operations for permutation groups that can be computed in polynomial time. Due to the succinct notation of groups proven in Lemma 6, which results in a short input length for algorithms, it is not obvious that these computations can be carried out in polynomial time.

3.1 Determine the G-orbits

Let the group $G \subseteq \operatorname{Sym} A$ be generated by the generators $g_1, ..., g_m$. We can use a the transitive hull algorithm shown in Algorithm 1 to compute the G-orbit of any element $a \in A$ [Luk82]. More specifically, we start with $H_a = \{a\}$ and keep adding the result of the operation $g_k(h)$, with $h \in H_a$ and $k = 1, ..., m$ to H_a until all operations do not yield new results anymore.

Algorithm 1 Algorithm to compute the G-orbits of all $a \in A$

```
input: set A, generators g_1, ..., g_m of group G ⊆ Sym A
output: collection of G-orbits H_a for each element a ∈ A

for all a ∈ A do
  H_a ← {a}
  repeat
    H_a ← H_a ∪ {g_k(h) | k = 1, ..., m and h ∈ H_a}
  until no new elements were added
end for
return {H_a | a ∈ A}
```

We can illustrate the transitive hull algorithm approach for a fixed member $a \in A$ with a graph that contains a node for each member of H_a and an edge going from every h to $g_k(h)$. Since all element eventually decent from a, the graph is connected. Since $|A|$ is an upper bound for the size of the G-orbit of a, and each node has at most m outgoing edges, one for each generator, we can conclude that the algorithm terminates within polynomial time.

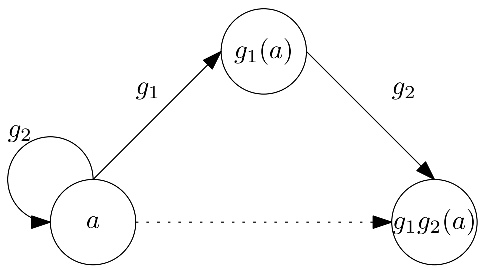


Figure 3.1: Example of the transitive hull algorithm to determine the G-orbit of a.

Example 7. Assume $A = \{a, b, c\}$ and $G = \langle g_1, g_2 \rangle$ with g_1 transposing a and b, and g_2 transposing b and c. $\operatorname{Sym} A$ has $|A|! = 3! = 6$ elements, and we can find six different members of $\langle g_1, g_2 \rangle$. Therefore, $G = \operatorname{Sym} A$. From this we can conclude that the G-orbit of a is $\{a, b, c\}$. However, neither g_1 nor g_2 map a to c directly. We can only derive c from a second iteration: $g_2(g_1(a)) = g_2(b) = c$. The corresponding graph is illustrated in Figure 3.1.

We formalize the result of this section in

Theorem 8. *Given a set A and generators $g_1, ..., g_m$ of $G \subseteq \operatorname{Sym} A$, we can compute the G-orbit of all $a \in A$ in polynomial time.*

3.2 Determine the Order of G

In order to determine the order of G from a set of generators $\{g_1, ..., g_m\}$, we write $A = \{a_1, ..., a_n\}$ and define a chain of subgroups G_i, each G_i containing only elements that fix elements a_1,...,a_i [FHL80]. This yields

$$\{1\} = G_n \subseteq G_{n-1} \subseteq ... \subseteq G_1 \subseteq G_0 = G.$$

Consider the quotients G_i/G_{i+1} in this chain. By definition, the quotient G_i/G_{i+1} is the collection of all cosets of G_{i+1} in G_i [FHL80]. The cosets can be characterized as equivalence classes of the equivalence relation $\sigma \equiv \tau \iff \sigma^{-1}\tau \in G_{i+1}$, $\sigma, \tau \in G_i$. In other words, σ and τ both fix the elements $a_1, ..., a_i$. They belong to the same equivalence class (that is, the same coset) if $\sigma^{-1}\tau$ fixes elements $a_1, ..., a_{i+1}$.

If σ and τ are in the same class, then $\sigma(a_{i+1}) = \tau(a_{i+1})$. To see this, assume $\sigma^{-1}(\tau(a_{i+1})) = a_{i+1}$. In the case that $\tau(a_{i+1}) = a_{i+1}$, we have $\sigma(a_{i+1}) = a_{i+1}$. In the other case, $\tau(a_{i+1}) = a_k$ for a $k \leq i$, we have $\sigma^{-1}(a_k) = a_{i+1}$ and thus $\sigma(a_{i+1}) = a_k = \tau(a_{i+1})$. We obtain the following lemma.

Lemma 9. *The quotient G_i/G_{i+1} consists of exactly the classes of the equivalence relation $\sigma \equiv \tau \iff \sigma(a_{i+1}) = \tau(a_{i+1})$, $\sigma, \tau \in G_i$.*

Thus, with the chain $G_{n-1} \subseteq ... \subseteq G_0$ we can represent every element $\sigma \in G$ as a product of members σ_i of quotient G_i/G_{i+1} in the chain, $\sigma = \sigma_{n-1}\sigma_{n-2} \cdots \sigma_1\sigma_0$. In this representation, σ_0 moves a_1 to the right place, then σ_1 fixes a_1 and moves a_2 to the right place, and so on. We call this representation *canonical*.

By Lagrange's theorem, we know that $|G_i| = [G_i : G_{i+1}]|G_{i+1}|$, and thus we can write

$$|G| = [G_0 : G_1][G_1 : G_2] \cdots [G_{n-1} : G_n].$$

In order to determine all $[G_i : G_{i+1}]$, we are going to compute a table that holds, once we are done, exactly one member of every coset in the subgroup chain. By Lemma 9, we know that for each step in the subgroup chain, we have at most n different subgroups. With the chain having n members, this results in a $n \times n$ table. In the i-th row we are going to store coset representatives of G_i/G_{i+1}, and the permutation in the j-th position fixes letters $1, ..., i-1$ and maps a_i to a_j. We call the table T, and the element in the i-th row and j-th column $T_{i,j}$. To fill the table, we use the following routine *sift*.

Algorithm 2 *Sift*: This algorithm fills table T based on a given element α.

input: an element α of G
No output, however contents and modifications of T are stored permanently.

for $i = 0...n-2$ **do**
  if there is a σ in the i-th row of T and $\sigma(a_{i+1}) = \alpha(a_{i+1})$ **then**
    // by Lemma 9, α and σ belong to the same coset
    $\alpha \leftarrow \gamma^{-1}\alpha$
  else
    // α represents a coset of G_i/G_{i+1} that we don't have in T yet
    $T_{i,j} = \alpha$ for the appropriate j
    return
  end if
end for

To get an idea of how *sift* works, assume element α written down as generated by the chain subgroups described above, $\alpha = \sigma_{n-1}\sigma_{n-2} \cdots \sigma_1\sigma_0$ with $\sigma_i \in G_i/G_{i+1}$. *sift* now works top-down through the table, checking in each row, if α represents an already known coset. If so, we remove the portion that belongs to the known coset and continue with the next row. (Another way to justify $\alpha \leftarrow \sigma^{-1}\alpha$ is that in the next row, we only consider permutations that fix elements $a_1, ..., a_{i+1}$.) If α represents a coset that we do not have in our table yet, we add it to the correct position and terminate *sift*.

Lemma 10. *With the definitions from above, T is complete after calling* sift *for all generators of G and calling* sift *for the product xy for all pairs (x, y) in T.*

Proof. Let $g \in G$. We can write g as product of generators $g_1, ..., g_m$ of G. In this product, write each generator as it's canonical product. We obtain

$$\begin{aligned} g &= g_1^{\alpha_1} g_2^{\alpha_2} \cdots g_m^{\alpha_m} \\ &= (\sigma_{n-1}^{(1)} \sigma_{n-2}^{(1)} \cdots \sigma_0^{(1)})^{\alpha_1} (\sigma_{n-1}^{(2)} \sigma_{n-2}^{(2)} \cdots \sigma_0^{(2)})^{\alpha_2} \cdots (\sigma_{n-1}^{(m)} \sigma_{n-2}^{(m)} \cdots \sigma_0^{(m)})^{\alpha_m} \end{aligned}$$

with $\alpha_i \in \{-1, 0, 1\}$ and $1 \leq i \leq m$. In order to obtain g in canonical form, we can use the canonical representation of xy for any x, y in the representation of g that are in wrong order. $\square$

Being able to compute the complete table in polynomial time enables us to compute the order of G in polynomial time.

Theorem 11. *Given a set of generators for a subgroup G of* $\mathrm{Sym}\, A$ *with $A = \{a_1, ..., a_n\}$, one can determine the order of $|G|$ in polynomial-time.*

Proof. Consider the following Algorithm 3.

Algorithm 3

input: generators $\{g_1, ..., g_m\}$ that generate permutation group G
output: $|G|$

sift all generators g_k
for each pair (x, y) in table T **do**
  sift xy
end for
return the product of the number of cosets in each row

By Lemma 10, after sifting all generators and products of pairs, the table is complete. Based on the number of cosets for each step in the subgroup chain, we can compute the total size of G with Lagrange's theorem.

Since *sift* works in polynomial time, and the number of elements in table T is polynomially bounded, the procedure completes in polynomial time. $\square$

3.3 Determine a Subgroup that Stabilizes

For a given subgroup H of G, the chain of subgroups from the previous chapter can be altered to

$$\{1\} = H_n \subseteq H_{n-1} \subseteq ... \subseteq H_1 \subseteq H \subseteq G,$$

in order to use Algorithm 3 to compute generators for this subgroup. If there is a polynomial-time membership test available, and the group has polynomial index in G, analysis of the algorithm above shows that this process completes in polynomial time as well.

Lemma 12. *Given a set of generators for a subgroup G of* $\mathrm{Sym}\, A$*, we can, in polynomial time, determine generators for any subgroup H of G which is known to have polynomially bounded index in G and for which a polynomial-time membership test is available.*

Theorem 13. *If G acts transitively on B, we can determine a subgroup H and $\tau \in G$ such that $G = H \cup \tau H$ and H stabilizes given G-blocks B' and B''.*

Proof. For any given $\sigma \in \operatorname{Sym} A$, we can check membership of H by checking if σ stabilizes B' and B''. This is possible in polynomial time by computing $\sigma(B')$. We write $G_{(i)}$ for the subgroup of G that stabilizes the first i blocks. The subgroup H as polynomial index in G, because $[G_{(i)} : G_{(i+1)}] \leq$ number of blocks $- i$. □

3.4 Determine a Minimal Block System

In this section we introduce Algorithm 4 due to Atkinson [Atk75]. For imprimitive groups, this algorithm is able to compute a minimal block system which contains a block that contains $\{1, \omega\}$ for any $\omega \in A$.

Algorithm 4 Polynomial-time algorithm to find the blocks of imprimitivity of a group from generating permutations.

---

input: non-empty set $A = \{1, ..., n\}$, set of generators $\{g_1, ..., g_m\}$ that generate permutation group $G \leq \operatorname{Sym} A$, element $\omega \in A$, $\omega \neq 1$
output: function f representing a block system that contains a smallest block $\supseteq \{1, \omega\}$

$C \leftarrow \{\omega\}$
$f(\alpha) \leftarrow \alpha$ for all $\alpha \in \Omega \setminus \{\omega\}$
$f(\omega) \leftarrow 1$
while $C \neq \emptyset$ **do**
  choose $\beta \in C$, delete β from C and $\alpha \leftarrow f(\beta)$
  for $j = 1...m-1$ **do**
    $\delta \leftarrow \beta g_j$
    if $f(\gamma) \neq f(\delta)$ **then**
      ensure $f(\delta) < f(\gamma)$ (rename if necessary)
      $C \leftarrow C \cup \{f(\gamma)\}$
      for all ϵ with $f(\epsilon) = f(\gamma)$ // refinement of f **do**
        $f(\epsilon) \leftarrow f(\delta)$
      end for
    end if
  end for
end while
return f

For the sake of analysis of this algorithm, we define f_i to be the function f in the algorithm after the i-th refinement of f in the inner for loop. In this notation, f_0 represents the initially defined f,

$$f_0(\alpha) = \begin{cases} 1 & (\alpha = \omega), \\ \alpha & (\alpha \neq \omega). \end{cases}$$

Let r be the highest index of these refinements. For each function f_i, we define an equivalence relation on A that partitions A by any element's image under f_i. For $\alpha, \beta \in A$, we say $\alpha \equiv \beta \iff f(\alpha) = f(\beta)$ and define Π_i to be the partition induced by the classes by this equivalence relation, and let $\Pi_i(\alpha)$ be the equivalence class that contains α.

In each refinement of f, we define f_{i+1} to be

$$f_{i+1}(\alpha) = \begin{cases} f_i(\delta) & (f_i(\alpha) = f_i(\gamma)), \\ f_i(\delta) & (\alpha = \delta), \\ f_i(\alpha) & \text{otherwise.} \end{cases}$$

Notice that in the second and third case, the values are taken from f_i and are not changed; in the first case however, for all α with $f_i(\alpha) = f_i(\gamma)$, we now have $f_{i+1}(\alpha) = f_i(\delta)$. Consequently, we are merging the equivalences classes $\Pi_i(f_i(\gamma))$ and $\Pi_i(f_i(\delta))$ from the partitioning Π_i into one partition $\Pi_{i+1}(f_{i+1}(\delta))$ in Π_{i+1}.

Lemma 14. *With the definitions from above:*

1. *We have $f_0(\alpha) \in \Pi_0(\alpha)$ for all $\alpha \in A$.*
2. *It is $f_i(\alpha) \in \Pi_i(\alpha)$ for all $\alpha \in A$ and $i = 1, ..., r$.*
3. *If $f_i(\alpha) = f_i(\beta)$ then $f_j(\alpha) = f_j(\beta)$ for $j = i, ..., r$.*
4. *For $i = 0, 1, ..., r$, the function f is idempotent: $f_i^2 = f_i$.*

Proof. 1. We have $f_0(\alpha) = \alpha$, and $\Pi_0(\alpha)$ is the partition containing α.

2. Since $f_0(\alpha) \in \Pi_0(\alpha)$, and each refinement of f only merges two partitions into one, we can conclude by an induction argument that $f_i(\alpha) \in \Pi_i(\alpha)$.
3. The statement $f_i(\alpha) = f_i(\beta)$ means that α and β belong to the same equivalence class. By construction and refinement of f, no class gets ever split up. Partitions are only merged. Therefore, the refinement of f will not separate two previously related elements of A.
4. For any $i = 0, 1, ..., r$ and $\alpha \in A$ we have $f_i(\alpha) = \beta$ with a $\beta \in \Pi_i(\alpha)$. Thus we have $f_i(f_i(\alpha)) = f_i(\beta) = \beta$ as $\alpha \equiv \beta$.

□

Lemma 15. *With the definitions from above:*

1. *It holds that $\alpha \geq f_0(\alpha) \geq f_1(\alpha) \geq ... \geq f_r(\alpha)$.*
2. *We have $\beta \neq f_r(\beta)$ for a $\beta \in A$ if and only if $b \in C$ at some point during execution.*
3. *In the case of (2), there exists $\alpha < \beta$ such that $f_r(\alpha) = f_r(\beta)$ and $f_r(\alpha g_j) = f_r(\beta g_j)$ for $j = 1, ..., m$.*

Proof. 1. By definition we have $\alpha \geq f_0(\alpha)$, and by renaming γ and δ if necessary we make sure that $f_i(\alpha) \geq f_{i+1}(\alpha)$.

2. Assume β was added to C during initialization, then $\beta = \omega$ and $\omega > f_0(\omega) = 1 = f_r(\omega)$. If β was added to C in the guise of $f(\gamma)$, then for some $i = 1, ..., r-1$ we have $f_{i+1}(\beta) = f(\delta) < f(\gamma) = \beta$ with $f(\delta)$ as defined in the algorithm. Conversely, if β never belonged to C then $f(\beta) = \beta$.
3. Assume β is the element that was deleted from C at the beginning of the while-loop. We know that $f_i(\beta) < \beta$ for some i and set $\alpha = f_i(\beta)$. From this we derive $f_i(\alpha) = f_i^2(\beta)$ and with Lemma 14(4) we conclude $f_i(\alpha) = f_i(\beta)$. Hence, $\alpha \equiv \beta$ and as classes only merge, we have $f_r(\alpha) = f_r(\beta)$. After the refinement of f in the inner for loop for some j we have $f_k(\alpha g_j) = f_k(\beta g_j)$ and thus $f_r(\alpha g_j) = f_r(\beta g_j)$.

□

Lemma 16. *The partitioning Π_r is invariant under G.*

Proof. We proof that every $g_1, ..., g_m$ preserves Π_r, this guarantees that any combination of g_k also preserves Π_r and thus Π_r is invariant under G. We assume that Π_r is not invariant under G, so suppose there are $\theta, \phi \in A$, $\theta < \phi$, such that $f_r(\theta) = f_r(\phi)$ but $f_r(\theta g_j) \neq f_r(\phi g_j)$. We choose θ minimal fulfilling this condition. We obtain $f_r(\phi) = f_r(\theta) \leq \theta < \phi$. By Lemma 15(3), ϕ was in C at some point during execution. Thus, by definition of the algorithm, there is an $\alpha < \phi$ with $f_r(\alpha) = f_r(\phi)$ and therefore $f_r(\alpha g_j) = f_r(\phi g_j)$. This yields the contradiction $f_r(\phi g_j) = f_r(\alpha g_j) = f_r(\theta g_j)$. □

The invariance of Π_r means that $\Delta = \Pi_r(1)$ is a G-block containing 1 and ω. The partitioning Π_r is a block system containing Δ.

Lemma 17. *The block Δ is the smallest block containing 1 and ω.*

Proof. Let Δ_1 be the smallest block containing 1 and ω so that $\Delta_1 \subseteq \Delta$. The set $\hat{\Pi} = \{g(\Delta_1) \mid g \in G\}$ is then a partition of A. We will show with an induction argument that each Π_i is a refinement of $\hat{\Pi}$. The base case Π_0 is a refinement of $\hat{\Pi}$ by definition. We now assume Π_i is a refinement. Each partition in Π_{i+1} is either also a partition of Π_i or the union of two partitions of Π_i. In the first case, we are done. In the latter case, the union can be written as

$$\Pi_i(f_i(\gamma)) \cup \Pi_i(f_i(\delta)) = \Pi_i(\gamma) \cup \Pi_i(\delta)$$

with $\gamma = \alpha g_j$ and $\delta = \beta g_j$, and $\Pi_i(\alpha) = \Pi_i(\beta)$, using the symbols from the algorithm's definition. By inductive assumption, $\hat{\Pi}(\alpha) = \hat{\Pi}(\beta)$. This yields

$$\hat{\Pi}(\gamma) = \hat{\Pi}(\alpha)g_j = \hat{\Pi}(\beta)g_j = \hat{\Pi}(\delta) \supseteq \Pi_i(f_i(\gamma)) \cup \Pi_i(f_i(\delta)).$$

We can conclude $\Delta = \Pi_r(1) \subseteq \hat{\Pi}(1) = \Delta_1$. □

Theorem 18. *Given a set of generators $\{g_1, ..., g_m\}$ of $G \leq \operatorname{Sym} A$, $A = \{1, ..., n\}$, and an element $\omega \in A$, we can compute a block system containing the smallest block that contains $\{1, \omega\}$.*

Proof. Algorithm 4 returns f, which describes the block system containing the block for $\{1, \omega\}$ by Lemma 17. The algorithm can operate in polynomial time, since the while-loop loops for every $\beta \in C$ at most once, the outer for-loop has a fixed length and the inner for-loop loops at most once for every $\epsilon \in A$. All other instructions can be carried out in constant time, respectively. □

4 Graphs with Valence Bounded By Three

In this section, we will show that testing the existence of an isomorphism of graphs with valence bounded by three is possible in polynomial time.

Definition 19. We define the following problems.

1. Given two trivalent graphs, the *Graph Isomorphism Problem for Trivalent Graphs* is to decide whether two graphs are isomorphic.
2. Given a trivalent graph X, the *Automorphism Generator Problem for Trivalent Graphs (with respect to e)* is to find a generating set for the group $\mathrm{Aut}_e(X)$, where e is a fixed edge in X.
3. Given a set of generators for a 2-subgroup G of $\mathrm{Sym}(A)$ with A being a colored set, the *Color Automorphism Problem for 2-groups* is to find generators for the biggest color-preserving subgroup of G.

Theorem 20. *The Graph Isomorphism Problem for Trivalent Graphs is polynomial-time reducible to the Automorphism Generator Problem for Trivalent Graphs.*

Theorem 21. *The Automorphism Generator Problem for Trivalent Graphs is polynomial-time reducible to the Color Automorphism Problem for 2-groups.*

Theorem 22. *There is a polynomial-time algorithm for the Color Automorphism Problem for 2-groups.*

Combining the results of these theorems yields

Corollary 23. *There is a polynomial-time algorithm for the Graph Isomorphism Problem for Trivalent Graphs.*

Proof. With Theorems 20 and 21 we obtain a polynomial time reduction from the Graph Isomorphism Problem for Trivalent Graphs to the Color Automorphism Problem for 2-groups. Theorem 22 states the latter is solvable in polynomial time. Therefore, we can decide the Graph Isomorphism Problem for Trivalent Graphs in polynomial time. $\square$

We will prove the theorems separately in the following subsections. To ensure the integrity of the proofs, no reference is made to the theorems above.

4.1 Reduction to the Automorphism Generator Problem

Let $X_1 = (V_1, E_1)$ and $X_2 = (V_2, E_2)$ be two disjoint graphs with valence bounded by three. We will state a polynomial-time algorithm to check whether X_1 and X_2 are isomorphic based on an algorithm that determines $\mathrm{Aut}_e(X)$, where X will be a trivalent graph built of X_1 and X_2.

By Lemma 5, the Graph Isomorphism Problem is polynomial time reducible to the Graph Isomorphism Problem for connected graphs, we assume X_1 and X_2 to be connected.

Definition 24. Let X_1 and X_2 be two graphs containing the edges $e_1 = v_1w_1 \in E_1$ and $e_2 = v_2w_2 \in E_2$. Choose distinct $x_1, x_2 \notin V_1 \cup V_2$ and define $X_1 * X_2$ to be the graph with nodes $V_1 \cup V_2 \cup \{x_1, x_2\}$ and edges

$$(E_1 - e_1) \cup \{v_1x_1, x_1w_1\} \;\cup\; (E_2 - e_2) \cup \{v_2x_2, x_2w_2\} \;\cup\; \{x_1x_2\},$$

that is, we insert the new nodes x_1 and x_2, breaking up edges e_1 and e_2 in two pieces, and connect x_1 and x_2 with a new edge.

Figure 4.1 shows $X_1 * X_2$. Notice that, since X_1 and X_2 are connected, and (x_1, x_2) connects two connected parts, $X_1 * X_2$ is connected. Furthermore, since the degree of v_1, v_2, w_1, w_2 remains unchanged and the degree of x_1 and x_2 is three, $X_1 * X_2$ has still valence three.

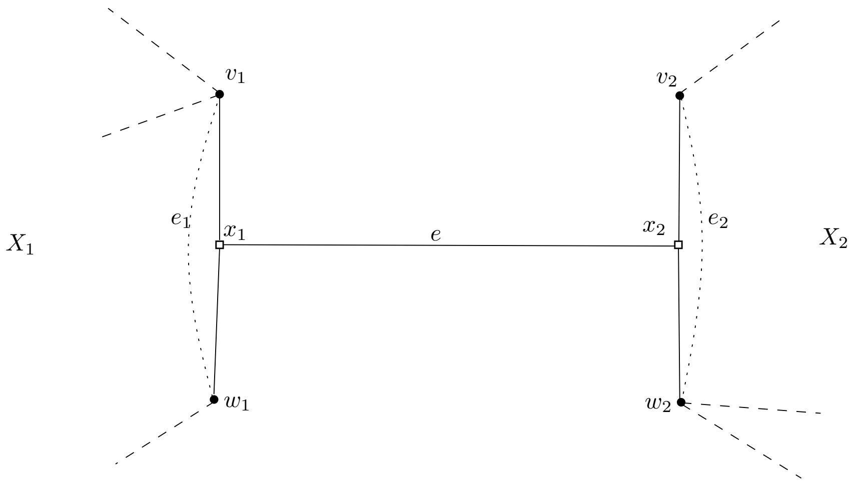


Figure 4.1: The graph $X_1 * X_2$, with X_1 being on the left and X_2 on the right. Untouched edges of X_1 and X_2 are dashed, new edges are painted solid. The two new nodes are shown as squares, whereas previously existing nodes are shown as disks. The edges drawn as dotted lines have been removed from the graph.

Although $X_1 * X_2$ depends on the choice of e_1 and e_2, all presented results does not depend on the choice of e_1 and e_2. We therefore omit this dependency in our formal notation.

Lemma 25. *Two graphs $X_1 = (V_1, E_1)$ and $X_2 = (V_2, E_2)$ are isomorphic if and only if there is an automorphism of $X_1 * X_2$ that transposes $x_1 x_2$, where x_1 and x_2 are the two newly added nodes of $X_1 * X_2$.*

Proof. Let $\sigma : X_1 \to X_2$ be an isomorphism. Then $\widetilde{\sigma} : (X_1 * X_2) \to (X_1 * X_2)$,

$$x \mapsto \widetilde{\sigma}(x) = \begin{cases} \sigma(x) & x \notin \{x_1, x_2\} \\ x_1 & x = x_2 \\ x_2 & x = x_1 \end{cases}$$

is an automorphism on $(X_1 * X_2)$ that transposes $x_1 x_2$. Notice that any node in $(X_1 * X_2) \cap X_1$ is mapped to X_2 and vice versa.

Let σ be an automorphism on $(X_1 * X_2)$ that transposes $x_1 x_2$, that is, we have $\sigma(x_1) = x_2$ and $\sigma(x_2) = x_1$. Since the automorphism preserves edge relations, any neighbor of x_1 (except for x_2) is mapped to some node in X_2. By induction, σ not only transposes $x_1 x_2$ but switches the connected components of $X_1 * X_2 - x_1 x_2$. Therefore, $\sigma|_{X_1}$ is a one-to-one mapping onto X_2 and X_1 and X_2 are isomorphic. □

Lemma 26. *Let X_1 and X_2 be two graphs, x_1, x_2 be the added nodes in $X_1 * X_2$ and let $e = x_1 x_2 \in E(X_1 * X_2)$. If there is an automorphism transposing x_1 and x_2, then any set of generators for $\mathrm{Aut}_e(X)$ will contain one.*

Proof. We will proof the contrapositive. Let G be a generating set of $\mathrm{Aut}_e(X_1 * X_2)$ that does not contain an automorphism transposing x_1 and x_2. Then for any $g \in G$, we have $g(x_1) = x_1$. As G is a generating set, for any $\sigma \in \mathrm{Aut}_e(X)$ there is a finite composition of $g_1, ..., g_n \in G \cup G^{-1}$ with $g_1 \circ ... \circ g_n = \sigma$. This yields $\sigma(x_1) = (g_1 \circ ... \circ g_n)(x_1) = x_1$. Thus, there is no automorphism transposing x_1 and x_2. □

Using this, we can go ahead and proof Theorem 20 by stating a polynomial time reduction. As a reminder, Theorem 20 states that the problem of deciding whether two trivalent graphs are isomorphic is polynomial time reducible to the problem of finding a generating set for $\mathrm{Aut}_e(X)$, where e is a fixed edge in the trivalent graph X.

Proof of Theorem 20. Assume we have an algorithm M that determines a generating set for $\mathrm{Aut}_e(X)$ for any trivalent graph X. Let X' and X'' be two connected trivalent graphs. Constructing $X' * X''$ involves adding and removing a constant number of edges and nodes and can thus be done in polynomial time. Combining Lemmas 25 and 26, X' and X'' are isomorphic if and only if the generating set returned by M on input $X' * X''$ contains an automorphism transposing the edge (x_1, x_2). As the size of the generating set is bounded by $\log_2 |Aut_e(X' * X'')|$ by Lemma 6 with $|\mathrm{Aut}_e(X' * X'')|$ being of exponential size in the input length $|(X', X'')|$, we can verify this condition in polynomial time. □

4.2 Reduction to the Color Automorphism Problem for 2-groups

Let X be a connected trivalent simple graph with n edges and $e = (v_1, v_2) \in E(X)$ be a distinguished edge. Let M be an algorithm that, given a colored set A and a set of generators for a 2-subgroup G of $\mathrm{Sym}\, A$, finds a generating set for the subgroup of color-preserving elements of G. We will show that we can determine a generating set for $\mathrm{Aut}_e(X)$ in polynomial time using polynomial-many calls to M.

Definition 27. For $r \in \{1, ..., n\}$, let X_r be the subgraph of X consisting of all edges and nodes that appear on paths of length up to r through e. Let $\pi_r : \mathrm{Aut}_e(X_{r+1}) \to \mathrm{Aut}_e(X_r)$ defined by $\sigma \mapsto \sigma|_{X_r}$, be the restriction of σ to X_r. For any σ, we have $\pi_r(\sigma) \in \mathrm{Aut}_e(X_r)$.

We obtain that $X_1 = (\{v_1, v_2\}, \{(v_1, v_2)\})$, since there is only one path of length one through e. Furthermore, it is $X_n = X$, since the distance from e to any node in X is less or equal to n.

Lemma 28. *For an graph X and an edge e we have $\mathrm{Aut}_e(X_1) = \{\sigma, \tau\}$ with σ being the identity function and τ transposing the two vertices of X_1.*

Proof. As mentioned before, X_1 is the graph consisting of all edges and nodes appearing of paths of length one through e and is thus only e itself. There are only two bijections of two elements, and both turn out to be an automorphism on X_1. Therefore, $\mathrm{Aut}_e(X_1) = \{\sigma, \tau\}$. □

Lemma 29. *The function π_r, the restriction of automorphisms in $\mathrm{Aut}_e(X_{r+r})$ to X_r, is a group homomorphism.*

Proof. Let $\sigma, \tau \in \mathrm{Aut}_e(X_{r+1})$. As $\sigma \circ \tau$ is a automorphism on X_{r+1} that fixes edge e, the restriction to $V(X_r)$ yields an automorphism in $\mathrm{Aut}(X_r)$. We thus obtain $\pi_r(\sigma \circ \tau) = (\sigma \circ \tau)|_{X_r} = (\sigma|_{X_r} \circ \tau|_{X_r}) = \pi_r(\sigma) \circ \pi_r(\tau)$. □

This shows that the layered structure of $X_1, ..., X_r$ is reflected in the automorphism groups $\mathrm{Aut}_e(X_1), ..., \mathrm{Aut}_e(X_r)$ and can be connected by the group homomorphism π_r. To get some more insight on the layer structure, we will have a closer look on the new nodes that are added with each layer.

Definition 30. For any node $v \in V(X_{r+1}) \setminus V(X_r)$, we define $f(v)$ to be the set of all neighbors of v in $V(X_r)$. That is, f is a function that maps nodes in $V(X_{r+1}) \setminus V(X_r)$ to A, with A being the set of all 1-, 2- and 3-subsets of $V(X_r)$. (There is no node in $X_{r+1} \setminus X_r$ that has no neighbor, because of the way the graph is constructed. There is no node with more than three neighbors, because valence is bounded by 3. Also notice that X_{r+1} does not contain edges in between nodes of $X_{r+1} \setminus X_r$, so $f(v) \subseteq V(X_r)$.) Any two different nodes $v_1, v_2 \in V(X_{r+1}) \setminus V(X_r)$ are called *twins*, if $f(v_1) = f(v_2)$.

If v_1 and v_2 are twins, there is no v_3 that is a twin to v_1 or v_2, which we prove by contradiction: assume $v_1, v_2, v_3 \in V(X_{r+1}) \setminus V(X_r)$ are (pair-wise) twins. Then there is at least one node w in X_r that is adjacent to all three nodes. However, since w is part of X_r, for $r \geq 2$, w must be adjacent to a node in X_{r-1}. For $r = 1$, w is either e_1 or e_2. In both cases, w has degree four, which contradicts the assumption of X being a trivalent graph.

Lemma 31. *For any $\sigma \in \mathrm{Aut}_e(X_{r+1})$ and any $v \in V(X_{r+1}) \setminus V(X_r)$, it holds that $f(\sigma(v)) = \sigma(f(v))$.*

Proof. Let $x \in f(\sigma(v))$. Then $x \in V(X_r)$ and x is adjacent to $\sigma(v)$. Then $\sigma^{-1}(x) \in V(X_r)$ is adjacent to v. Therefore, $\sigma^{-1}(x) \in f(v)$ and hence $x \in \sigma(f(v))$.

Now let $x \in \sigma(f(v))$. Then $\sigma^{-1}(x) \in f(v)$, that is, $\sigma^{-1}(x)$ is adjacent to v. Therefore, x is adjacent to $\sigma(v)$ and thus $x \in f(\sigma(v))$. □

Lemma 32. *A set of generators for $\mathrm{Ker}\,\pi_r$ can be determined in polynomial time. The set of generators contains only elements of order 2.*

Proof. Let $\sigma \in \mathrm{Aut}_e(X_{r+1})$. If $\sigma \in \mathrm{Ker}\,\pi_r$, that is, σ fixes all nodes of X_r, we have $f(\sigma(v)) = f(v)$, since $f(v) \subseteq V(X_r)$ for every node v in X_{r+1} by definition. It follows that either $\sigma(v) = v$ or $\sigma(v)$ and v are twins. Since σ preserves neighborhoods, σ either maps v to itself or transposes $\sigma(v)$ and v. The subgroup $\mathrm{Ker}\,\pi_r$ is thus generated by the transpositions of each pair of twins. Following the layered structure of X_r, we can determine all pairs of twins in polynomial time: For each layer in $r \in 1, ..., n-1$ we determine the neighborhoods of all vertices in $X_{r+1} \setminus X_r$. For every pair of twins, we add the transposition to the generator. □

Corollary 33. *For each r, the size of $\mathrm{Aut}_e(X_r)$ is a power of 2.*

Proof. The kernel $\mathrm{Ker}\,\pi_r$ is generated by elements of order 2, that is, an abelian 2-group. We have $|\mathrm{Aut}_e(X_{r+1})| = |\mathrm{Im}\,\pi_r| \cdot |\mathrm{Ker}\,\pi_r|$ and therefore, by induction, the size of $\mathrm{Aut}_e(X_r)$ is a power of 2. □

A closer look at the layered structure of the graphs $X_1, ..., X_r$ will unveil a way to find generators for $\mathrm{Im}\,\pi_r$. To do that, we fix a layer r for the following theorems and assume by Lemma 28 and induction, we know a generating set for $\mathrm{Aut}_e(X_{r-1})$. Using f, we will divide the set A of all 1-, 2- and 3-subsets of $V(X_r)$ into three disjoint subsets A_1, A_2 and $A_0 = A \setminus (A_1 \cup A_2)$.

Definition 34. Let A_1 be the set of all 1-, 2- and 3-subsets a of $V(X_r)$ for which there is only one unique $v \in V(X_{r+1}) \setminus V(X_r)$ with $f(v) = a$. Furthermore, let A_2 be the set of all 1-, 2- and 3-subsets a that are adjacent to twins, i.e. all $a \in A$ such that $a = f(v_1) = f(v_2)$ for some $v_1, v_2 \in V(X_{r+1}) \setminus V(X_r)$, $v_1 \neq v_2$. Finally, let A' be the set all 2-subsets of $V(X_r)$ that are, in X_{r+1}, adjacent to each other.

Lemma 35. *Any member of $\mathrm{Im}\,\pi_r$ stabilizes A_1, A_2 and A'.*

Proof. Let $\tau \in \mathrm{Im}\,\pi_r$, and let $\sigma \in \mathrm{Aut}_e(X_{r+1})$ such that $\pi_r(\sigma) = \tau$. Notice that for every node v in X_r we have $\tau(v) = \sigma(v)$.

Let $a \in A_1$. We will show that $\tau(a) = \sigma(a)$ is a member of A_1 as well. Suppose w and w' are nodes in $X_{r+1} \setminus X_r$ such that $f(w) = f(w') = \sigma(a)$. We will show that in this case, we always have $w = w'$. There are $v, v' \in V(X_{r+1}) \setminus V(X_r)$ such that $f(\sigma(v)) = f(\sigma(v')) = \sigma(a)$. Applying Lemma 31, we obtain $\sigma(f(v)) = \sigma(f(v')) = \sigma(a)$ and therefore $f(v) = f(v') = a$. Since $a \in A_1$, it holds that $v = v'$ and $w = w'$. Hence, $\sigma(a) = \tau(a) \in A_1$ and τ stabilizes A_1.

Let $a \in A_2$. We are going to prove that $\tau(a)$ belongs to A_2. Since $a \in A_2$, there are $v_1, v_2 \in V(X_{r+1}) \setminus V(X_r)$ such that $f(v_1) = f(v_2) = a$ and $v_1 \neq v_2$. Hence, $\sigma(f(v_1)) =$

$\sigma(f(v_2)) = \sigma(a)$ and by Lemma 31 $f(\sigma(v_1)) = f(\sigma(v_2)) = \sigma(a)$ with $\sigma(v_1) \neq \sigma(v_2)$ and thus $\sigma(a) = \tau(a) \in A_2$. Therefore, τ stabilizes A_2.

Let $a \in A'$. Then $a = \{v_1, v_2\}$ such that $(v_1, v_2) \in E(X_{r+1})$. Notice that, by definition of A, $v_1, v_2 \in V(X_r)$. As σ preserves edge relations in X_{r+1}, $(\sigma(v_1), \sigma(v_2)) = (\tau(v_1), \tau(v_2)) \in E(X_{r+1})$ and therefore $\tau(a) \in A'$. Thus, τ stabilizes A'. $\square$

Lemma 36. *Any $\tau \in \mathrm{Aut}_e(X_r)$, that stabilizes A_1, A_2 and A' is a member of $\mathrm{Im}\,\pi_r$.*

Proof. We show that τ can be extended to an automorphism $\sigma \in \mathrm{Aut}_e(X_{r+1})$. Let $\sigma|_{X_r} = \tau$, definitions for $\sigma(v)$ with v being a node of $X_{r+1} \setminus X_r$ follow below.

For any $v \in V(X_{r+1}) \setminus V(X_r)$ with $f(v) \in A_1$ we have $\tau(f(v)) \in A_1$, since τ stabilizes A_1. As $\tau(f(v)) \in A_1$, there is a uniquely determined w such that $f(w) = \tau(f(v))$. Therefore, we define $\sigma(v) = w$. Since any neighbor of v is a member of $f(v)$ and any neighbor of w lies in $f(w) = \tau(f(v))$, this extension of τ preserves edge relations.

For any $v_1, v_2 \in V(X_{r+1}) \setminus V(X_r)$ with $a := f(v_1) = f(v_2) \in A_2$, it holds that $\tau(a) \in A_2$, since τ stabilizes A_2. Similar to the mapping of A_1, there are two nodes $w_1, w_2 \in V(X_{r+1}) \setminus V(X_r)$ such that $f(w_1) = f(w_2) = \tau(a)$. We define $\sigma(v_1) = w_1$ and $\sigma(v_2) = w_2$. This preserves edge relations, as for $i \in \{1, 2\}$ we have $f(w_i) = \tau(a)$.

This yields an automorphism $\sigma \in \mathrm{Aut}_e(X_{r+1})$. Therefore, $\tau \in \mathrm{Im}\,\pi_r$. $\square$

Corollary 37. *Given a set of generators for $\mathrm{Aut}_e(X_r)$, a set of generators for $\mathrm{Im}\,\pi_r$ can be determined in polynomial time, using one call to M.*

Proof. Let H be a set of generators for $\mathrm{Aut}_e(X_r)$.

Let M be an algorithm that finds the biggest color-preserving subgroup of G, where $G \leq \mathrm{Sym}\,A$ is a 2-group and A is a colored set. We color A with six colors to distinguish the partitions

$$A_0 \cap A', \quad A_1 \cap A', \quad A_2 \cap A', \quad A_0 \setminus A', \quad A_1 \setminus A', \quad A_2 \setminus A',$$

with $A_0 = A \setminus (A_1 \cup A_2)$ as shown in Figure 4.2.

By Corollary 33 we know that $\mathrm{Aut}_e(X_r)$ is a 2-subgroup of $\mathrm{Sym}\,A$. Combining Lemma 35 and 36, we know that $\mathrm{Im}\,\pi_r$ is exactly the group of all automorphisms in $\mathrm{Aut}_e(X_r)$ that stabilize the sets A_1, A_2 and A'. Hence, it also stabilizes A_0 and thus all six partitions. Vice versa, any automorphism σ that stabilizes all six partitions, also stabilizes A_1, A_2 and A', and therefore $\sigma \in \mathrm{Im}\,\pi_r$. We can conclude that the set of all stabilizing automorphisms (with respect to the six partitions above) in $\mathrm{Aut}_e(X_r)$ is exactly $\mathrm{Im}\,\pi_r$.

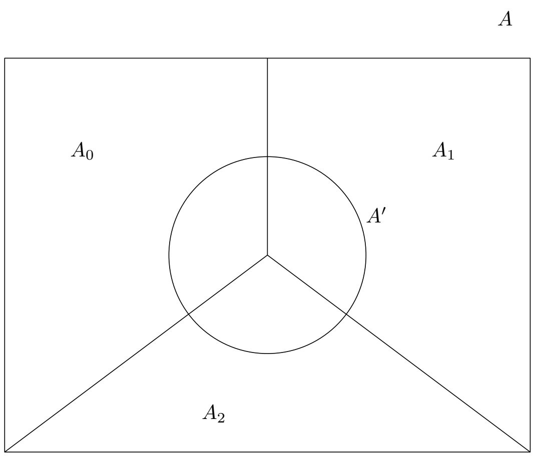


Figure 4.2: The six partitions of A.

This enables us to use algorithm M on the above coloring of A and the generating set H of $\mathrm{Aut}_e(X_r)$ to find a generating set of $\mathrm{Im}\,\pi_r$ in polynomial time. $\square$

Corollary 38. *Given a set of generators for* $\mathrm{Aut}_e(X_r)$*, we can determine a generating set for* $\mathrm{Aut}_e(X_{r+1})$ *in polynomial time with one call to* M*.*

Proof. By Lemma 32, we can determine a generating set K for $\mathrm{Ker}\,\pi_r$ in polynomial time and by Corollary 37, we can determine a generating set $I' \subseteq \mathrm{Aut}_e(X_r)$ for $\mathrm{Im}\,\pi_r$ in polynomial time using one call to M. We can extend each automorphism in I' using the identity function on any node in $X_{r+1} \setminus X_r$ and thus obtain the set

$$I = \{\sigma' \mid \sigma' \text{ is } \sigma \text{ extended with identity on all nodes in } X_{r+1} \setminus X_r\}.$$

This yields $f(I) = I'$ and we can use Lemma 3 to obtain that $K \cup I$ generates $\mathrm{Aut}_e(X_{r+1})$. $\square$

Proof of Theorem 21. We know $\mathrm{Aut}_e(X_1)$ by Lemma 28. With $|X|$ iterations of Corollary 38, we can obtain $\mathrm{Aut}_e(X)$ using polynomial time and $|X|$ calls to M. $\square$

4.3 Solving the Color Automorphism Problem for 2-groups in polynomial time

The Graph Isomorphism Problem for graphs with valence bounded by three was reduced to the Color Automorphism Problem for 2-groups by Sections 4.1 and 4.2. This section will show that there is a polynomial time divide and conquer algorithm that can solve the Color Automorphism Problem. Ultimately, this will prove that we can solve the Graph Isomorphism Problem for graphs with valence bounded by three in polynomial time as stated in Corollary 23.

To unveil the recursive structure of the problem, we need the notion of a "filter" for color-preserving elements from a set of permutations.

Definition 39. For a colored set A, a subset $B \subseteq A$ and $K \subseteq \mathrm{Sym}(A)$ we define $C_B(K) = \{\sigma \in K \mid \sigma(b) \sim b \text{ for all } b \in B\}$, where $a \sim b$ for $a, b \in A$ is true if and only if a and b have the same color.

We can think of $C_B(K)$ as all permutations in K that preserve colors for all elements in B.

Let A be a colored set. As stated in Definition 19(3), we need to give a polynomial-time algorithm that finds the biggest color-preserving subgroup of G, where G is a 2-subgroup of $\mathrm{Sym}(A)$ given by a set of generators. That is, we need to find $C_A(G)$. We will prove in Lemma 40(3) that $C_A(G)$ is indeed a group.

Lemma 40. *Let* $K, K' \subseteq \mathrm{Sym}(A)$*,* $B', B'' \subseteq B$ *and let* G *be a subgroup of* $\mathrm{Sym}(A)$*.*

1. $C_B(K \cup K') = C_B(K) \cup C_B(K')$.
2. $C_{B' \cup B''}(K) = C_{B''}(C_{B'}(K))$.
3. *If* G *stabilizes* B*, then* $C_B(G)$ *is a subgroup of* G*.*
4. *Let* G *stabilize* B*. If* $C_B(\sigma G)$ *is not empty then it is a left coset of the subgroup* $C_B(G)$*.*

Proof.
1. The color-preserving elements in $K \cup K'$ are exactly the color-preserving elements in K plus the ones in K'.
2. Any $\sigma \in K$ that preserves color for all elements in B' and B'' preserves the color for all elements in $B' \cup B''$, and vice versa.

3. For $1 \in \operatorname{Sym}(A)$ we have $1 \in G$, and 1 preserves color, therefore $1 \in C_B(G)$. For any two elements σ and τ in G that preserve color, $\sigma \circ \tau$ preserves color as well. $\sigma \circ \tau$ is a permutation on B because G stabilizes B. If σ preserves color on B, so does σ^{-1} (otherwise, $\sigma \circ \sigma^{-1} \neq 1$).

4. We just demonstrated that $C_B(G)$ is a subgroup of G. For a $\sigma_0 \in C_B(\sigma G)$ we have $\sigma G = \sigma_0 G$, as $\sigma_0 = \sigma g$ for some $g \in G$. We are going to show that $C_B(\sigma_0 G) = \sigma_0 C_B(G)$. Let $\tau \in G$ and $b \in B$, then we have $\tau(b) \in B$, since B is G-stable. Since σ_0 is color-preserving for all elements of B, we $\sigma_0\tau(b)$ and $\tau(b)$ have the same color. Hence, $\sigma_0\tau$ is in $C_B(\sigma_0 G)$ if and only if $\tau \in C_B(G)$. That is, $C_B(\sigma_0 G) = \sigma_0 C_B(G)$.

□

To allow recursive calls, the algorithm's input will be a set $B \subset A$ and a coset, represented by a $\sigma \in \operatorname{Sym} A$ and a generating set for a subgroup G. The algorithm will then find $C_B(\sigma G)$. By Lemma 40(4), this is either empty or a coset as well. We can thus represent the output by a generator and an element of $\operatorname{Sym} A$. To use the algorithm to find $C_A(G)$, we will use A, $1 \in \operatorname{Sym} A$ and a generating set for G as input values.

Each recursive call of the algorithm will use two or four sub-calls to itself, using a (possibly) different coset and some $B' \subsetneq B$ as input. The smaller B' guarantees the algorithm will terminate eventually, since for G-stable B with $|B| = 1$, $B = \{b\}$, we have

$$C_B(\sigma G) = \begin{cases} \sigma G & \text{if } \sigma(b) \sim b, \\ \emptyset & \text{if } \sigma(b) \nsim b. \end{cases}$$

Algorithm 5 Polynomial-time divide and conquer algorithm for the Color Automorphism Problem for 2-groups.

input: a set $B \subseteq A$, permutation σ, set of generators for 2-subgroup $G \leq \operatorname{Sym} A$
output: $C_B(\sigma G)$, a coset represented by a set of generators and a permutation

// base case
if $|B| = 1$ **then**
  let b be the only element in B
  if $\sigma(b) \sim b$ **then**
    return σG
  else
    return $\emptyset$
  end if
end if

// recursive step
if B is a union of G-stable subsets B', B'' **then**
  // divide an conquer by Lemma 40(2)
  find B', B'' (Thm. 8)
  return $C_B(\sigma G) = C_{B''}(C_{B'}(\sigma G))$
else
  // divide and conquer by Lemma 41
  find G-blocks B' and B'' such that $B = B' \cup B''$ (Thm. 18)
  find a subgroup H with $G = H \cup \tau H$ that stabilizes B' and B'' (Thm. 13)
  return $C_B(\sigma G) = C_{B''}(C_{B'}(\sigma H)) \cup C_{B''}(C_{B'}(\sigma\tau H))$
end if

The correctness of the first recursive branch of the algorithm was already shown in Lemma 40(2). The following lemma proves the correctness of the second branch.

Lemma 41. *If B is not the union of two G-stable subsets, then there are G-blocks B' and B'', a B'- and B''-stable subgroup $H \leq G$ and a $\tau \in \operatorname{Sym} A$ such that $B = B' \cup B''$ and $G = H \cup \tau H$. Furthermore, $C_B(\sigma G) = C_{B''}(C_{B'}(\sigma H)) \cup C_{B''}(C_{B'}(\sigma\tau H))$.*

Proof. Lemma 2 guarantees the existence of B' and B'', and Theorem 13 shows the existence and polynomial-time computability of H. The representation of the result follows from Lemma 40. □

Theorem 42 (Correctness of Algorithm 5)**.** *Given a colored set A and a 2-subgroup G of $\operatorname{Sym} A$, Algorithm 5 returns a set of generators for $C_A(G)$, which is the biggest color-preserving subgroup of G.*

Proof. For the first input to the algorithm, we set $\sigma = 1$ and $B = A$. The correctness in each case is guaranteed by Lemma 40(2) in the intransitive case and Lemma 41 in the transitive case. The algorithm thus returns $C_A(G)$, which is the biggest color-preserving subgroup by 40(3). □

Lemma 43. *In Algorithm 5, the runtime for each recursive step is polynomial in n.*

Proof. Checking and handling for the base case only takes constant time. For the recursive step we need to check whether the action of G on B is transitive or intransitive. We can do this in polynomial time with Algorithm 1 (Theorem 8). In the intransitive case, we compute orbits B' and B'' and do two recursive calls. (The recursive calls are not accounted for in this proof, see also Lemma 44.) We do not need to combine the two sub-results in any way, as we just pass the results of recursive call of $C_{B'}(\sigma G)$ as input to $C_{B''}$. In the transitive case, we use Algorithm 4 (Theorem 18). We can then find a subgroup H according to our needs with Theorem 13. To make the recursive calls, we compute $\sigma\tau$ in linear time. For the union of two sub-results, we can compute the union of their generators in linear time. □

Lemma 44. *Given a colored set A with n elements and a generating set for a 2-subgroup G of $\operatorname{Sym} A$, Algorithm 5 uses less than $4 \cdot \log_2 |A|$ number of recursive calls.*

Proof. In each recursive step the algorithm splits up the given set B into B' and B'', having each half the size of B. In the transitive case, we have two recursive calls; the intransitive yields four recursive calls. The number of recursive calls is thus bounded by $4 \cdot \log_2 |A|$. □

Theorem 45 (Runtime of algorithm 5)**.** *Given a colored set A with n elements and a generating set for a 2-subgroup G of $\operatorname{Sym} A$, Algorithm 5 uses polynomial time in n to terminate.*

Proof. By Lemma 43, the runtime for each recursive step is polynomial in n. By 44, there are less than $4 \cdot \log_2 |A|$ recursive calls. This yields a polynomial runtime in n. □

With these preparations, we are now ready to proof our main result, showing that we can decide the existence of isomorphisms for graphs with valence bounded by three in polynomial time.

Proof of Theorem 22. Theorem 42 guarantees correctness of algorithm 5, while Theorem 45 proves polynomial runtime. Therefore, we have a polynomial-time algorithm for the Color Automorphism Problem for 2-groups. □

5 Conclusion

After giving a short introduction into the Graph Isomorphism Problem, we introduced the reader in Section 2 to notions in the fields of Algebra, Complexity Theory and Graphs and showed basic theorems in these fields. This included a short introduction into Group Theory, on which this entire thesis is based. In Complexity Theory, we introduced the reader into the basic notions of runtime and reductions; in Graph Theory, we defined the problem that gave this thesis it's name. Finally in the first section, we motivate the study of polynomial-time algorithms for groups by showing in Lemma 6 that groups can be represented in a succinct way.

In Section 3, we present some polynomial-time algorithms for problems related with permutation groups. These results are the foundation of our main result, but are interesting on their own as well. In the various subsections we show that for a given permutation group G, we can determine G-orbits, the size of G, stabilizing subgroups and minimal blocks systems in polynomial time.

Finally, we present out main result due to Luks [Luk82] in Section 4 by proofing that for graphs with valence bounded by three, we can decide the Graph Isomorphism Problem in polynomial time. The approach uses two different reductions from the Isomorphism Problem to the problem of generating certain automorphism groups to the problem of computing generators for a certain subgroup. The latter problem is then solved by a divide-and-conquer algorithm that relies on the results proven in Section 3.

We refer the reader to the Luks' paper [Luk82], in which he shows that for graphs with valence bounded by *any* constant, the Graph Isomorphism Problem can be decided in polynomial time. To show this, an abstraction of the algorithm described in this thesis is used.

References

[Atk75] Michael D Atkinson, *An algorithm for finding the blocks of a permutation group*, Mathematics of computation (1975), 911–913.

[Bra] Nicolas Bray, *Coset. From MathWorld—A Wolfram Web Resource, created by Eric W. Weisstein*, Last visited on October 24th, 2013.

[FHL80] Merrick Furst, John Hopcroft, and Eugene Luks, *Polynomial-time algorithms for permutation groups*, Proceedings of the 21st Annual Symposium on Foundations of Computer Science (Washington, DC, USA), SFCS '80, IEEE Computer Society, 1980, pp. 36–41.

[HS01] S. Homer and A.L. Selman, *Computability and complexity theory*, Texts in computer science, Springer, 2001.

[HT74] John Hopcroft and Robert Tarjan, *Efficient planarity testing*, Journal of the ACM (JACM) **21** (1974), no. 4, 549–568.

[Kre88] Mark W. Krentel, *The complexity of optimization problems*, J. Comput. Syst. Sci. **36** (1988), no. 3, 490–509.

[KS05] Neeraj Kayal and Nitin Saxena, *On the ring isomorphism and automorphism problems.*, IEEE Conference on Computational Complexity, IEEE Computer Society, 2005, pp. 2–12.

[Luk82] Eugene M. Luks, *Isomorphism of graphs of bounded valence can be tested in polynomial time.*, J. Comput. Syst. Sci. **25** (1982), no. 1, 42–65.

[Ser97] Ákos Seress, *An introduction to computational group theory*, Notices Amer. Math. Soc **44** (1997).

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde vorgelegt habe.

Berlin, 27. Mai 2015