Technische Universität München

Zentrum Mathematik

Stabilization Procedures and Applications

Oliver Bastert

Vollständiger Abdruck der von der Fakultät für Mathematik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Rupert Lasser

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Gottfried Tinhofer
2. Priv.-Doz. Dr. Luitpold Babel
3. Univ.-Prof. Dr. Adalbert Kerber
   Universität Bayreuth (schriftliche Beurteilung)

Die Dissertation wurde am 15.11.2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Mathematik am 10.01.2001 angenommen.

Stabilization Procedures and Applications

Oliver Bastert *
Zentrum Mathematik
Technische Universität Müchen
bastert@ma.tum.de

*The author has been mainly supported by the Deutsche Forschungsgemeinschaft through the graduate program Angewandte Algorithmische Mathematik, Technische Universität München, and partially by the German Israeli Foundation for Scientific Research and Development under contract # I-0333-263.06/93.

Thanks

I am grateful to my supervisor Prof. Dr. Gottfried Tinhofer who gave me the chance to work with him at the Zentrum Mathematik at the Technische Universiät München. He introduced me to the field of algorithmic and algebraic graph theory. His suggestions and comments improved this work a lot. I enjoyed the discussions on mathematical subjects as well as the conversations during the daily coffee breaks very much.

I thank Prof. Dr. Gritzmann and Dr. Andreas Brieden who spent lots of time to maintain the Graduiertenkolleg "Angewandte Algorithmische Mathematik" in which I was enrolled for the last two years.

I thank my former colleagues PD Dr. Luitpold Babel, Dr. Stefan Baumann and Dr. Mariel Lüdecke for a nice and friendly environment especially for an atmosphere lacking of envy and exaggerated ambition.

I am grateful to my parents Heera and Horst Bastert and to my parents in law Helga and Herbert Bauer who took care of Fabian several times during the last months.

I thank my children Fabian and Laurin. Fabian borrowed me his dinosaur and his hippopotamus to help me finishing my thesis and I thank Laurin for sleeping at night very soon after his birth.

I am indebted to my wife Dr. Petra Bauer who carefully read this thesis and provided many suggestions and corrections. Furthermore, she enabled that I could work more than usual during the last months.

Contents

1 Introduction **1**

2 Basics **5**
2.1 Definitions 5
2.1.1 Graphs 5
2.1.2 Colored Graphs 6
2.1.3 Isomorphism and Automorphism 8
2.1.4 Colorings 8
2.1.5 Notation 10
2.2 Structures 11
2.2.1 Equitable Partitions 11
2.2.2 Coherent Algebras 13
2.2.3 k-stable Colorings 15
2.3 Stabilization Procedures 16
2.3.1 1-dimensional Stabilization 16
2.3.2 2-dimensional Stabilization 18
2.3.3 k-dimensional Stabilization 21
2.4 Invariants, Canonical Colorings and Canonical Labelings 21
2.5 Complexity Theory 22

3 Algorithmic Aspects of Stabilization Procedures **25**
3.1 Discussion of Known Algorithms and Ideas 25
3.2 Prerequisites 28
3.3 1-stable Colorings 28
3.4 2-stable Colorings 33
3.4.1 Proper Colorings 33
3.4.2 The New Algorithm 35
3.5 k-stable Colorings 43
3.6 Computing the Basis of a Coherent Algebra 47

4 Further Aspects **49**
4.1 Another k-dimensional Approach 49
4.2 Relations Between k-dimensional Stabilization Algorithms 50

4.3 Pebbling Games . . . 51
4.4 k-dimensional Stabilization Does Not Suffice . . . 53
4.5 Upper and Lower Bounds on the Number of Steps . . . 56
4.5.1 A Lower Bound on the Number of Steps for $k = 2$ . . . 56
4.6 Classical Invariants . . . 58
4.6.1 The Spectrum . . . 58
4.6.2 The Powers of the Adjacency Matrix . . . 59
4.7 Pointed Graphs . . . 60
4.8 Coarsest Non-Trivial k-stable Partitions . . . 63
4.9 Cayley Graphs . . . 63

5 Applications 65
5.1 Robinson Graphs . . . 65
5.1.1 Landscapes . . . 65
5.1.2 Genetic Trees . . . 66
5.1.3 The Cell Partition of Γ_n . . . 69
5.1.4 Eigenvalues of the Laplacian of the Robinson Graph . . . 81
5.2 Graph Isomorphism, Automorphism and Canonical Labeling . . . 84
5.2.1 Canonical Labeling . . . 84
5.2.2 Isomorphism Testing . . . 86
5.2.3 Computations . . . 88
5.3 Polynomial Time Solvable Cases . . . 92
5.3.1 Compact and Weakly Compact Graphs . . . 92
5.3.2 Higher Dimensions . . . 93
5.3.3 Recognition . . . 94
5.4 Stabilization Procedures in Chemistry . . . 95
5.4.1 The Algorithm of G. Rücker and Ch. Rücker . . . 96
5.4.2 Computational Experiments . . . 98

6 Implementation Details and Computations 101
6.1 Computations . . . 101
6.1.1 2-stable Colorings . . . 101
6.1.2 1-stable Colorings . . . 113
6.2 The Test Instances . . . 114
6.3 Testing Environment . . . 115
6.3.1 **qStab** and **qWeil** . . . 116
6.3.2 Other Programs . . . 116

7 Conclusions 119

A The Complete Algorithm 121

Bibliography 125

Index **130**

Chapter 1

Introduction

Stabilization algorithms are useful tools for perceiving the symmetries of graphs and for recognizing whether two graphs are equivalent or not. These problems are known as the *graph automorphism* and the *graph isomorphism* problem, respectively. Both are strongly related to the problem of *canonically labeling* graphs. The graph isomorphism problem, the most prominent one among the above problems, has neither been shown to be $\mathcal{NP}$-complete nor to be polynomial time solvable, but there are strong indications that it is not $\mathcal{NP}$-complete [65]. The graph automorphism problem is polynomially time equivalent to the graph isomorphism problem. A polynomial algorithm for the canonical labeling problem would solve also the isomorphism problem.

In standard approaches for solving the above problems the *automorphism partition problem* plays a crucial role. The *automorphism partition* of a graph G is a partition of the vertex set with the following property: two vertices are in the same set of the partition if and only if there exists a graph automorphism which maps one of the vertices onto the other. The sets of the automorphism partition are the *orbits* of the *automorphism group* of G. Therefore, the automorphism partition is also called the *orbit partition* of G. Finding the orbit partition is as hard as the graph isomorphism problem.

Among other results, a *stabilization procedure* yields a partition of the vertex set which is in general coarser than the automorphism partition. Nevertheless, in many favorable cases these two partitions coincide. Stabilization procedures start with an initial partition, refine it iteratively, and stop if no further refinement is obtained. All stabilization procedures discussed in this thesis run in polynomial time.

H. L. Morgan was probably the first to deal with stabilization procedures [52]. Basically he did the following: assign to each vertex as a label the number of its neighbors. This assignment induces a partition of the vertex set. Next assign to each vertex the sum of the labels of its neighbors as a new label. Iterate this procedure until the number of different labels does not increase anymore.

H. L. Morgan invented this procedure to find a "unique machine description for chemical structures", i.e., he was looking for a canonical labeling of molecular graphs.

The 1-dimensional stabilization procedure discussed in this thesis is an improved version of such an algorithm. It computes a so-called *equitable partition*. This notion has been introduced by H. Sachs [63, 64] and has been since then broadly discussed in several publications, see for example [33]. Among others, D. G. Corneil and C. C. Gotlieb [19] improved this idea of iterated vertex coloring further when they were seeking for an efficient algorithm for solving graph isomorphism problems.

A new idea was introduced independently by B. J. Weisfeiler and A. A. Leman [73, 72], and J. Hinteregger and G. Tinhofer [39]. They not only partition the set of vertices but also the set of edges, and instead of considering neighbors only, they consider edges and the number and type of triangles to which they belong. This approach is called 2-dimensional stabilization.

B. J. Weisfeiler and A. A. Leman provide examples showing that their algorithm is not powerful enough to solve graph isomorphism problems in polynomial time. They show that the outcome of their 2-dimensional stabilization procedure is the basis of a so-called *coherent algebra*, a notion which also has been introduced and investigated independently by D. G. Higman[37].

Nowadays, coherent algebras are well studied objects in algebraic graph theory and have many applications in various areas. Recently, results on landscape and recombination graphs [67, 69], and recognition of circulant graphs [55] have been obtained by exploiting features of their coherent algebras.

Furthermore, stabilization procedures can be used in a polyhedral approach to the graph isomorphism problem, an approach introduced by G. Tinhofer [70, 71, 22].

Higher dimensional stabilization procedures have been introduced by several authors [73, 23, 42, 14].

The 2-dimensional stabilization algorithm can be implemented to run in time $O(n^3 \log(n))$ (see [42]). The problem is that a straightforward implementation needs $O(n^3)$ space. In this work, an algorithm is presented which reduces the space requirements to $O(n^2)$. An implementation based on this algorithm is very efficient in practice. Furthermore, the ideas used in the 2-dimensional case are applied also to the 1-dimensional case and to a new k-dimensional stabilization algorithm.

The stabilization algorithms have several applications in chemistry, for example for recognizing the symmetries and the structure of chemical compounds. A new application is the reconstruction of phylogenies in chemical biology [11]. In this context, the cells of configuration graphs are determined. These graphs have been introduced by D. F. Robinson [59].

This thesis is structured as follows. Chapter 2 presents the basic notions and

elementary versions of the stabilization algorithms.

Chapter 3 introduces an efficient way of computing 1- and 2-stable partitions. Moreover, a particular version of a k-dimensional stabilization algorithm is presented.

In Chapter 4, several aspects of stabilization procedures are considered. We discuss bounds on the number of steps and capabilites of k-dimensional stabilization algorithms. In addition, the main invariants obtained by stabilization procedures are summarized. This chapter ends with an introduction to pointed k-dimensional stabilization algorithms.

Chapter 5 deals with applications. A class of graphs is investigated which is of importance for the problem of reconstructing phylogenies. Other applications concern graph isomorphism, automorphism and canonical labeling problems. Then some graph classes are presented for which the isomorphism problem is solvable in polynomial time. Finally, we exhibit and discuss some algorithms used in chemistry.

In Chapter 6, some refinements of the algorithms and computational results are given.

Chapter 2

Basics

First, we need to fix some notations and to introduce the basic concepts. Here, elementary definitions are given, the fundamental notions of *equitable partitions* and *coherent algebras* are presented, and the more general concept of *k-colorings* is introduced. Afterwards, the algorithmic aspects of the concepts just mentioned are discussed.

2.1 Definitions

2.1.1 Graphs

Let $G = (V, E)$ be a (*directed*) *graph* with *vertex set* $V = V_n := \{v_1, v_2, \ldots, v_n\}$ and *edge set* $E \subseteq V \times V \setminus \{(v, v) \mid v \in V\}$ with $m := |E|$. The number of vertices n is called the *size* or *order* of G.

Two vertices v and w are *adjacent* if and only if $(v, w) \in E$ or $(w, v) \in E$. In this case, v and w are *neighbors*. We say a vertex u is *incident* with an edge $e = (v, w)$ if $u = v$ or $u = w$. We call v the *tail* and w the *head* of e.

An edge $(u, v) \in E$ is *undirected* if also $(v, u) \in E$. Undirected edges are denoted by $[v, w]$. An *undirected graph* is a graph where the adjacency relation is symmetric, i.e., $(v, w) \in E \Leftrightarrow (w, v) \in E$, and thus, all edges in E are undirected.

Figure 2.1: The drawing of a directed edge between the vertices v_1 and v_2 and of an undirected edge between v_3 and v_4.

The *outdegree* of a vertex v is the number of edges e such that v is the tail of e. Similarly, the *indegree* of v is the number of edges for which v is the head. The indegree and the outdegree of a vertex v are denoted by $\mathrm{indeg}(v)$ and $\mathrm{outdeg}(v)$, respectively. If G is undirected, the *degree* of a vertex is the number of vertices adjacent to v. An undirected graph is k-regular if all vertices have degree k. If the

specific value of k is of no importance, we simply speak of *regular* (undirected) graphs.

The *subgraph* of a graph $G = (V, E)$ *induced* by a vertex set V' or an edge set E' is the graph

$$G(V') := (V', E_{V'}), \quad E_{V'} := \{(u, v) \in E \mid u, v \in V'\},$$

or

$$G(E') := (V_{E'}, E'), \quad V_{E'} := \{v \in V \mid \exists u \in V : (u, v) \in E' \text{ or } (v, u) \in E'\},$$

respectively.

A list of vertices $p = (u_1, u_2, \ldots, u_t)$ is called a *path* of *length* $t - 1$ if

$$\forall i \in \{1, 2, \ldots, t-1\} : \ (u_i, u_{i+1}) \in E.$$

A path is called *simple* if

$$\forall i, j \in \{1, 2, \ldots, t\}, i \neq j : \ u_i \neq u_j.$$

A path $(u_1, u_2, \ldots, u_t)$, $t \geq 3$, such that $(u_1, u_2, \ldots, u_{t-1})$ is simple and $u_1 = u_t$ is called a *cycle* of length $t - 1$.

A graph is called *strongly connected* if

$$\forall v, w \in V \, \exists \text{path } p : \ p = (v = u_1, u_2, \ldots, u_t = w).$$

An undirected graph is *connected* if and only if it is strongly connected [20].

A *forest* is an undirected graph without cycles and a connected forest is a *tree*.

Given a graph G, let $dist(u, v)$ denote the distance from u to v, i.e., the length of a shortest path from u to v in G. The maximum distance of two vertices is called the *diameter* and is denoted by $\text{diam}(G)$.

Let $A(G)$ denote the *adjacency matrix* of G, i.e.,

$$A(G) = (a_{ij})_{i,j \in \{1,2,\ldots,n\}} \in \{0, 1\}^{n \times n},$$

and $a_{ij} = 1$ if $(v_i, v_j) \in E$ and 0 otherwise. See **Figure 2.2** for an example. Define

$$\overline{V} := \{(v, v) \mid v \in V\} \text{ and } \overline{E} := E \cup \overline{V}.$$

2.1.2 Colored Graphs

A *colored graph* $G_f = (V, E, f)$ is a graph together with a coloring f. A *coloring* is a function $f : \mathcal{D}_f \longrightarrow \{1, 2, \ldots, n^2\}$, $\mathcal{D}_f \subseteq V \times V$. The color of a vertex v is by definition the color of (v, v), i.e., we put $f(v) := f((v, v))$. We write $f(u, v)$ instead of $f((u, v))$. If it is clear from the context which f belongs to G, we

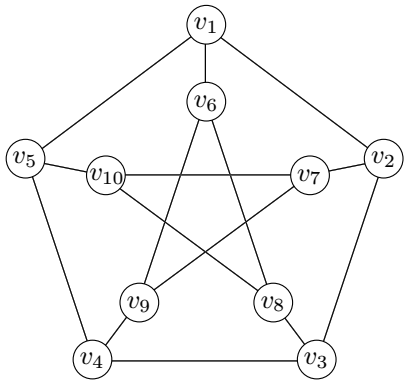
sometimes omit f and write simply G instead of G_f. A coloring with $\mathcal{D}_f = \overline{V}$ is a *vertex-coloring*, one with $\mathcal{D}_f = \overline{E}$ an *edge-coloring*, and a *complete coloring* is one with $\mathcal{D}_f = V \times V$. In the latter case, G_f is called *completely colored*. Define $\mathcal{F}_f := f(\mathcal{D}_f)$ and let $r_f := |\mathcal{F}_f|$ be the *rank* of G_f.

For a set $F \subseteq V \times V$, we define $F^t := \{(v,u) \mid (u,v) \in F\}$. Instead of $\{e\}^t$, we write e^t. The elements $(v,w) \in V \times V \setminus \overline{E}$ are the *non-edges* of G.

Let f, f' be two colorings of a graph G with $\mathcal{D}_f = \mathcal{D}_{f'}$. We say that f is *coarser* than f' ($f \preceq f'$) or equivalently, f' is *finer* than f, if

$$\forall (u,v),(w,z) \in \mathcal{D}_f : \; f'(u,v) = f'(w,z) \Rightarrow f(u,v) = f(w,z).$$

Two colorings f and f' are *equivalent*, denoted as $f \simeq f'$, if $f \preceq f'$ and $f' \preceq f$. Furthermore, two vertices u, v of G are *distinguished* by f if $f(u) \neq f(v)$.



(a) A drawing ...

$$\begin{pmatrix}
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0
\end{pmatrix}$$

(b) ... and its adjacency matrix

Figure 2.2: The Petersen graph

2.1.3 Isomorphism and Automorphism

Two graphs $G = (V, E)$ and $H = (W, F)$ are *isomorphic* if and only if there exists a bijection $\phi : V \to W$ such that

$$\forall u, v \in V : (u, v) \in E \Leftrightarrow (\phi(u), \phi(v)) \in F$$

holds. ϕ is called an *isomorphism* between G and H.

Similarly, two colored graphs G_f and H_g are *isomorphic as colored graphs* if they are isomorphic and there exists an isomorphism $\phi : V \to W$ such that

$$\forall (u, v) \in \mathcal{D}_f : f(u, v) = g(\phi(u), \phi(v))$$

holds. ϕ is called *automorphism* if $G = H$ or $G_f = H_g$, respectively. The *automorphism partition* $\mathcal{V}$ of the graph G is a partition of the vertex set, i.e.,

$$\mathcal{V} := \{V_1, V_2, \ldots, V_r\},$$

$$\forall i, j : V_i \cap V_j = \emptyset \Leftrightarrow i \neq j,$$

$$V = \bigcup_{i \in \{1,2,\ldots,r\}} V_i,$$

with the property that two vertices u, v belong to the same set V_i if and only if there exists an automorphism which maps u onto v.

A bijection $\rho : V \to \{1, 2, \ldots, n\}$ is a *labeling* of the vertex set in the following sense. It transfers a graph G_f into a graph

$$(\rho(V), \{(\rho(v_i), \rho(v_j)) \mid (v_i, v_j) \in E\}, f \circ \rho^{-1}).$$

This graph will be denoted by $\rho(G_f)$ and is called a *labeled graph*. By setting $\rho = \mathrm{id}$, every graph with vertex set $\{v_1, v_2, \ldots, v_n\}$ can be considered as a labeled graph.

Note that the set of automorphisms of a graph G_f forms a group, the so called *automorphism group* of G_f. A graph G is *vertex-transitive* if the automorphism group acts transitively on the vertices of G, i.e., for every two vertices u, v there exists an automorphism which maps u onto v.

2.1.4 Colorings

For the illustration of the algorithms in this treatise it is preferable to think of colorings of graphs instead of partitions. A vertex coloring f induces a partition $\mathcal{V} = \{V_1, V_2, \ldots, V_{r_f}\}$ of the vertex set in the following way.

$$\forall v \in V : v \in V_c \Leftrightarrow f(v) = c.$$

Analogously, an edge coloring f defines a partition of the edge set and a complete coloring defines a partition of $V \times V$. On the other hand, each partition defines a coloring. For example, a partition $\mathcal{E} = \{E_1, E_2, \dots E_{r_f}\}$ of E defines the coloring

$$f(e) := c \Leftrightarrow e \in E_c,\ c \in \{1, 2, \dots, r_f\}.$$

Due to this correspondence, all notions defined for colorings are applicable to partitions as well and vice versa. Therefore, partitions and colorings are considered as synonyms for the same combinatorial object.

To address all edges having color c, we define a *color-class*

$$\mathcal{C}(c) := f^{-1}(c) = \{e \in V \times V \mid f(e) = c\}.$$

For a compact statement of the algorithms in this thesis, it is useful to require some properties of colorings. We assume w.l.o.g. that the coloring f fulfills the conditions

$$\forall c\, \exists \overline{c} :\ \mathcal{C}(c)^t = \mathcal{C}(\overline{c}). \tag{2.1}$$

$$\forall v \in \overline{V} \cap D_f,\ e \in (V \times V \setminus \overline{V}) \cap D_f :\ f(v) \neq f(e). \tag{2.2}$$

If f does not have this property, we refine f appropriately. A complete coloring fulfilling (2.1) and (2.2) is called *proper*. See Section 3.4.1 for an algorithm for computing the coarsest proper coloring from an arbitrary complete coloring. Observe that every vertex-coloring fulfills (2.1) and (2.2).

If G_f is a completely colored graph, we define the *color matrix* of G_f

$$C(G_f) = (c_{ij})_{i,j \in \{1,2,\dots,n\}} \in \mathbb{N}^{n \times n}, \quad c_{ij} := f(i,j).$$

If an uncolored graph is given, an initial complete coloring can be defined by

$$f_{int}(e) = \begin{cases} 1, & \forall e = (u,u) \\ 2, & \forall e \in E \\ 3, & \text{otherwise.} \end{cases}$$

To complete a coloring of a partially colored graph G_f, we assign to the uncolored vertices, uncolored edges and uncolored non-edges, respectively, a new color. The algorithm for obtaining this coloring is as follows:

$f' \equiv f$;
$currentColor := r_f + 1$;
if $\overline{V} \not\subseteq \mathcal{D}_{f'}$ **then**
  $\forall (v, v) \notin \mathcal{D}_{f'} : f'(v) := currentColor$;
  $currentColor++$;
end
if $E \not\subseteq \mathcal{D}_{f'}$ **then**
  $\forall (v, w) \in E \setminus \mathcal{D}_{f'} : f'(v, w) := currentColor$;
  $currentColor++$;
end
$\forall (v, w) \in V \times V \setminus (\overline{E} \cup D_{f'}) : f'(v, w) := currentColor$;
$f \equiv f'$;

This means that first all uncolored vertices obtain the smallest unused color, then all uncolored edges obtain the next smallest unused color, and finally the uncolored non-edges get the then smallest unused color. In this way, every graph G turns into a completely colored graph preserving the initial partial coloring. If initially no coloring is given, i.e., $\mathcal{D}_f = \emptyset$, then this algorithm computes f_{int}.

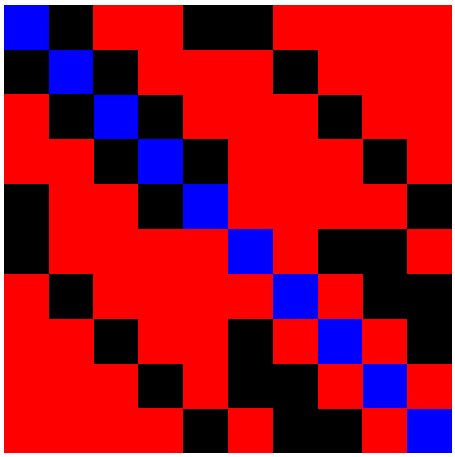


Figure 2.3: The color matrix of f_{int} for the Petersen graph (vertices are blue, edges black and the non-edges red).

2.1.5 Notation

Graphs will be denoted by G and H with vertex set V and W, respectively, and edge set E and F, respectively. By u, v and w, we always denote vertices, e denotes an edge, b and c denote colors, and f, g, and h denote colorings.

2.2 Structures

The key concepts introduced in this section are *equitable partitions* (*1-stable colorings*) and *coherent algebras* (*2-stable* or *coherent colorings*). We will state them in the classical way and in terms of colorings, and introduce the generalized view of *k-stable partitions* and *colorings*, respectively.

2.2.1 Equitable Partitions

Equitable partitions were first introduced in [63, 64, 66]. They play a crucial role in attacking the graph automorphism and the graph isomorphism problem (see Section 5.2 and [48]). For a general reference on equitable partitions see [15, 18, 33, 68].

Let G_f be a colored graph and f a vertex coloring. The integers

$$p_v^c := |\{w \in V \mid f(w) = c \text{ and } (v,w) \in E\}|,\ v \in V \text{ and } c \in \mathcal{F}_f,$$

are called the *1-dimensional structure values* of G_f, respectively, the *structure values* of the corresponding vertex partition induced by f. p_v^c is the number of successors of v with color c.

Let

$$L^1(v) := \{(c, p_v^c) \mid c \in \mathcal{F}_f, p_v^c \neq 0\}$$

be the *1-dimensional structure list of* v and

$$L^1(c) := \{(v, p_v^c) \mid v \in V, p_v^c \neq 0\}$$

be the *1-dimensional structure list of* c.

In $L^1(v)$, we collect the colors of the neighbors of v together with the number of neighbors being of the respective color. $L^1(c)$ collects vertices and their number of neighbors with color c.

A partition $\mathcal{V} = \{V_1, V_2, \ldots V_r\}$ of the vertex set is called *equitable* if

$$\forall i \in \{1, 2, \ldots, r\}\, \forall v, w \in V_i :\ L^1(v) = L^1(w).$$

Similarly, a coloring $f(v)$ of the vertex set is called *equitable* if and only if

$$\forall v, w \in V : f(v) = f(w) \Rightarrow L^1(v) = L^1(w).$$

By analogy to the term k-stable introduced later on, we frequently use the term *1-stable* instead of equitable.

If f is equitable, we introduce for each color b the *1-dimensional structure constant* p_b^c and define $p_b^c := p_v^c$, for some v with $f(v) = b$. This is well defined since for an equitable coloring $p_v^c \equiv const$ for all $v \in \mathcal{C}(b)$. These constants p_b^c, $b, c \in \mathcal{F}_f$, are called the *structure constants* of f, respectively, of the equitable partition defined by f.

The color-classes of vertices corresponding to an equitable coloring are called the *cells* of this coloring. A union of cells is called a *cellular set*. Analogously, the sets defining a vertex partition are called the *cells* of the partition.

In the following, we will define more general notions of stability. The terms cell and cellular set will always be used with respect to the currently used notion.

Some Properties

Given a graph G_f, there is a unique coarsest equitable coloring which is finer than f. If $f \equiv f_{int}$, then this coarsest equitable partition is also known as the *total degree partition*.

An important equitable partition of a graph G_f is its automorphism partition. In general, the total degree partition is coarser than the automorphism partition, in particular cases (for instance for trees) these two partition coincide.

Equitable partitions and their structure constants are convenient tools for investigating the spectra of graphs. Proofs for claims which are not proven here can be found for example in [68, 33].

The *eigenvalues* of a graph G are the solutions of the characteristic polynomial of the adjacency matrix of G, i.e., λ is an eigenvalue of G if it is a solution of the equation $det(A(G) - xI) = 0$. The spectrum of G is the set of different eigenvalues of G.

Let $\mathcal{P} = \{V_1, V_2, \dots V_k\}$ be an equitable partition of a graph G. Then the *quotient graph* $G/\mathcal{P}$ is defined to be the graph having the sets V_i as its vertices and p_i^j edges going from V_i to V_j, where the p_i^j's are the structure constants of $\mathcal{P}$. The adjacency matric of $G/\mathcal{P}$ is by definition the matrix $(p_i^j)_{1 \le i,j \le k}$.

The following lemma is well known.

Lemma 2.1 *Let $\mathcal{P}$ be an arbitrary equitable partition of G. Then*

$$\mathrm{spec}(G/\mathcal{P}) \subseteq \mathrm{spec}(G).$$

Let $\mathcal{P}_i$, $1 \le i \le k$, be a sequence of partitions of G. If for each vertex v of G the set $\{v\}$ is a cell of at least one $\mathcal{P}_i$, then this sequence is called a *complete sequence of partitions*.

Lemma 2.2 *Let $\mathcal{P}_i$, $1 \le i \le k$, be a complete sequence of equitable partitions. Then*

$$\mathrm{spec}(G) \subseteq \bigcup_{1 \le i \le k} \mathrm{spec}(G/\mathcal{P}_i)$$

holds.

We define

$$\Delta(G) := \begin{pmatrix} \mathrm{outdeg}(v_1) & 0 & \dots\dots & 0 \\ 0 & \mathrm{outdeg}(v_2) & 0 \ \dots & 0 \\ & & \ddots & \\ 0 & \dots\dots\dots\dots & 0 & \mathrm{outdeg}(v_n) \end{pmatrix}$$

and call $A(G) - \Delta(G)$ the *Laplacian* of G. Observe that if G is k-regular then λ is an eigenvalue of G if and only if $\lambda + k$ is an eigenvalue of the Laplacian of G. The Laplacian of a graph is used in Section 5.1.

2.2.2 Coherent Algebras

Coherent configurations, which are collections of relations on V having some special properties, were introduced by D. G. Higman[37]. The adjacency matrices of the relations in a coherent configuration constitute the linear basis of a so-called *coherent algebra* [38]. Coherent algebras were introduced independently under the name *cellular algebras* by B. J. Weisfeiler and A. A. Leman[73]. An important special case of coherent configurations are the so-called *association schemes* [8, 12].

Let G_f be a completely colored graph. The integers

$$p_e^{c,d} := |\{w \in V \mid f(u,w) = c,\ f(w,v) = d \text{ and } e = (u,v)\}|$$

are called the *2-dimensional structure values* of G_f. $p_e^{c,d}$ is the number of triangles with *basis edge* e whose *non-basis edges* are colored with the colors c and d.

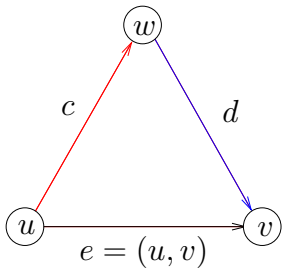


Figure 2.4: Triangles counted by $p_e^{c,d}$.

Let

$$L^2(e) := \{(c, d, p_e^{c,d}) \mid p_e^{c,d} \neq 0\}$$

be the *2-dimensional structure list of* e and

$$L^2(c, d) := \{(e, p_e^{c,d}) \mid p_e^{c,d} \neq 0\}$$

be the *2-dimensional structure list of* c *and* d.

In $L^2(e)$, the numbers of triangles which contain e as basis edge are collected, distinguished by the colors of the non-basis edges. $L^2(c, d)$ collects edges together with the respective number of triangles whose non basis edges are colored with the colors c and d.

A complete coloring f is called *2-stable* if and only if

$$\forall e, e' \in V \times V : f(e) = f(e') \Leftrightarrow L^2(e) = L^2(e').$$

If f is 2-stable, we introduce for each color b the *2-dimensional structure constant* $p_b^{c,d}$ defining $p_b^{c,d} := p_e^{c,d}$ for some e with $f(e) = b$. This is well defined since $p_e^{c,d} \equiv const$ for all $e \in \mathcal{C}(b)$. Similar to the 1-dimensional case, we define 2-stable partitions and their structure constants.

Let us define a *matrix representation* $\mathcal{B}(f)$ of a complete coloring f as $\mathcal{B}(f) := \{E_c \mid c \in \mathcal{F}_f\}$ where E_c is the $n \times n$ matrix with $(E_c)_{ij} = 1$ if $f(v_i, v_j) = c$ and 0 otherwise. Let E^* denote the conjugate transpose of E, $\mathbb{I}$ the identity matrix, and $\mathbb{J}$ the matrix which all entries are equal to 1. The representation $\mathcal{B}(f)$ is called *coherent* if it has the following properties:

$$\mathbb{J} = \sum_{c \in \mathcal{F}_f} E_c \tag{2.3}$$

$$\mathbb{I} = \sum_{c \in \mathcal{I}} E_c \text{ for some } \mathcal{I} \subset \mathcal{F}_f \tag{2.4}$$

$$\forall c \exists d : E_c = E_d^* \tag{2.5}$$

$$\forall c, d : E_c E_d = \sum_{b \in \mathcal{F}_f} p_b^{c,d} E_b \tag{2.6}$$

The following lemma states a well known fact in terms of colorings.

Lemma 2.3 *A coloring is proper and 2-stable if and only if its matrix representation fulfills (2.3)-(2.6).*

Proof. Given a coloring f, (2.3) holds if and only if f is complete. Similarly, (2.4) holds if and only if vertices have colors different from the colors of edges. Observe that (2.5) is equivalent to (2.1). Thus, f is a proper coloring if and only if $\mathcal{B}(f)$ fulfills (2.3)-(2.5).

Assume that the matrix representation of f fulfills (2.6). $(E_c E_d)_{uv}$ denotes the number $p_{(u,v)}^{c,d}$ of paths (u,w,v) with $f(u, w) = c$ and $f(w, v) = d$. Due to (2.6), this number is equal for all edges with color b, namely $p_b^{c,d}$, and this holds for arbitrarily chosen colors c and d. Thus, the coloring is 2-stable. The opposite direction holds with the same argumentation. □

Let f be a 2-stable coloring. Then $\mathcal{B}(f)$ defines the linear base of an algebraic structure, called *coherent* or *cellular algebra*, in the following way (see [29] and [38] for details).

Let $\mathcal{M}_n$ be the algebra of the complex valued $n \times n$ matrices and $A \circ B$ the *Schur-Hadamard product* $(A \circ B)_{ij} := (a_{ij} b_{ij})$ of $A = (a_{ij})$ and B=(b_{ij}). A subalgebra $\mathcal{M}_s$ of $\mathcal{M}_n$ is a subset of $\mathcal{M}_n$ with the following properties. Given two matrices $M_1, M_2 \in \mathcal{M}_s$ and a complex scalar λ. Then $M_1 + M_2 \in \mathcal{M}_s$, $M_1 M_2 \in \mathcal{M}_s$, and $\lambda M_1 \in \mathcal{M}_s$ holds.

A *coherent algebra* $\mathcal{A}$ is a sub-algebra of $\mathcal{M}_n$ which is closed under conjugate transposition and Schur-Hadamard multiplication and contains the matrices $\mathbb{I}$

and $\mathbb{J}$. The span of $\mathcal{B}(f)$ is a coherent algebra [38]. We call it the *coherent algebra corresponding to* f and denote it by $\mathcal{A}(f)$.

While a coherent algebra $\mathcal{A}$ like any matrix algebra has different linear bases, it has exactly one base consisting only of $0-1$ matrices, which is called the *standard base* of $\mathcal{A}$. For this reason, every coherent algebra is associated with some 2-stable coloring f. Obviously, $\mathcal{B}(f)$ is the standard base of $\mathcal{A}(f)$. The numbers $p_b^{c,d}$ are called *structure constants* of $\mathcal{A}(f)$.

The color-classes of a 2-stable coloring f constitute a system of relations on V which is called a *coherent configuration*. The matrices in $\mathcal{B}(f)$ are the adjacency matrices of these relations. In this way, 2-stable colorings, coherent algebras, and coherent configurations correspond to each other in a unique way.

Two completely colored 2-stable graphs G_f and $G'_{f'}$ are *equivalent* if

$$\forall i,j,k,l \in \{1,2,\dots,n\} : f(v_i,v_j) = f(v_k,v_l) \Leftrightarrow f'(v'_i,v'_j) = f'(v'_k,v'_l).$$

Given a completely colored graph G_f there is a unique coarsest 2-stable coloring $\tilde{f}$ which is finer than f. The coherent algebra corresponding to $\tilde{f}$ is called the coherent algebra generated by G_f, respectively, by the color matrix $C(G_f)$ of G_f. It is the smallest coherent algebra containing $C(G_f)$. It is an important tool for investigating the symmetries of graphs.

2.2.3 k-stable Colorings

The notions presented in the previous two sections can be generalized to the k-dimensional case, $k \in \mathbb{N}$. This has been proposed by several authors [14, 44]. In Chapter 4 different definitions of k-stability will be discussed. Instead of coloring vertices and edges like in the previous sections, k-tuples are colored.

A *k-tuple* $(t_1,t_2,\dots,t_k)$ will be denoted by t^k. A *k-starlet* S at w is an ordered k-tuple of edges $((v_1,w),(v_2,w),\dots(v_{k-1},w),(w,v_k))$. We say that S is *incident* with a k-tuple v^k of vertices if $v^k = (v_1,v_2,\dots,v_k)$. A k-tuple of vertices together with an incident k-starlet forms a *$k+1$-tangle*.

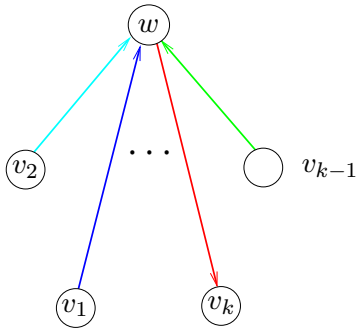


Figure 2.5: A $(k+1)$-tangle consisting of a k-tuple and an incident k-starlet

Two starlets $S = (e_1, e_2, \ldots, e_k)$ and $S' = (e'_1, e'_2, \ldots, e'_k)$ are *equally colored* if $f(e_i) = f(e'_i)$ for all $i \in \{1, 2, \ldots, k\}$.

Up to now, colors have been assigned to 1-tuples and 2-tuples of vertices only. A *k-coloring* is a partial mapping f from $\mathcal{D}_f \to \{1, 2, \ldots, n^k\}$, $\mathcal{D}_f \subseteq V^k$. Implicitly, each such k-coloring defines an l-coloring for $1 \leq l \leq k$. To make this evident, we generalize the definition of the color of vertices in the 2-dimensional case and define

$$f(u^l) := f((u_1, u_2, \ldots, u_l, \underbrace{u_l, \ldots, u_l}_{(k-l)\text{-times}})),\ u^l = ((u_1, u_2, \ldots, u_l),\ 1 \leq l \leq k.$$

Given an arbitrary k-tuple, we define the *reduced dimension* of it as the number of different vertices in the k-tuple.

For a graph G_f with a complete k-coloring f, the integers

$$p_{v^k}^{c^k} := |\{w \in V \mid \forall i \in \{1, 2, \ldots, k-1\} : f(v_i, w) = c_i \text{ and } f(w, v_k) = c_k|\}$$

are called the *k-dimensional structure values* of G_f.

Let

$$L^k(v^k) := \{(c^k, p_{v^k}^{c^k}) \mid p_{v^k}^{c^k} \neq 0\}$$

be the *k-dimensional structure list of* $v^k$and

$$L^k(c^k) := \{(v^k, p_{v^k}^{c^k}) \mid p_{v^k}^{c^k} \neq 0\}$$

be the *k-dimensional structure list of* c^k .

f is called *k-stable* if and only if

$$\forall v^k, u^k \in V^k : f(v^k) = f(u^k) \Leftrightarrow L^k(v^k) = L^k(u^k).$$

If f is k-stable, $p_{v^k}^{c^k}$ are the k-dimensional structure constants.

Note that a k-coloring f induces a partition of V^l, $1 \leq l \leq k$, and vice versa.

2.3 Stabilization Procedures

The first goal is to describe a *1-dimensional stabilization procedure* which can be used to compute the coarsest equitable partition of a graph G_f. Afterwards, *k-dimensional stabilization* algorithms for $k \geq 2$ are introduced.

2.3.1 1-dimensional Stabilization

The algorithm presented in this section computes the coarsest equitable partition of a given graph G_f.

If f is the empty coloring ($\mathcal{D}_f = \emptyset$), then the the algorithm starts by coloring the vertices according to their degrees, i.e., two vertices obtain the same color

if and only if they have the same degree. In the next step, two vertices obtain the same color if and only if they had the same color before and, for each color occurring in the graph, they have the same number of neighbors of this color. Observe that the number of colors either increases or stays the same during one step. The algorithm stops as soon as the number of colors does not increase anymore.

If the input graph is a colored graph G_f, this procedure has to be adjusted in order to respect the initial coloring. The following algorithm is stated for a general input graph G_f. If no initial coloring is given, we may use f_{int} as initial coloring.

Algorithm 1: Coarsest Equitable Partition (1-**stab**)

Data : $G_f = (V, E, f)$, f a vertex coloring

Result: A 1-stable coloring f^1 of G_f

1: $f^1 \equiv f$;
2: **repeat**
3: compute $L^1(v)\ \forall v \in V$;
4: splitcolor, i.e., $\overline{f^1}(v) = \overline{f^1}(w) :\Leftrightarrow L^1(v) = L^1(w)$ and $f^1(v) = f^1(w)$, $v, w \in V$;
5: recolor, i.e., $f^1 \equiv \overline{f^1}$;
until r_{f^1} *did not change*;

Obviously, **Algorithm 1** computes an 1-stable coloring. It is well known that, for $f = f_{int}$, the algorithm computes the total degree partition [48]. For an example see Example 1.

Lines 3-5 of **Algorithm 1** are denoted by ⟨step⟩. The corresponding lines in the algorithms 2-**stab** and k-**stab**, which will be introduced later on, are denoted by ⟨step⟩ as well. It will always be clear from the context which ⟨step⟩ is addressed. Furthermore, keywords in the algorithms which actually represent whole procedures are marked in the text by ⟨⟩. For example, when referring to line 5 in **Algorithm 1**, we write ⟨recolor⟩.

H. L. Morgan was probably the first to use the idea of vertex coloring for finding a canonical graph representation [52]. Instead of considering the neighbors of each vertex for each color separately, he looked only at the sum of the colors over all neighbors of each vertex. In our terminology, colors are natural numbers and in this way, the sum of colors is well defined. Among many others, D. G. Corneil and C. C. Gotlieb published an article [19] where they stated an algorithm in a similar manner as we did above.

We say that an algorithm **A** computes a coloring in a *canonical way*, if the resulting coloring is independent of the vertex numbering. Such a coloring is called a *canonical* coloring. An canonical coloring is an important graph isomorphism invariant.

It is easy to see that **Algorithm 1** does not compute the automorphism partition for every input graph. The most prominent class of graphs for which the total-degree partition does not coincide with the automorphism partition are the non-transitive regular graphs. In fact, for regular graphs **Algorithm 1** does not refine at all, i.e., no splitting of the vertex set appears. A simple example is given in Example 2.

Example 1

Let the graph $\mathbf{D}_6 = (V_6, E)$,

$$E = \{[v_1, v_2], [v_2, v_3], [v_3, v_4], [v_4, v_5], [v_4, v_6]\}$$

be given.

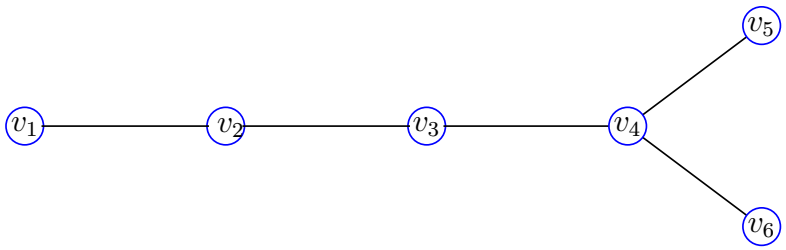
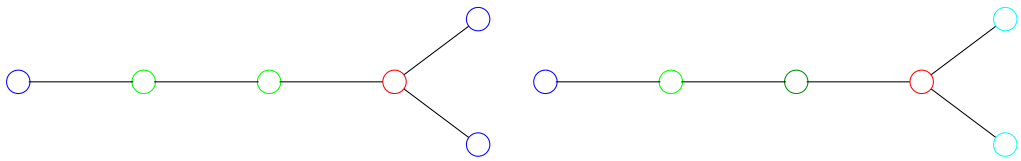


Figure 2.6: $\mathbf{D}_6$



(a) After one step (b) After the second and final step

Figure 2.7: Colorings of $\mathbf{D}_6$

See Section 6.2 for a detailed description of the graph class $\mathbf{D}_n$. **Figure 2.6** shows the graph $\mathbf{D}_6$ and **Figure 2.7** the colorings after one and two steps of **Algorithm 1**, respectively.

2.3.2 2-dimensional Stabilization

The first 2-dimensional stabilization algorithm was introduced by B. J. Weisfeiler and A. A. Leman in 1968 [73]. The algorithm can be stated as follows.

Algorithm 2: Weisfeiler-Leman algorithm (2-**stab**)

Data : $G_f = (V, E, f)$, f proper

Result: The coarsest 2-stable coloring f^2 of G_f

1: $f^2 \equiv f$;
2: **repeat**
3: compute $L^2(e)\ \forall e \in V^2$;
4: splitcolor, i.e., $\overline{f^2}(e) = \overline{f^2}(e') :\Leftrightarrow L^2(e) = L^2(e')$, $e, e' \in V^2$;
5: recolor, i.e., $f^2 \equiv \overline{f^2}$;
until r_{f^2} *did not change*;

Obviously, **Algorithm 2** computes a 2-stable coloring. Furthermore, it is well known that it computes the coarsest 2-stable coloring finer than the initial coloring of a given colored graph. For a brief historical survey of different algorithms for computing 2-stable colorings see Section 3.1.

In the 2-dimensional case, the analogue to regular graphs are strongly regular graphs, i.e., for those graphs 2-**stab** does not refine the initial coloring f_{int} at all. A graph is *strongly regular* if it is regular and there are numbers $\lambda, \mu \in \mathbb{N}$ such that two arbitrary adjacent vertices have λ common neighbors and two arbitrary non-adjacent vertices have μ common neighbors. For examples of non-vertex transitive, strongly regular graphs see [72].

Example 2

Let the graph $G = (V_8, E)$,

$$E = (\{[v_1, v_2], [v_1, v_7], [v_1, v_8], [v_2, v_3], [v_2, v_4], [v_3, v_4], [v_3, v_8], \\ [v_4, v_5], [v_5, v_6], [v_5, v_7], [v_6, v_7], [v_6, v_8]\})$$

be given.

Consider now the structure lists of $e = (v_1, v_2)$ and $e' = (v_3, v_4)$. The structure lists of f_{int} (the vertices have color 1, edges have color 2, and non-edges have color 3) are

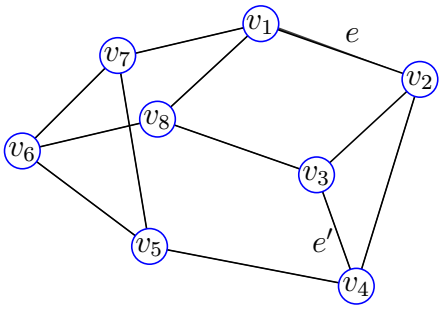
$$L^2(e) = \{(2, 3, 2), (3, 2, 2), (3, 3, 2), (1, 2, 1), (2, 1, 1)\}$$

and

$$L^2(e') = \{(2, 2, 1)(2, 3, 1), (3, 2, 1), (3, 3, 3), (1, 2, 1), (2, 1, 1)\}.$$

To make it more clear, $p_{e'}^{3,3}$ counts the paths of length 2 from v_3 to v_4 using only non-existing edges. These are the paths via the vertices v_1, v_6, v_7, Similarly, $p_e^{3,2} = |\{v_3, v_4\}|$.

A drawing is given in Figure 2.8.



(a) G

Figure 2.8:

The resulting color matrix representation after stabilization has 3 cells (diagonal elements) distinguished by three different shadings of yellow.

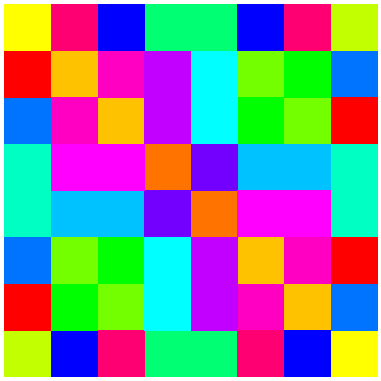


Figure 2.9: The resulting coloring

B. J. Weisfeiler and A. A. Leman state their algorithm in terms of matrix multiplication. Instead of a colored graph, they take as input a matrix containing non-commutative variables. In their language, a ⟨step⟩ is nothing but multiplying the matrix by itself and then assigning different variables to different entries in the resulting matrix. Furthermore, they refine the coloring in each step according to condition (2.1). Due to the following lemma this is not necessary and thus left out in our formulation of **Algorithm 2**.

Lemma 2.4 *If condition (2.1) is true before some ⟨step⟩ of **Algorithm 2** then it is true throughout the rest of the algorithm.*

Proof. Assume that condition (2.1) is true before ⟨step⟩ s. Let

$$L(e) = \{(c_1, d_1, p_e^{c_1,d_1}), (c_2, d_2, p_e^{c_2,d_2}), \ldots, (c_{l_e}, d_{l_e}, p_e^{c_{l_e},d_{l_e}})\}$$

be the structure list of the edge $e \in \mathcal{C}(b)$. Since condition (2.1) holds, the structure list of $e^t \in \mathcal{C}(\overline{b})$ looks as follows.

$$L(e^t) = \{(\overline{d_1}, \overline{c_1}, p_e^{c_1,d_1}), (\overline{d_2}, \overline{c_2}, p_e^{c_2,d_2}), \ldots, (\overline{d_{l_e}}, \overline{c_{l_e}}, p_e^{c_{l_e},d_{l_e}})\}$$

Thus, two edges $e, e' \in \mathcal{C}(b)$ get the same color if and only if $e^t, e'^t \in \mathcal{C}(\overline{b})$ get the same color and hence condition (2.1) is still true after $\langle \text{step} \rangle$ s. □

2.3.3 k-dimensional Stabilization

Algorithm 3: k**-stab**

Data : $G_f = (V, E, f), k \geq 2$, f a complete k-coloring

Result: The coarsest k-stable coloring f^k of G_f

1: $f^k \equiv f$;
2: **repeat**
3: compute $L^k(v^k)$ $\forall v^k \in V^k$;
4: splitcolor, i.e., $\overline{f^k}(v^k) = \overline{f^k}(w^k) :\Leftrightarrow L^k(v^k) = L^k(w^k)$ $\forall v^k, w^k \in V^k$;
5: recolor, i.e., $f^k \equiv \overline{f^k}$;
until r_{f^k} *did not change*;

If $k = 2$ and the given coloring is a proper 2-coloring, **Algorithm 3** computes a coloring equivalent to a coloring computed by the Weisfeiler-Leman algorithm. 1**-stab** does not fit in this framework since the old color of each vertex is not considered properly in each step.

A complete k-coloring f of a graph G is called *k-stable* if it does not change if **Algorithm 3** is applied to it. A graph G_f is called *k-stable* if f is a k-stable coloring.

2.4 Invariants, Canonical Colorings and Canonical Labelings

A *canonical label* is a mapping which assigns to each graph G_f a labeling ρ_{G_f} with the following property. Given two arbitrary graphs G_f and $G'_{f'}$ and their canonical labelings ρ_{G_f} and $\rho_{G'_{f'}}$, respectively, G_f and $G'_{f'}$ are isomorphic if and only if ${\rho_{G'_{f'}}}^{-1} \circ \rho_{G_f}$ defines an isomorphism from G_f to $G'_{f'}$. ρ_{G_f} is called a *canonical labeling* of G_f. Observe that a graph can have different canonical labelings.

A graph-invariant is a graph-theoretical property or parameter which is preserved by isomorphism, in other words, which does not depend on the labeling

of the graph. Examples are the number of vertices and the number of edges, the sorted sequence of degrees and so on.

To be more precise, an *l-invariant* ι is a mapping which assigns to each graph an l-tuple of numbers having the property that for any two given isomorphic graphs G and G' we have $\iota(G) = \iota(G')$. Observe that the adjacency matrix is not an invariant since it depends on how the graph is labeled. However, if we use a canonical labeling for each graph, the adjacency matric becomes an invariant (see Section 5.2.1).

An invariant ι is *complete* if two graphs G and G' are isomorphic if and only if $\iota(G) = \iota(G')$. Algorithms for computing invariants are called *invariant procedures*.

It is well known that the sets of k-dimensional final structure lists of graphs computed by the algorithms presented above define invariants [73, 48, 44]. Two graphs are called *weakly isomorphic* if they have the same final 2-dimensional structure lists.

2.5 Complexity Theory

In this section, we introduce some basic notions of complexity theory, which enable us to speak about the "difficulty" of a problem and the "efficiency" of an algorithm. Establishing the whole theoretical framework needed for a precise introduction to complexity theory would exceed the scope of this chapter. Thus we will constitute the framework on a more or less informal level. For a precise and comprehensive introduction to complexity theory, we refer to A. V. Aho, J. E. Hopcroft and J. D. Ullman [1], M. R. Garey and D. S. Johnson [32], or C. Papadimitriou and K. Steiglitz [56].

For our purposes a *problem* is a general question to be answered which is defined on several formal parameters (or variables) whose values are left open. To define a problem, we need a description of all its parameters and of the properties an (optimal) solution is required to satisfy. If all the parameters are fixed to certain values, we get an *instance* of the problem.

We consider two types of problems. *Decision problems*, which require an answer "yes" or "no", and *optimization problems* whose solutions have to minimize (or maximize) a certain objective function.

We assume that we have an encoding scheme which represents each instance of a problem and each of its solutions as a binary string of 0's and 1's.

For an informal discussion it is sufficient to consider an *algorithm* as a computer program solving a problem step by step. *Solving* a problem means, accepting a string representing an instance of the problem and giving back a solution. It is reasonable to measure the performance of an algorithm depending on the "size" of the problem instances to be solved. Therefore to each instance I of a problem, we associate a *size* or an *encoding length* which is defined as the length

$l(I)$ of the string representing the instance according to our encoding scheme. If $\mathbf{A}$ is an algorithm for the solution of a problem P, we define its *running time* for the instance I as the number of elementary steps $\mathbf{A}$ requires to get a solution of I. For our purpose we consider elementary arithmetic operations such as additions, multiplications, etc, and read/write operations as elementary steps.

Define

$$O(p(n)) := \{q(n) \mid q(n) : \mathbb{N} \to \mathbb{N}, \exists C : \forall_{n\in\mathbb{N}} : \ q(n) \leq Cp(n)\} \text{ and}$$
$$\Omega(p(n)) := \{q(n) \mid q(n) : \mathbb{N} \to \mathbb{N}, \exists c : \forall_{n\in\mathbb{N}} : \ q(n) \geq cp(n)\}.$$

The *time complexity function* of an algorithm $\mathbf{A}$ for a problem P is a function $t_{\mathbf{A}} : \mathbb{N} \to \mathbb{N}$ giving for each n the maximum running time required for the solution of an instance I with $l(I) \leq n$. $\mathbf{A}$ is a *polynomial* time algorithm, if there exists a polynomial p with $t_{\mathbf{A}}(n) \in O(p(n))$. Analogously, we define space complexity which measures the amount of space that is required to run an algorithm.

The class of all decision problems which can be solved by a polynomial time algorithm is denoted by $\mathcal{P}$. A decision problem Π is in $\mathcal{NP}$ if for each instance I of Π whose solution is "yes", there exists a structure S such that with the help of S the correctness of the "yes" solution can be checked in polynomial time. A decision problem is $\mathcal{NP}$*-complete* if it belongs to the "hardest" problems in $\mathcal{NP}$ in the following sense. If there is a polynomial time algorithm to solve an $\mathcal{NP}$-complete problem, then all problems in $\mathcal{NP}$ can be solved in polynomial time using this algorithm as a subroutine, i.e., all problems in $\mathcal{NP}$ are *polynomially time equivalent*.

Obviously $\mathcal{P} \subseteq \mathcal{NP}$. The question whether $\mathcal{P} = \mathcal{NP}$, which is widely believed to be false, is still, since 1971, one of the major open problems in complexity theory.

An optimization problem is said to be $\mathcal{NP}$*-hard*, if it has the property that the existence of a polynomial time algorithm for its solution would imply the polynomial time solvability of an $\mathcal{NP}$-complete problem. We want to point again to the fact that we kept this introduction at an informal level.

Chapter 3

Algorithmic Aspects of Stabilization Procedures

Since the motivation of the present work is to find a fast algorithm for computing 2-stable colorings, we begin with a brief survey of the existing algorithms and implementations. Then we introduce some necessary definitions and lemmas.

We present algorithms for computing k-stable colorings, $k = 1$, $k = 2$, and $k \geq 2$.

3.1 Discussion of Known Algorithms and Ideas

B. J. Weisfeiler and A. A. Leman in [73] did not give time or space bounds of their algorithms for computing 2-stable colorings. Time bounds were first considered by S. Friedland in [29]. The methods presented in both of these papers are of high theoretical but of little practical interest. They just verify the existence of polynomial time algorithms for the problem. In the last decade, possibly among others, three algorithms for computing 2-stable colorings have been developed and implemented: **stabil** by I. V. Chuvaeva, M. Klin and D. V. Pasechnik [7], **stabcol** by L. Babel and S. Baumann [5, 6] and **CC** by I. N. Ponomarenko [58].

We will first discuss these algorithms briefly and then introduce some ideas which proved useful for an improved approach to graph stabilization.

A straightforward implementation of **Algorithm 2** which just computes all structure lists in each step would have a running time of $O(n^5 \log(n))$ and $O(n^3)$ space would be needed. This is obvious since every $\langle \text{step} \rangle$ needs $O(n^3)$ time and space for computing the structure values and $O(n^3 \log(n))$ time to assign the new colors to the edges. Recoloring can be done by sorting the edges according to a lexicographical ordering of their structure lists and then assigning the new colors in this order.

Such a procedure needs $O(n^2 \log(n))$ comparisons of structure lists, each taking up to $O(n)$ time. Since theoretically $O(n^2)$ steps could be necessary, this

sums up to a running time of $O(n^5 \log(n))$.

The only possibility to reduce the running time is to compute less than n^3 triangles in each ⟨step⟩ of **Algorithm 2**. We will show how this can be done without changing the outcome of the algorithm. First, it is necessary to describe more precisely how ⟨recoloring⟩ will be done.

During ⟨recoloring⟩, a color-class $\mathcal{C}(c_0)$ will be split into l color-classes. The largest class will keep the *old color* c_0 and the other classes will get *new colors* $c_1, c_2, \ldots, c_{l-1}$. We refer to this strategy as the *LCOC rule* (Largest Class Old Color). The new colors remain *new* until the next recoloring.

The following two lemmas are stronger formulations of lemmas given by L. Babel in [5].

Lemma 3.1 *It suffices to compute only those entries in each structure list which contain at least one new color.*

Proof. Let f be the resulting coloring after the sth recoloring. Assume that the colors c_0 and d_0 have been split during the sth recoloring to $c_0, c_1, \ldots c_{l_c-1}$ and $d_0, d_1, \ldots d_{l_d-1}$. Let $e, e' \in \mathcal{C}(b)$ after the sth recoloring. Of course

$$\sum_{i=0}^{l_c-1} \sum_{j=0}^{l_d-1} p_e^{c_i,d_j} = \sum_{i=0}^{l_c-1} \sum_{j=0}^{l_d-1} p_{e'}^{c_i,d_j} \tag{3.1}$$

holds. If we compute only those entries in each structure list which contain at least one new color, then the values $p_e^{c_0,d_0}$ and $p_{e'}^{c_0,d_0}$ will not be computed. Assume that $p_e^{c_0,d_0} \neq p_{e'}^{c_0,d_0}$. Due to (3.1), there is a pair $(i,j) \neq (0,0)$ such that $p_e^{c_i,d_j} \neq p_{e'}^{c_i,d_j}$. For that reason, different colors are assigned to e and e' in the $(s+1)$th recoloring. A similar argument holds when only one color, say c_0, has been split up ($l_d = 1$). □

An edge has been *recently recolored* if it has obtained a new color in the preceding ⟨step⟩. A triangle is called *necessary* if at least one of its non-basis edges has been recently recolored. The part of a structure list which contains only structure values counting necessary triangles is called *reduced structure list*.

Although the above lemma eventually reduces the number of triangles which are considered in a ⟨step⟩, the worst case bound on the number of triangles considered in a single ⟨step⟩ is still n^3. However, the total number of triangles to be considered reduces considerably.

Lemma 3.2 *When using the LCOC rule for recoloring, the overall number of triangles which have to be considered during the entire run of the algorithm is bounded by $O(n^3 \log(n))$.*

Proof. Due to the LCOC rule, each new color-class $\mathcal{C}(c_1), \mathcal{C}(c_2), \ldots \mathcal{C}(c_{l_c-1})$ has at most half of the size of the color-class it originated from. Thus, each triangle can only be $2\log(n^2)$ times necessary. Since there are n^3 triangles in the

graph, the proof is finished. □

Due to the previous lemma, **Algorithm 2** can be implemented to run in time $O(n^3 \log(n))$ Up to now, this is the best known time bound of an 2-dimensional stabilization algorithm.

A canonical coloring can be obtained by sorting the elements of the structure lists $L(e)$ lexicographically and then assigning new colors according to the lexicographic order of the structure lists. This can be combined with the recoloring step described above to get the canonical coloring algorithm presented by L. Babel in [5]. The ideas of L. Babel were implemented in **stabcol**. The drawbacks of **stabcol** are the space requirements of $O(n^3)$ and the bad practical running time.

Another implementation of **Algorithm 2** is the program **stabil**. This implementation is very efficient in practice, has a theoretical time bound of $O(n^7)$ and needs $O(n^2)$ space. The authors of **stabil** tried to reduce the number of structure lists they have to store simultaneously. The idea is to compute only the structure lists of edges having currently the same color. To start with color-classes of small size, they perform a degree partition of the vertices and recolor edges based on this information. Since the size of the largest color-class can still be up to $\Omega(n^2)$, in a $\langle$step$\rangle$ of **stabil** only the first $O(n)$ different structure lists of a color-class are considered. Also this approach has two disadvantages. First, **stabil** does not color canonically and, secondly, **stabil** might need more steps than the generic Weisfeiler-Leman algorithm (see Section 2.3.2), although still less than $O(n^2)$ of course.

An implementation of the generic Weisfeiler-Leman algorithm, which yields a canonical coloring, is included in the package **CC** of I. N. Ponomarenko. It is basically a smart implementation of the original algorithm in [73] with the focus on using only little memory. It has a running time of $O(n^5 \log(n))$ and needs $O(n^2)$ space.

The problem is now to find an algorithm and an implementation combining the advantages of all known algorithms. It should be theoretically efficient with time complexity bound at most $O(n^3 \log(n))$, require at most $O(n^2)$ space, compute a canonical coloring and it should be fast in practice.

The main idea which leads to such an algorithm is to compute not all (reduced) structure lists but only parts of them at a time. This reduces the space requirements immediately, but some work has to be done to keep the required time bound.

B. D. McKay [48] and J. E. Hopcroft [40, 1] have presented some versions of algorithms for the 1-dimensional case. Furthermore, N. Immerman and E. Lander [42] were the first to recognize that k-stable colorings can be computed using ideas analogous to the ideas presented by J. E. Hopcroft and R. Tarjan[41]. Their approach is also used in [14] and is discussed in 4.1. In the way they state it, the algorithm needs $\Omega(n^{k+1})$ space and is thus inferior to ours which needs only

$O(kn^k)$ space.

3.2 Prerequisites

Loops like **while** and **foreach** will be processed in the current ordering of the elements. We use an in situ implementation of bucket sort. To simplify the notations, we denote colorings f^1, f^2 and f^k in this chapter by f. It is always clear from the context which coloring is used. The algorithms presented in this chapter for computing 1-stable and 2-stable colorings are implemented in the programs **qStab** and **qWeil**, respectively (see Chapter 6).

3.3 1-stable Colorings

The algorithm described in this section was obtained as a special case of the algorithm in Section 3.4. However, since this special case is much easier to state and to understand, it serves as an introduction.

Consider **Algorithm 1**. In this section, we want to find a fast implementation for lines 3-5 of that algorithm. These lines are denoted by ⟨step⟩. The idea of our approach is to compute only one entry of each structure list, but for all vertices at a time. That is, we compute the sets $L(c)$ instead of $L(v)$. So ⟨step⟩ can be reformulated as follows.

Procedure 4: step

1: $\overline{f} \equiv f$;
2: **foreach** $c \in \mathcal{N}$ **do**
3: | compute $L(c)$;
4: | splitcolor(c), i.e., split the colors in the following way:
| $\overline{f}(v) = \overline{f}(w) :\Leftrightarrow \overline{f}(v) = \overline{f}(w)$ and $p_v^c = p_w^c$, $\forall v, w \in V$;
end
5: $\mathcal{N} := \overline{f}(V) \setminus f(V)$;
6: recolor $f \equiv \overline{f}$;

Note that the structure values p_v^c are defined with respect to f and are not changed within a **foreach** loop. We refer to $\overline{f}$ as *pseudo coloring* of the vertices. $\overline{f}$ immediately leads to a set of *new colors* $\mathcal{N}$. $\mathcal{N}$ is initially defined as the set $f(V)$ and will be recomputed directly prior to every ⟨recolor⟩ operation as $\mathcal{N} := \overline{f}(V) \setminus f(V)$.

Let $S \subseteq V$ be a set of vertices. We define

$$\overline{N}(S) := \{w \in V \mid \exists v \in S : (v, w) \in E\}.$$

We write $\overline{N}(v)$ instead of $\overline{N}(\{v\})$. Observe that this definition differs from the standard definition of the neighborhood of a set of vertices S, $|S| > 1$. Furthermore, we define

$$||\overline{N}(S)|| := \sum_{v \in S} |\overline{N}(v)|.$$

We store the input graph G using adjacency lists, i.e., for every vertex, we store a list of its neighbors, and the vertices are stored in an array of size n. **Algorithm 1** with **Procedure 4** as implementation for ⟨step⟩ requires only linear space (linear in m).

We are now going to prove the time bound of $O(m \log(n))$ for **Algorithm 1** with **Procedure 4** as implementation for ⟨step⟩. First note that Lemma 3.1 and Lemma 3.2 are valid also for this version of the algorithm. It will be shown that ⟨compute $L(c)$⟩, ⟨splitcolor⟩, and ⟨recolor⟩ can be implemented to run in time $O(||\overline{N}(\mathcal{C}(c))|| + |\mathcal{C}(c)|)$, $O(||\overline{N}(\mathcal{C}(c))||)$, and $O(|L|)$, respectively. Here, $|L|$ is the list of vertices used in **Procedure 7**.

To see this, it is necessary to describe the used data structure in more detail. The color-classes are stored in an array. Each class consists of a doubly linked list of its members. Furthermore, every vertex knows its colors, i.e., $f(v)$ and $\overline{f}(v)$ is available in constant time. The structure lists $L(c)$ are also stored as doubly linked lists. This makes it possible to carry out append, delete and update operations in $O(1)$.

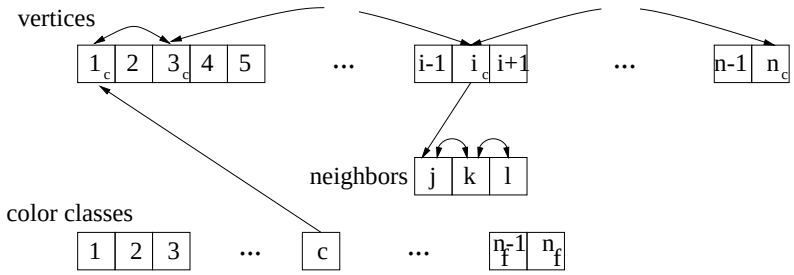


Figure 3.1: The color-class c consists of the elements $1, 3, \ldots, i, \ldots n$, and i has the neighbors j, k, l

To present a fast version of ⟨splitcolor⟩, we need some more notation. A vertex v is called *hit* by c if $p_v^c > 0$, a color-class $\mathcal{C}(b)$ is called *hit* by c if some $v \in \mathcal{C}(b)$ is hit by c. $\mathcal{C}(b).hit$ denotes the number of elements hit by c of $\mathcal{C}(b)$. This number is needed in ⟨splitcolor⟩ and can easily be computed in **Procedure 5** (which is done in line 7). $\mathcal{C}(b).size$ denotes the current size of $\mathcal{C}(b)$.

Procedure 5: compute $L(c)$

```
1: foreach w ∈ C(c) do
2:     foreach v ∈ N̄(w) do
3:         p_v^c := 0;
       end
   end
4: foreach w ∈ C(c) do
5:     foreach v ∈ N̄(w) do
6:         if p_v^c = 0 then
7:             C(f̄(v)).hit++;
8:             append (v, p_v^c) to L(c);
           end
9:         p_v^c++;
10:        update the entry of (v, p_v^c) in L(c);
       end
   end
```

We stress once more that the values p_v^c are defined with respect to the old color c. Obviously, the time for computing the structure list $L(c)$, needed in line 3 of **Procedure 4**, is bounded by $O(||\overline{N}(\mathcal{C}(c))|| + |\mathcal{C}(c)|)$ (see **Procedure 5** for details). Since each vertex is recolored at most $\log(n)$ times, the sum over the computing times of all structure lists computed during the execution of the algorithm is bounded by $O(m \log(n))$.

We now turn to the analysis of **Procedure 6** ⟨splitcolor⟩. Using bucket sort, the sorting of $L(c)$ by increasing values p_v^c (line 1) can be bounded by $O(|L(c)| + |\mathcal{C}(c)|)$. This is because the largest p_v^c is not greater than $|\mathcal{C}(c)|$.

In ⟨splitcolor⟩, the pseudo recoloring will be done in the following way. New pseudo colors are assigned according to an increasing ordering of the values p_v^c. We say that a vertex v is an element of a structure list $L(c)$, denoted by $v \in L(c)$, if there exists a tuple (v, p_v^c) in $L(c)$.

In **Procedure 6**, we determine the smallest p_v^c of each color-class $\mathcal{C}(b)$ hit by c (stored in $\mathcal{C}(b).current\_p$). We want vertices with the smallest p_v^c to keep their old pseudo color and the others to obtain new pseudo colors. These temporary pseudo colors will be reassigned in ⟨recolor⟩. Observe that if some vertices of $\mathcal{C}(b)$ are not hit by c, i.e., $p_v^c = 0$, they do not appear in $L(c)$. It is not possible to find the smallest p_v^c by scanning through all elements of $\mathcal{C}(b)$ because $\mathcal{C}(b)$ or at least the sum of the sizes of all hit color-classes might be too large to keep the time bound. One possible solution for computing the smallest p_v^c is shown inside the loop of lines $2-8$. These lines need the sizes of the pseudo color-classes which are updated in lines 13 and 15. In the loop starting in line 9, the new pseudo colors are allocated and assigned as described before. Summing up all terms in this discussion, **Procedure 6** has an (amortized) overall running time

of $O(|L(c)| + |\mathcal{C}(c)|) = O(|\overline{N}(\mathcal{C}(c))| + |\mathcal{C}(c)|)$ which equals $O(m \log(n))$.

Procedure 6: splitcolor(c)

```
1: sort L(c) by increasing values of p_v^c;
2: foreach v is first vertex in L(c) with color f̄(v) do
3:     b = f̄(v);
4:     if C(b).hit < C(b).size then
5:         C(b).current_p := 0;
       else
6:         C(b).current_p := p_v^c;
       end
7:     C(b).current_color := b;
8:     C(b).hit := 0;
   end
9: foreach v ∈ L(c) do
10:    if C(f̄(v)).current_p ≠ p_v^c then
11:        C(f̄(v)).current_p := p_v^c;
12:        C(f̄(v)).current_color := n_f̄ + 1;
       end
13:    C(f̄(v)).size − −;
14:    f̄(v) := C(f̄(v)).current_color;
15:    C(f̄(v)).size + +;
   end
```

Denote by $\mathcal{N}_b$ the set of all colors emerging from b in a specific step, i.e., $\mathcal{N}_b := \{\overline{f}(v) \mid f(v) = b\}$.

Procedure 7: recolor

```
1: Let L be the list of all vertices which got a new pseudo color;
2: Let L' := {f(v)|v ∈ L};
3: foreach v ∈ L do
4:     delete v from its color-class C(f(v));
5:     append v to C(f̄(v));
   end
6: foreach b ∈ L' do
7:     find d ∈ N_b with |C(d)| = max_{d'∈N_b} |C(d')|;
8:     if |C(d)| > |C(b)| then
9:         exchange the (pseudo) colors of the color-classes C(b) and C(d);
       end
   end
10: f ≡ f̄;
```

To finish ⟨step⟩, we have to transform the pseudo colors assigned by ⟨splitcolor⟩ into the final new coloring. This is done by **Procedure 7** which is an implementation of ⟨recolor⟩ and ensures that the largest color-class keeps its old colors and does the updating of the color-classes and colors in a correct way. This is necessary for maintaining the LCOC rule.

Computing the list L (line 1) can be done by keeping track of the new colors during each ⟨step⟩. In order to update our data structures, the vertices have to be moved from their old color-class to their new one. In our data structures, deleting an element from its color-class and appending an element to a new class takes time $O(1)$. Thus, lines 2-4 of procedure ⟨recolor⟩ take only $O(|L|)$ time. Since the sizes of the new color-classes are known, all executions of line 7 during one execution of **Procedure 7** take time $O(|L|)$. Hence, lines 6-9 take time $O(|L|)$ as well since two colors will be exchanged if and only if the color-class of the new color is larger than the old one. The final line of this procedure can be implemented in time linear in $|L|$.

We conclude that all statements of **Procedure 7** can be executed in time $O(|L|)$.

Theorem 3.3 Algorithm 1 *using procedure* **Procedure 4** *has a worst-case running time of* $O(m \log(n))$.

Theorem 3.4 *Given a graph* G_f*, the coarsest 1-stable coloring of* G *which is finer than* f *can be computed in* $O(m \log(n))$ *time and* $O(n)$ *space.*

The algorithm presented above does not compute a canonical coloring which is due to the fact that the ordering inside the structure lists L(c) depends on the ordering of the vertices. But the algorithm can easily be adjusted to compute a canonical coloring. To achieve this, sorting of L(c) by the pseudo colors of the vertices prior to line 1 of **Procedure 6** is necessary. For details see the discussion of the analogous problem in the 2-dimensional case. Using heapsort to sort $L(c)$, the overall running time of ⟨splitcolor⟩ and thus the whole algorithm is bounded by $O(m \log^2(n))$.

Corollary 3.5 *The coarsest canonical 1-stable coloring can be computed in time* $O(m \log^2(n))$ *and space* $O(n)$.

Due to the fact that there are at most $O(n \log(n))$ executions of **Procedure 6**, using bucket sort instead of heapsort gives another time bound.

Corollary 3.6 *The coarsest canonical 1-stable coloring can be computed in time* $O(n^2 \log(n))$ *and space* $O(n)$.

The latter result has also been obtained by B. D. McKay [48] for a similar algorithm. His algorithm does not compute the complete structure lists L(c) at

a time but only the parts for vertices having a specific color. This works well for 1-dimensional stabilization but the approach cannot be generalized to obtain new theoretical results for higher dimensional stabilization algorithms.

Note that Corollary 3.6 improves the time bound of B. D. McKay for sparse graphs. For a somewhat enhanced version of B. D. McKay's algorithm, one can obtain the same bound as in Corollary 3.6. In such an algorithm, the computation of structure values which are zero has do be omitted. Nevertheless, the bound of Theorem 3.4 cannot be reached without considering all neighbors of a color-class at a time.

It should be mentioned that B. D. McKay's algorithm as well as the algorithm just described needs only $O(n)$ memory in addition to the memory needed for storing the graph. The algorithm of J. E. Hopcroft and R. Tarjan which also has time bound $O(m \log(n))$ needs $O(m)$ additional space.

3.4 2-stable Colorings

In this section, the ideas presented in the previous section will be extended to obtain a new algorithm for computing 2-stable colorings.

3.4.1 Proper Colorings

We want to construct an algorithm which produces in each ⟨recoloring⟩ a proper coloring. As mentioned in Section 2.1.4, we can assume w.l.o.g. that the input graph is completely colored. To turn this given coloring into a proper coloring, we eventually have to modify it to meet the conditions (2.1) and (2.2). The following algorithm (**Algorithm 8**) computes the coarsest proper coloring finer than a given initial coloring f.

Since line 8 of **Algorithm 8** is the most expensive one, the worst-case time bound is obviously $O(n^2)$ (by using bucket sort). The algorithm requires $O(n^2)$ memory. If the graph structure is not represented by the coloring, e.g., some edges and non-edges have the same color, the above algorithm can be easily adjusted to distinguish between edges and non-edges.

There exist simple examples for the fact that if the initial coloring is not proper, the 2-dimensional stabilization algorithm not necessarily computes the basis of a coherent algebra. Take the graph with adjacency matrix $\mathbb{J}$ in which all elements of $V \times V$ have the same color. No new colors would be introduced during **Algorithm 2** if **Algorithm 8** would not be applied first. The result would violate property (2.1).

Algorithm 8: Proper Coloring

Data : Completely colored graph $G = (V, E, f)$

Result: Proper coloring $\overline{f}$ of G

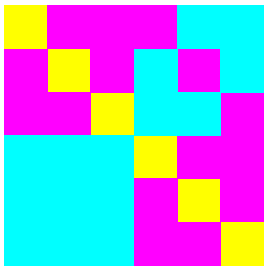
```
1:  T := ∅;
2:  foreach (u, v) ∈ V × V do
3:      if u = v then
4:          edge := 0;
        else
5:          edge := 1;
        end
6:      triple(u, v) = (edge, f(u, v), f(v, u));
7:      append triple(u, v) to T;
    end
8:  sort T lexicographically;
9:  color := 0;
10: triple = (0, 0, 0);
11: foreach triple(u, v) ∈ T do
12:     if triple ≠ triple(u, v) then
13:         triple := triple(u, v);
14:         color + +;
        end
15:     f̄(u, v) = color;
    end
```

Consider the graph with adjacency matrix depicted in Figure 3.2(a). If **Algorithm 8** is applied first, we obtain **Figure 3.2(b)**, otherwise **Figure 3.2(c)**. Obviously, **Figure 3.2(c)** violates property (2.2).

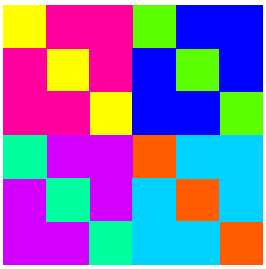
Although the necessity of starting with a proper coloring is obvious, the authors of some of the algorithms mentioned above probably were not aware of it since their algorithms fail on this instance[1]. **Figure 3.2(c)(a)** was worked out using **qWeil**.

As previously shown (see Lemma 2.4), a proper coloring stays proper throughout the whole run of **Algorithm 2**.

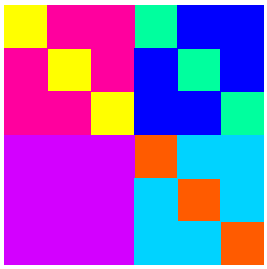
[1] **stabil** exits with a segmentation fault and **stabcol** computes **Figure 3.2(c)(a)**



(a)



(b)



(c)

Figure 3.2:

3.4.2 The New Algorithm

In this section, we will show that canonical coherent colorings can be computed in $O(n^2)$ space and $O(n^3 \log(n))$ time. As done in the 1-dimensional case, we start by considering a new $\langle$step$\rangle$ function.

Procedure 9: step

1: $\overline{f} \equiv f$;
2: **foreach** $(c, d) \in \mathcal{F} \times \mathcal{F}$ **do**
3: splitcolor(c, d), i.e., compute $L(c, d)$ and split the colors in the following way: $\overline{f}(e) = \overline{f}(e') :\Leftrightarrow \overline{f}(e) = \overline{f}(e')$ and $p_e^{c,d} = p_{e'}^{c,d}$, $\forall e, e' \in E$;
end
4: recolor, i.e., $f \equiv \overline{f}$;

As mentioned before, the entries of lists (or sets) are always visited according to the current (natural) ordering of the list (set). In **Procedure 9** we suppose entries are visited in increasing order of c and d.

$\langle$splitcolor$(c, d)\rangle$ stores a pseudo color $\overline{f}$ at each edge. This guarantees that the information of the previously computed structure values will be memorized.

Finally, ⟨recolor⟩ overwrites the color of each edge with its current pseudo color. Obviously, the result of this refinement procedure is the same as the one of lines 2 and 3 in the generic algorithm **Algorithm 2**.

Consider the lists $L(c,d)$ in **Procedure 9** again. Since every list $L(c,d)$ has length up to n^2 and there are up to n^4 such lists, this approach does not seem to be very promising at first sight. But at least, it is only necessary to store the list of one pair (c,d) at a time and still get a canonical coloring (for details see **Procedure 11**). Thus, this approach finally makes it possible to work in $O(n^2)$ space only.

From now on, we will try to reduce the running time to $O(n^3 \log(n))$ and keep the desired space bound. To reduce the number of pairs (c,d) to be considered, we can apply Lemma 3.1 and replace **Procedure 9** by **Procedure 10**. As in **Procedure 4** of Section 3.3, $\overline{f}$ is needed to define the set of new colors $\mathcal{N}$. The old colors are denoted by $\mathcal{O} := \mathcal{F} \setminus \mathcal{N}$. Both, $\mathcal{N}$ and $\mathcal{O}$, are initially, prior to the first execution of **Procedure 10**, set to $\mathcal{F}$.

Procedure 10: step

```
1: f̄ ≡ f;
2: foreach c ∈ N do
3: |   foreach d ∈ F do
4: |   |   splitcolor(c, d);
   |   end
   end
5: foreach c ∈ O do
6: |   foreach d ∈ N do
7: |   |   splitcolor(c, d);
   |   end
   end
8: N := f̄(V) \ f(V); O := F \ N;
9: recolor;
```

To make a running time analysis of **Procedure 10**, we need to go into the details of the used data structure.

We store the colored graph G as a colored matrix M, i.e., $M_{uv} = f(u,v)$, $\forall u,v \in V$, and the set of color-classes in an array of length n^2. With each color-class $\mathcal{C}(c)$, we associate doubly linked lists of a row-wise and a column-wise encoding of the edges in that color-class (these correspond to sparse matrix representations of E_c). By row-wise and column-wise encoding, respectively, we mean that the edges in the list appear ordered lexicographically by the tuple

(head, tail) and (tail, head), respectively. We denote these lists by $rowwise(c)$ and $columnwise(c)$, respectively.
Furthermore, we store doubly linked lists of the first edges in the rows and columns of each color-class. These lists are called $rows(c)$ and $columns(c)$. The lists $L(c,d)$ are stored as unordered doubly linked lists and are sorted if necessary.

It takes $O(n^2)$ time to initialize this data structure. In the following, we do not describe explicitely how and when occurring lists and variables are deleted and reset, respectively. It always should be clear from the context how and when this is done. For an edge set W, $rowindices(W)$ and $columnindices(W)$ denote the sets of row indices and column indices of W with respect to M. If $W = \{e\}$, we write $rowindex(e)$ and $columnindex(e)$ instead of $rowindices(\{e\})$ and $columnindices(\{e\})$ respectively.

To achieve the time bound claimed above, we also need to explain the implementations of ⟨splitcolor(c,d)⟩ and ⟨recolor⟩ in more detail. We have implemented these functions to run in time $O(n + \#triangles)$ and $O(n + \sum triangles)$. By $\#triangles$, we mean the number of triangles which are considered for computing $L(c,d)$, i.e., $\#triangles = \sum_{e \in L(c,d)} p_e^{c,d}$, and by $\sum triangles$, the number of triangles which are considered in the current ⟨step⟩.

An edge e is called *hit* by (c,d) if $p_e^{c,d} > 0$, a color-class $\mathcal{C}(b)$ is called *hit* by (c,d) if some $e \in \mathcal{C}(b)$ is hit by (c,d). $\mathcal{C}(b).hit$ denotes the number of hit elements of $\mathcal{C}(b)$. This number is needed in ⟨splitcolor⟩ and is computed in **Procedure 12** (in line 4).

In ⟨splitcolor⟩ (**Procedure 11**), the pseudo recoloring will be done in the following way. New pseudo colors are assigned according to an increasing ordering of $\overline{f}(e)$ and $p_e^{c,d}$ to obtain a canonical coloring. Moreover, we determine the smallest $p_e^{c,d}$ of each color-class $\mathcal{C}(b)$ hit by (c,d) (stored in $\mathcal{C}(b).current\_p$) because the edges with the smallest $p_e^{c,d}$ keep their old (pseudo) color and the other ones get new (pseudo) colors. It is not possible to do this by scanning through all elements of $\mathcal{C}(b)$ because $\mathcal{C}(b)$ or at least the sum of the sizes of all hit color-classes might be too large.
One possible solution for computing the smallest $p_e^{c,d}$ is shown in lines $4-10$. That is why we need to update the sizes of the color-classes immediately. This is done in lines 15 and 17. In lines $11-17$, the new pseudo colors are assigned as described before.

Procedure 11: splitcolor(c, d)

```
1:  compute L(c,d);
2:  sort L(c,d) by the values f̄(e);
3:  sort L(c,d) (in situ) by the values p_e^{c,d};
4:  foreach e with (e first edge in L(c,d) with color f̄(e)) do
5:      b = f̄(e);
6:      if C(b).hit < C(b).size then
7:          C(b).current_p := 0;
        else
8:          C(b).current_p := p_e^{c,d};
        end
9:      C(b).current_color := b;
10:     C(b).hit := 0;
    end
11: foreach e ∈ L(c,d) do
12:     if C(f̄(e)).current_p ≠ p_e^{c,d} then
13:         C(f̄(e)).current_p := p_e^{c,d};
14:         C(f̄(e)).current_color := n_{f̄} + 1;
        end
15:     C(f̄(e)).size − −;
16:     f̄(e) := C(f̄(e)).current_color;
17:     C(f̄(e)).size + +;
    end
```

The first line of **Procedure 11** is implemented in **Procedure 12**. It is a special sparse matrix multiplication and can be done in the required time of $O(n + \#triangles)$. To compute $L(c, d)$, the matrix product $E := E_c \cdot E_d$, which is nothing but a matrix representation of $L(c, d)$, has to be computed.

Procedure 12: compute $L(c, d)$

```
1: foreach w ∈ columnindices(c) ∩ rowindices(d) do
2:     foreach e = (u,v) with (u,w) ∈ C(c) and (w,v) ∈ C(d) do
3:         if p_e^{c,d} = 0 then
4:             C(f̄(e)).hit + +;
5:             append (e, p_e^{c,d}) to L(c,d);
           end
6:         p_e^{c,d} + +;
       end
   end
```

Observe that the number of iterations of the loop in line 1 of **Procedure**

12 which equals $|columnindices(c)) \cap rowindices(d))|$ is bounded by $\#triangles$ and the computing of $columnindices(c) \cap rowindices(d)$ can be done by scanning through the lists $columns(c)$ and $rows(d)$ whose lengths are bounded by n.

The inner loop needs an overall amortized time of $O(\#triangles)$ because together with the index w, we get the first elements with column index w and row index w of the color-classes $\mathcal{C}(c)$ and $\mathcal{C}(d)$ respectively, and have access in time $O(1)$ to the successors of the elements.

To analyze the running times of lines 2 and 3 in **Procedure 11**, we need to explain the sorting procedure we used. Let a list L of length m be given. Assume that each element of L consists of at most k numbers out of the interval $\{1, 2, \ldots, n\}$ (or of one natural number bounded by n^k), then L can be sorted using bucket sort in time $O(k(n+m))$ and space $O(m+n)$ [1].

It follows that the sorting in line 3 in **Procedure 11** can be done with bucket sort in time $O(\#triangles)$ since the $p_e^{c,d}$ in $L(c,d)$ are bounded by $\#triangles$. Line 2 needs time $O(n+\#triangles)$ and thus, this procedure has a running time of $O(n+\#triangles)$.

Since $\langle \text{splitcolor}(c,d) \rangle$ is invoked in accordance to the lexicographical order of (c,d) and the assignment of the new colors depends only on the structure values and the previous coloring, the pseudo coloring is again canonical.

To finish the implementation of $\langle \text{step} \rangle$, we have to transform the pseudo colors assigned by $\langle \text{splitcolor} \rangle$ into a new coloring according to the LCOC rule. An appropriate updating of color-classes and colors which ensures that the largest color-classes get the old colors is done by **Procedure 13**.

Denote by $\mathcal{N}_c$ the set of all colors emerging from c in one iteration of $\langle \text{step} \rangle$, i.e., $\mathcal{N}_c := \{\overline{f}(e) \mid f(e) = c\}$.

Procedure 13: recolor

1: Let L be the list of all edges which got a new pseudo color;
2: Let $L' := \{f(v)|v \in L\}$;
3: **foreach** $e \in L$ **do**
4: | delete e from its color-class $\mathcal{C}(f(e))$;
5: | append e to $\mathcal{C}(\overline{f}(e))$;
end
6: sort L by the tuples $(rowindex(e), columnindex(e))$ and generate with the help of this ordering the row encodings of the new color-classes;
7: sort L by the tuples $(columnindex(e), rowindex(e))$ and generate with the help of this ordering the column encodings of the new color-classes;
8: **foreach** $c \in L'$ **do**
9: | find $d \in \mathcal{N}_c$ with $|\mathcal{C}(d)| = \max_{d' \in \mathcal{N}_c} |\mathcal{C}(d')|$;
10: | **if** $|\mathcal{C}(d)| > |\mathcal{C}(c)|$ **then**
11: | | exchange the colors of the color-classes $\mathcal{C}(c)$ and $\mathcal{C}(d)$;
| **end**
end
12: $f \equiv \overline{f}$;

In order to update our data structures, the edges have to be moved from their old color-classes to their new ones. In our data structures, deleting an element and appending an element to a new list – without any further updating of the data structure – takes time $O(1)$ (see lines 3 and 5).

The sorting in **Procedure 13** (lines 6 and 7) can be done in time $O(n + \sum triangles)$ using bucket sort. This sorting done, the initialization of the row and column encodings is nothing but an appending procedure and thus can be done in time $O(\sum triangles)$. Lines 9-11 take time $O(\sum triangles)$ since two colors will be exchanged only if the new color is larger than the old color. We conclude that **Procedure 13** can be executed in time $O(n + \sum triangles)$.

An amortized cost analysis of the algorithm described so far, yields a worst case running time for **Procedure 10** of $O(n^5)$. The reason for this still bad time bound lies in the fact that many empty structure lists are computed and that the running time of ⟨splitcolor⟩ depends linearly on n.

To improve the time bound, we have to make sure that only triangles with a combination of colors which exist in the currently colored graph are processed. And so the multiplication will only take time $O(\#triangles)$. In addition, we move the sorting in line 2 of **Procedure 11** to **Procedure 13**. In this way, we again change the implementation of ⟨step⟩. The result is **Procedure 14**, which first computes all paths of length 2 with a color combination which actually exists in the graph and then computes the corresponding triangles.

Procedure 14: step

```
   Examine triangles whose first non-basis edge was recently
   colored;
1: f̄ ≡ f;
2: foreach c ∈ 𝒩 do
3:     ℒ(c) = ∅;
4:     foreach (u, w) ∈ columns(c) do
5:         for (v = 1; v ≤ vertices; v++) do
6:             if (w, v) is first of its color-class in the row of M corresponding
               to w then
7:                 append ((u, w), (w, v)) to the list ℒ(c);
               end
           end
       end
8:     sort ℒ(c) by the colors of the second edges;
9:     foreach d ∈ ℒ(c) do
10:        splitcolor(c, d);
       end
11:    delete ℒ(c);
   end
   the same as in loop 2 has to be done for the case when the
   second color is new;
   ...
   recolor;
```

The list $\mathcal{L}(c)$ consists of all directed paths of length 2 whereby the first edge has color c and is the first of its color-class in some column in M and the second edge is the first of its color-class in some row of M. Therefore, $\mathcal{L}(c)$ has at most $n \cdot |columns(c)|$ elements. The sorting in line 8 can be done in time proportional to $O(|\mathcal{L}(c)| + n) = O(\sum_c triangles)$. $\sum_c triangles$ denotes the number of triangles with the first edge colored by the color c to be considered in this step, which is at least as large as n. Since the sets $columnindices(c) \cap rowindices(d)$ are stored in $\mathcal{L}(c)$, the multiplication in **Procedure 12** is executed in time proportional to the number of triangles.

To move the sorting in line 2 of **Procedure 11** to **Procedure 13**, we need to introduce more data structures. With each edge e, we store its parent color $parent(e)$, i.e., the (pseudo) color-class to which e belonged before its color was changed the last time, and with each color-class $\mathcal{C}(c)$ a list of children, i.e., edges which had color c before they were recolored the last time. If a child of a color-class $\mathcal{C}(c)$ is recolored, it is deleted from the list of children of $\mathcal{C}(c)$. During the initialization, each edge e will be defined as child of $\mathcal{C}(f(e))$. For details see procedure **Procedure 15**, which differs from **Procedure 11** in that in addition

it builds up this new data structure.

Procedure 15: splitcolor(c, d)

```
1:  compute L(c,d);
2:  sort L(c,d) by the values p_e^{c,d};
3:  foreach e with e first edge in L(c,d) with color f̄(e) do
4:      b := f̄(e);
5:      if C(b).hit < C(b).size then
6:          C(b).current_p := 0;
        else
7:          C(b).current_p := p_e^{c,d};
        end
8:      C(b).current_color := b;
9:      C(b).hit := 0;
    end
10: foreach e ∈ L(c,d) do
11:     if C(f̄(e)).current_p ≠ p_e^{c,d} then
12:         C(f̄(e)).current_p := p_e^{c,d};
13:         C(f̄(e)).current_color := n_{f̄} + 1;
        end
14:     C(f̄(e)).size − −;
15:     delete e from its parent's children list;
16:     append e to the children list of C(f̄(e));
17:     parent(e) := C(f̄(e));
18:     f̄(e) := C(f̄(e)).current_color;
19:     C(f̄(e)).size + +;
    end
```

Using this parent-children relationship, we are able to keep the current coloring canonical by inserting some appropriate lines in the procedure ⟨recolor⟩ (see **Procedure 16**).

Observe that ⟨splitcolor⟩ is still implemented to run in time $O(\#triangles)$ and the worst-case running time of ⟨recolor⟩ did not change. Furthermore, the data structure for the parent-children relationship can be stored in $O(n^2)$ space.

The algorithm needs at most n^2 calls of ⟨step⟩ to compute a stable coloring. However, the overall running time of all calls of ⟨splitcolor⟩ (including **Procedure 12**) is proportional to the number of all triangles which are considered. This number is bounded by $O(n^3 \log(n))$ because each triangle is considered at most $O(\log(n))$ times. The same amount of time is needed for all executions of the remaining part of **Procedure 14**.

Procedure 16: insert in **Procedure 13** between line 1 and 2

```
1: sort L by the old colors f;
2: introduce a dummy root r;
3: foreach e ∈ L, f(parent(e)) ≤ n_f do
4: |   assign e as child of r;
5: |   assign r as parent of e;
   end
   the parent-children relationship defines a tree T on the
   color-classes in L.;
   assign new colors n_f + 1, ..., n_f̄ to the color-classes in T and to the edges
   in L by walking through T in post order (or some other well defined order);
```

Since one execution of ⟨recolor⟩ takes time $O(n + \sum triangles)$, the overall running time for this part of ⟨step⟩ is also bounded by $O(n^3 + n^3 \log(n)) = O(n^3 \log(n))$.

Hence, we finally have obtained a running time of $O(n^3 \log(n))$.

Theorem 3.7 *Given a colored graph G_f, a canonical coarsest 2-stable coloring of G which is finer than f can be computed in time $O(n^3 \log(n))$ and space $O(n^2)$.*

A complete version of the implementation presented here can be found in the appendix.

3.5 k-stable Colorings

In this section, the general case of computing k-stable colorings is described. We adopt the ideas from the 2-dimensional case to obtain an algorithm for the k-dimensional case. Instead of coloring only vertices and edges, k-tuples are colored. However, every k-coloring implicitly determines an l-coloring, $1 \leq l \leq k$ (see Section 2.2.3).

Instead of considering edges (2-tuples) and the triangles (3-tangles) in which they are contained, we consider k-tuples and the $(k+1)$-tangles in which they are contained. We know from the foregoing discussions that we need to consider only k-starlets with at least one recently colored edge.

It is possible to generalize the notion of proper colorings. Let π denote a permutation of $\{1, 2, \ldots, k\}$. A k-coloring f is *proper* if it is complete (i.e., $\mathcal{D}_f = V^k$) and for an arbitrary π and two k-tuples $v^k = (v_1, v_2, \ldots, v_k)$ and $u^k = (u_1, u_2, \ldots, u_k)$ the following holds:

$$f(v_1, v_2, \ldots, v_k) = f(u_1, u_2, \ldots, u_k) \Leftrightarrow f(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(k)}) = f(u_{\pi(1)}, u_{\pi(2)}, \ldots, u_{\pi(k)}) \quad (3.2)$$

and

$$\dim(v^k) \neq \dim(u^k) \Rightarrow f(v^k) \neq f(u^k). \tag{3.3}$$

Let v^k and u^k be two k-tuples. v^k is called a *permutation* of u^k if there exists a permutation π of $\{1, 2, \ldots, k\}$ such that

$$(v_1, v_2, \ldots, v_k) = (u_{\pi(1)}, u_{\pi(2)}, \ldots, u_{\pi(k)}).$$

We start our algorithm with a graph being properly k-colored. To obtain such a coloring, assume we are given a properly 2-colored graph. Note that k-tuples of the form $(u, u, u, \ldots, u)$ represent vertices and that k-tuples of the form $(u, v, v, \ldots, v)$ represent edges. Such k-tuples obtain the given colors of the vertices and edges, respectively. Furthermore, permutations v^k of such k-tuples u^k get the color of u^k.

Let $c_1, c_2, \ldots c_{k-2}$ be some unused colors. Color each l-dimensional k-tuple with the color c_{l-2}, $l > 2$. In this way, we get an initial k-coloring which is obviously proper. In the 2-dimensional case, we have observed that, if we start with a proper coloring, this property is maintained throughout the algorithm. The same is valid in the general case for $k > 2$ (see Section 2.3.3).

Lemma 3.8 *If the initial coloring is proper then the coloring remains proper throughout* k**-stab** *(**Algorithm 3**).*

Proof. Property (3.3) is trivially valid throughout the algorithm. Consider now the property (3.2). Assume that the coloring is proper after some step and examine what happens in the next step. Let two k-tuples of vertices v^k and u^k have the same color and dimension. Furthermore let the structure lists of v^k and u^k be denoted by L_{v^k} and L_{u^k}, respectively, and let π be an arbitrary permutation of $\{1, 2, \ldots, k\}$.

Take two arbitrary equally colored starlets S_{v^k} represented in L_{v^k} and S_{u^k} represented in L_{u^k}. Then the starlets $\pi(S_{v^k})$ and $\pi(S_{u^k})$ are equally colored and are contained in the structure list of $\pi(v^k)$ and $\pi(u^k)$, respectively. This argument can be generalized by considering the structure values of colored k-tuples (see 3.8 for details) and thus, the coloring remains proper after the next step. □

Lemma 3.8 justifies that we only consider starlets where only one direction of each edge is present. Involving the backward edges as well would not make a difference for the final coloring.

Further, this observation enables us now to simplify the discussion by working with k-stars instead of k-starlets. A *k-star* $S(w)$ at w is an ordered k-tuple of edges $((v_1, w), (v_2, w), \ldots (v_k, w))$. We say that S is *incident* to the k-tuple $(v_1, v_2, \ldots, v_k)$ of vertices.

Instead of writing down the k-dimensional algorithm completely, we will refer to the 2-dimensional case and expose the differences.

First, in an adjusted version of **Procedure 14**, we have to compute the k-stars. This is done analogously to the 2-dimensional case. We choose a new color c and a position $p \in \{1, 2, \ldots, k\}$. Then we compute k-stars having an edge of color c at position p. As before, we do not compute all of them but only the ones where the edge at position p is the first in its column having color c. The list of these k-stars is denoted by $\mathcal{S}^k(c, p)$. Assume now that we have a fixed edge ending in w. There are n^{k-1} possible ways to extend this edge to a k-star. Hence, the size of $\mathcal{S}^k(c, p)$ is bounded by $O(n^k)$.

To obtain a good time bound, we have to avoid computing k-tangles twice. The complete procedure is shown below. To state the algorithm, we need one more definition. Let L be a set of k-stars. Then $\mathcal{CP}(L) = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_l\}$ is the partition of L such that for all $S, S' \in \mathcal{S}_i$ if and only if S and S' are equally colored.

Procedure 17: step

```
1: f̄ ≡ f;
2: foreach c ∈ 𝒩 do
3:     foreach p ∈ {1, 2, ..., k} do
4:         compute k-stars(𝒮^k(c, p));
5:         sort 𝒮^k(c, p) lexicographically;
6:         foreach 𝒮 ∈ 𝒞𝒫(𝒮^k(c, p)) do
7:             splitcolor(𝒮);
           end
       end
   end
8: recolor;
```

Since $\mathcal{S}^k(c, p)$ is sorted (in line 5), the computation of $\mathcal{CP}(\mathcal{S}^k(c, p))$ is straightforward. By using bucketsort, it takes time $O(k(n + |\mathcal{S}^k(c, p)|)$. Since $|\mathcal{S}^k(c, p)|$ is in any case at least as large as n, this is the same as $O(k|\mathcal{S}^k(c, p)|)$. Let $S(w) = ((v_1, w), (v_2, w), \ldots, (v_k, w))$ be a k-star at w. The assignment $S(w)_p := u$, $u \in V$ and $p \in \{1, 2, \ldots, k\}$, means that v_p in $S(w)$ is replaced by u.

Procedure 18: compute k-stars($\mathcal{S}^k(c, p)$)

```
1: foreach (u, w) ∈ columns(c) do
2:     S(w)_p := u;
3:     recursion(S(w), p, 1);
   end
```

Procedure 19: recursion($S(w)$, p, j)

```
1: if j = k + 1 then
2:     append S(w) to S^k(c, p);
   end
3: if j = p then
4:     recursion(S(w), p, j + 1);
   end
5: if p < j then
6:     foreach v ∈ V, f(v, w) ≠ c do
7:         S(w)_j := v;
8:         recursion(S(w), p, j + 1);
       end
   end
9: if p > j then
10:    foreach v ∈ V do
11:        S(w)_j := v;
12:        recursion(S(w), p, j + 1);
       end
   end
```

From the list $\mathcal{S}^k(c,p)$, we are able to compute the remaining k-stars and the lists $L^k(c^k)$ for each c^k realized in $\mathcal{S}^k(c,p)$ (see **Procedure 12**).

It is left to show that the structure lists can be computed and sorted in the required time. The sorting can be done by bucketsort and since we have at most $O(kn^k \log(n))$ lists $\mathcal{S}^k(c,p)$ and at most $O(kn^{k+1}\log(n))$ k-stars to consider, the overall sorting time is bounded by $O(k^2 n^{k+1}\log(n))$.

Assume that all k-stars in $\mathcal{S}$ are equally colored and let $c^k(\mathcal{S})$ be this k-tuple of colors. Now the structure lists $L^k(c^k(\mathcal{S}))$ can be computed knowing $\mathcal{S}$. Apart from a reformulation of the first three lines, ⟨splitcolor⟩ (**Algorithm 20**) stays the same as in Section 3.4 except that it has an overall running time of $O(\#k\text{−stars})$.

Procedure 20: splitcolor($\mathcal{S}$)

```
1: Let c^k = c^k(S);
2: compute L^k(c^k));
3: sort L^k(c^k) by the values p_{v^k}^{c^k};
   .
   .
   .
```

The k-tuples are stored in an k-dimensional array analogously to the 2-

dimensional case. But we only need to have the row-wise and column-wise encoding of the edges and not of all k-tuples. So **Procedure 13** ⟨recolor⟩ can be adopted.

Summarizing the above discussion, we obtain the following theorem.

Theorem 3.9 *Given a colored graph G_f, a canonical coarsest k-stable coloring of G which is finer than f can be computed in time $O(k^2 n^{k+1} \log(n))$ and space $O(kn^k)$.*

Although this algorithm reduces the memory requirements by a factor of n compared to the algorithm presented in Chapter 4.1, this algorithm is still not applicable to large graphs.

Therefore, we suggest another generalization of k-stability. Since storing the colors of k-tuples is too expensive, we only color the edges but still consider k-tuples they are contained in. A *k-roof* v^k contains a k-tuple $(v_1, v_2, \ldots, v_{k-1}, v_k)$ of vertices together with all edges going from each vertex to all vertices having larger index except the edge (v_1, v_k). v^k is said to *cover* the edge (v_1, v_k).

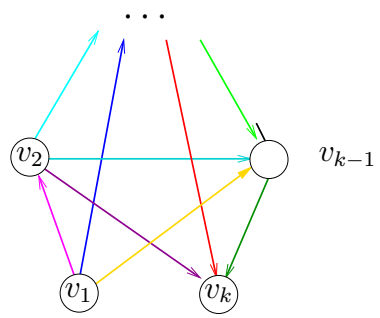


Figure 3.3: A k-roof covering (v_1, v_k)

In the suggested algorithm, one would distinguish edges by the different k-roofs each edge is covered with. A coloring which is stable with respect to this algorithm is called *loosely k-stable*. The algorithm coincides for $k \in \{1, 2\}$ with **1-stab** and **2-stab**, respectively. This simplified approach reduces the memory requirements to $O(n^2)$ but for $k > 2$ there is no known implementation which is efficient in practice (and keeps the space bound of $O(n^2)$).

3.6 Computing the Basis of a Coherent Algebra

To compute the basis of a coherent algebra of a given set of complex $n \times n$ matrices $\{A_1, ..., A_s\}$, one has to compute a linear basis first. Afterwards, the Weisfeiler-Leman algorithm can be applied. Time bounds for this procedure have been given by S. Friedland [29] with $O(sn^4 + n^{10})$ and I. N. Ponomarenko [57] $O(n^3 m^2 \log(nm) + n^5 \log(n))$.

The up to now best time bound has been achieved by L. Babel [5].

Lemma 3.10 *[5] The basis of the linear subspace* $\mathcal{L}$ $(I, J, A_1, \ldots, A_s, A_1^*, \ldots, A_s^*)$ *can be found in time* $O(sn^2 log\, n)$.

The proof is done by giving an algorithm which computes the basis of $\mathcal{L}$. Revisiting the proof in [5] shows that this is done in a space-optimal way, i.e., the algorithm needs only $O(s \cdot n^2)$ space, which is the space needed for storing m matrices of dimension $n \times n$. The result can be interpreted as a completely colored graph and thus might be used as input for the 2-dimensional stabilization algorithm presented before.

Corollary 3.11 *The basis of a coherent algebra generated by* $A_1, A_2, \ldots, A_s$ *can be computed in* $O((s+n)n^2 \log(n))$ *time and* $O(s \cdot n^2)$ *space.*

Chapter 4

Further Aspects

4.1 Another k-dimensional Approach

N. Immerman and E. Lander present in [42] a multidimensional approach to stabilization of graphs which is different from the one presented in Section 3.5. In [14], the authors refer to it as the *k-dimensional Weisfeiler-Leman algorithm*Weisfeiler-Leman algorithm!k-dimensional*.

As we will see in the following, their and our approach coincide, in the cases $k \in \{1, 2\}$, for larger k their approach can yield stronger refinements. They perform a stabilization procedure in the same way as we do, except that they define the structure lists in a different way.

In the 1-dimensional case, N. Immerman and E. Lander define their structure lists as follows:

$$\begin{aligned} \overline{L}^1(v) &:= (f(v), \{(c, y_c, n_c) \mid c \in \mathcal{F}\}), \text{ where} \\ y_c &:= |\{(v, u) \in E \mid f^1(u) = c\}| \text{ and} \\ n_c &:= |\{(v, w) \notin E \mid f^1(w) = c\}|. \end{aligned}$$

These are structure lists which contain some obvious redundant information (note that $n = y_c + n_c$ for all $c \in \mathcal{F}$). For $k \geq 2$, they define the structure lists in a different way, namely

$$\overline{L}^k(v^k) := \{f(v^k), \{l(f, v^k, u), u \in V\},$$

where

$$l(f, v^k, u) := (f(v^k(v_1/u)), f(v^k(v_2/u)), \ldots, f(v^k(v_k/u)))$$

and where for $v^k = (v_1, v_2, \ldots, v_k)$ and arbitrary vertex u $v^k(v_i/u)$ denotes the k-tuple $(v_1, v_2, \ldots, v_{i-1}, u, v_{i+1}, \ldots, v_k)$.

The algorithm of N. Immerman and E. Lander starts with an initial coloring similar to ours. They suggest to color each k-tuple according to its isomorphism

type. Starting with a proper coloring as defined in Section 3.5 would work as well and would be less costly. Here, we call the final coloring found by their algorithm *strongly k-stable*.

The authors obtain the complexity result of $O(k^2 n^{k+1} \log(n))$ time and $O(kn^{k+1})$ space for a canonical algorithm generalizing the results in [41, 1]. We refer to these algorithms as **il***(k)*.

4.2 Relations Between k-dimensional Stabilization Algorithms

It is well known that 1-**stab** computes a coarser vertex coloring than 2-**stab**. And in general, $(k+1)$-**stab** computes a finer k-coloring than k-**stab**. This can easily be seen by the following argument. If $(k+1)$-**stab** would only consider $(k+1)$-tuples of the form $(v_1, v_2, \ldots, v_k, v_k)$, then it would compute a coloring as fine as the coloring of k-**stab**.

We are now going to introduce parts of k-tuples. Consider the operation

$$v^k_{|i} := (v_1, v_2, \ldots, v_{i-1}, v_i, v_i, v_{i+2}, \ldots, v_k), \quad i \in \{1, 2, \ldots, k-1\}.$$

Define $PO(v^k, 0) := \{v^k\}$,

$$PO(v^k, p) := \{u^k \mid \exists v'^k \in PO(v^k, p-1) \exists i \in \{1, 2, \ldots, k-1\} : u^k = v'^k_{|i}\},$$

and $PO(v^k) := \bigcup_{p \in \{1,2,\ldots,\dim(v^k)\}} PO(v^k, p)$. $PO(v^k)$ is called the *set of parts* of v^k and $u^k \in PO(v^k)$ is *part* of v^k. If u^k is part of v^k then there exist $i_1, i_2, \ldots, i_p$, $p \in \{1, 2, \ldots, k\}$, such that $u^k = (\ldots((v^k_{|i_1})_{|i_2})\ldots)_{|i_p}$. We associate with each part u^k of v^k the *reduction code* of u^k with respect to v^k

$$\min\{(p, (i_1, i_2, \ldots, i_p)) \mid u^k = (\ldots((v^k_{|i_1})_{|i_2})\ldots)_{|i_p}\}.$$

Let two l dimensional k-tuples v^k and u^k be given. Two q-dimensional k-tuples v'^k and u'^k, $q < l$, being part of v^k and u^k, respectively, are called *corresponding* if the reduction codes with respect to v^k and u^k, respectively, are equal.

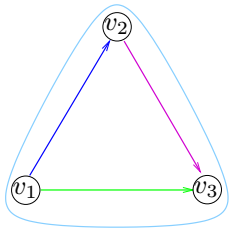
Lemma 4.1 *Let f be a k-stable coloring of an initially properly colored graph computed with k-**stab**. If two k-tuples v^k and u^k have the same color then two corresponding k-tuples v'^k and u'^k being part of v^k and u^k have the same color.*

Proof. From the definition of the initial coloring and the way of recoloring, this is clear. □

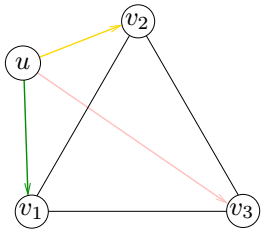
The same result holds for **il**(k) as well.

The best way to see the differences between the algorithms k-**stab** and **il**(k) is to study a picture of the structure values of each of the algorithms. We will

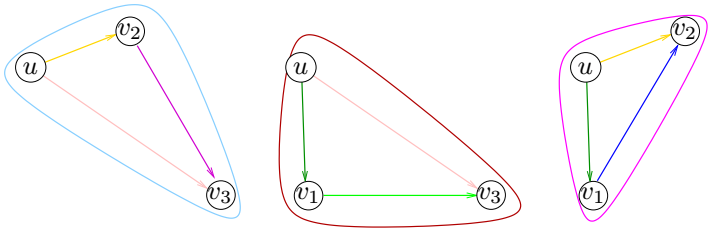
do so for the case $k = 3$. Consider an arbitrary triangle v^3 depicted in **Figure 4.1(a)** and an arbitrary vertex u.



(a) A colored triangle



(b) The star $((u, v_1), (u, v_2), (u, v_3))$



(c) The triangles (u, v_2, v_3), (v_1, u, v_3), (v_1, v_2, u)

Figure 4.1:

The outer curve illustrates the color of the triangle. While k-**stab** only considers the stars (**Figure 4.1(b)**) and thus colors of edges, **il**(k) uses the colors of the triangles (**Figure 4.1(c)**) (u, v_2, v_3), (v_1, u, v_3), (v_1, v_2, u) for the refinement.

Due to the previous theorem (two k-tuples can have the same color only if corresponding edges included in the triangles have the same colors), the following result holds.

Theorem 4.2 k-**stab***and* **il***(k) compute equivalent partitions for $i \in \{1, 2\}$ and* **il***(k) might compute finer partitions than k-***stab** *for $k > 2$.*

It was not possible to find examples were **il**(k) and k-**stab**differ for $k = 3$. Due to the amount of memory needed by **il**(k), we were able to check graphs for up to about 50 vertices only.

4.3 Pebbling Games

The games presented here are variants of games introduced by A. Ehrenfeucht[21] and R. Fraissé [28]. They have been stated in this form in [14]. The authors show

the equivalence of the second game to the k-dimensional stabilization algorithm $\mathbf{il}(k)$ presented above.

Let $G = (V, E)$ and $H = (W, F)$ be two graphs, $m, k \in \mathbb{N}$, and $P_k := \{p_1, p_2, \ldots p_k\}$ be k pairs of pebbles. Define the *m-move $\mathcal{P}_k$ game* to be a 2-player game on G and H as follows. On each move, Player I chooses an $i \in \{1, 2, \ldots, k\}$, picks up the pair p_i and places one of the pebbles belonging to p_i on one vertex of one of the graphs. Player II must then place the other pebble belonging to p_i on a vertex of the other graph. The game starts with all pebbles outside the two graphs. As we shall see later, a good Player I will in general try to put as many as possible pebbles onto the graphs. Therefore, after k-moves, if the game has not already ended, each graph will be covered by k-pebbles.

Define a *k-configuration* on a pair of graphs G, H to be a pair (v, w) of partial functions,

$$g : P_k \to V \text{ and } h : P_k \to W$$

such that the domains are equal. Thus, a k-configuration is a valid position of the $\mathcal{P}_k$ game on G and H in the following sense. If $g(p_i) = v$ and $h(p_i) = w$ then one pebble belonging to p_i is placed on $v \in V$ and the other one is placed on $w \in W$, and if $p_i \notin \text{domain}(g) = \text{domain}(h)$ then the pebbles in p_i are not placed on the board. Define the mapping ϕ,

$$\phi : \text{image}(g) \to \text{image}(h), \quad g(p_i) \to h(p_i).$$

Let (g_r, h_r) be the configuration of the game after move number r. Player I *wins* after move r if ϕ does not define an isomorphism between $G(\text{image}(g_r))$ and $H(\text{image}(h_r))$. Player I wins the m-move $\mathcal{P}_k$ game if he wins after move r, for some $r \in \{1, 2, \ldots, m\}$. Player II wins if Player I does not win. We say that Player II has a winning strategy for the $\mathcal{P}_k$ game on G and H if, for all m, Player II has a winning strategy for the m-move $\mathcal{P}_k$ game on G and H.

Thus, Player II has a winning strategy if he can always find matching vertices to preserve the isomorphism. Player I is trying to point out the difference between the two graphs and Player II is trying to keep them looking the same.

A modification of the $\mathcal{P}_k$ game is the $\mathcal{P}_k^c$ game. As before, there are two players and k pairs of pebbles. The difference is that each move has two parts.

1. Player I picks up the pair p_i for some i. Then he chooses a set A of vertices from one of the graphs. Now, Player II answers with a set B of vertices from the other graph. B must have the same cardinality as A.

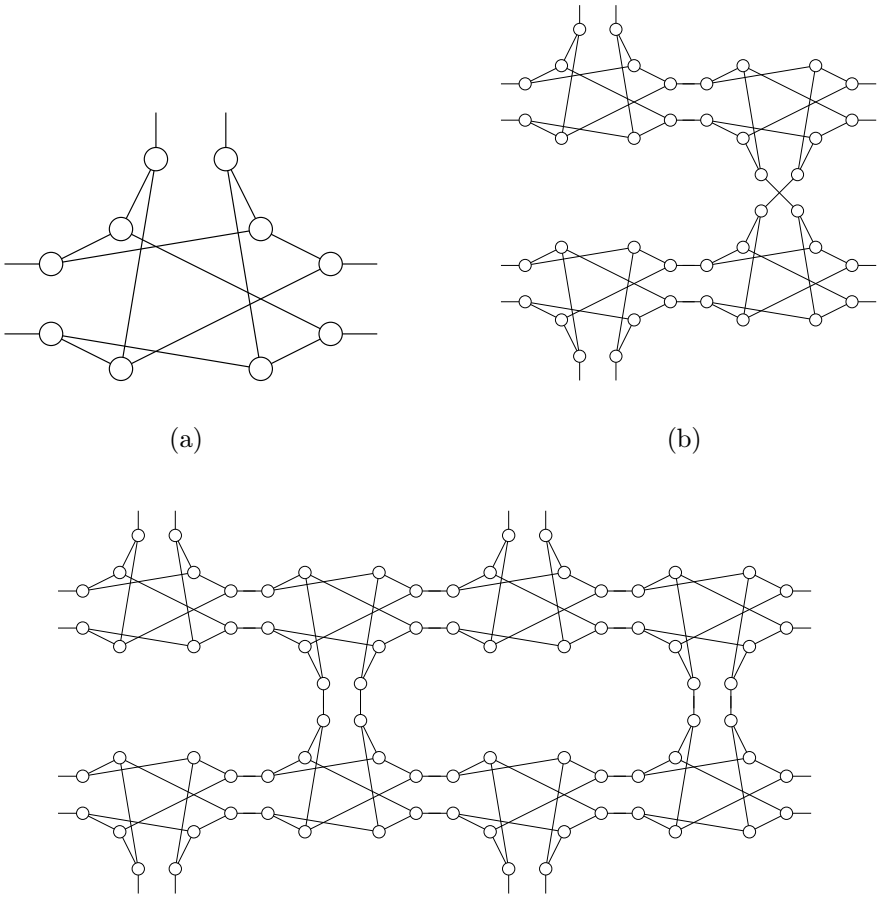
2. Player I places one of the p_i pebbles on some vertex $u_B \in B$ and Player II answers by placing the other p_i pebble on some $u_A \in A$.

The definition of winning is as before. J.-Y. Cai, M. Fürer, and N. Immerman [14] have shown that Player II has a winning strategy for the $\mathcal{P}_k^c$ game on G and H if G and H have the same set of k-dimensional structure constants with respect to $\mathbf{il}(k)$.

4.4 k-dimensional Stabilization Does Not Suffice

M. Fürer[30] and J.-Y. Cai, M. Fürer, and N. Immerman [14] showed that there exist graphs which have equivalent (strongly) k-stable colorings and are not isomorphic.

M. Fürer[30] gives an explicit construction of a series of pairs of graphs with the above property. The graphs are made of the component depicted in **Figure 4.2(a)**.



(a) (b)

(c)

Figure 4.2: The pieces of the graph of M. Fürerand F_2^1

F_m^1 consists of $2m^2$ copies of **Figure 4.2(a)**. We begin with putting $2m$ copies side by side to form a band and then putting m copies of this band on top of each other. The single components are glued together as depicted in **Figure 4.2(c)**

for the graph F_2^1 and in **Figure 4.3** for the graph F_4^1. The edges on the right are identified with the corresponding edges on the left and the edges on top are identified with the edges at the bottom.

F_m^2 is constructed in almost the same way with the difference that at exactly one arbitrary position in the graph, a parallel pair of edges linking two different basic graphs is replaced by a pair of crossing edges **Figure 4.2(b)**. In **Figure 4.3** this is indicated by dotted lines.

M. Fürer shows that F_k^1 and F_k^2 are not isomorphic and that $\mathbf{il}(k)$ computes the same set of k-dimensional structure lists for F_k^1 and F_k^2.

In [14] J.-Y. Cai, M. Fürer, and N. Immerman give an alternative proof of the fact that there exists no $k \in \mathbb{N}$ such that two arbitrary graphs are isomorphic if and only if they have the same k-dimensional structure constants. The authors use the interrelation between the functioning of the algorithm $\mathbf{il}(k)$, the notion of expressibility in special first order logic languages and the winning strategies of pebble games introduced in Section 4.3.

A *separator* of a graph is a minimal set of vertices such that the graph is disconnected. The main property of the graphs they need in order to prove the result is that they have a large separator, which applies to the graphs of M. Fürer as well.

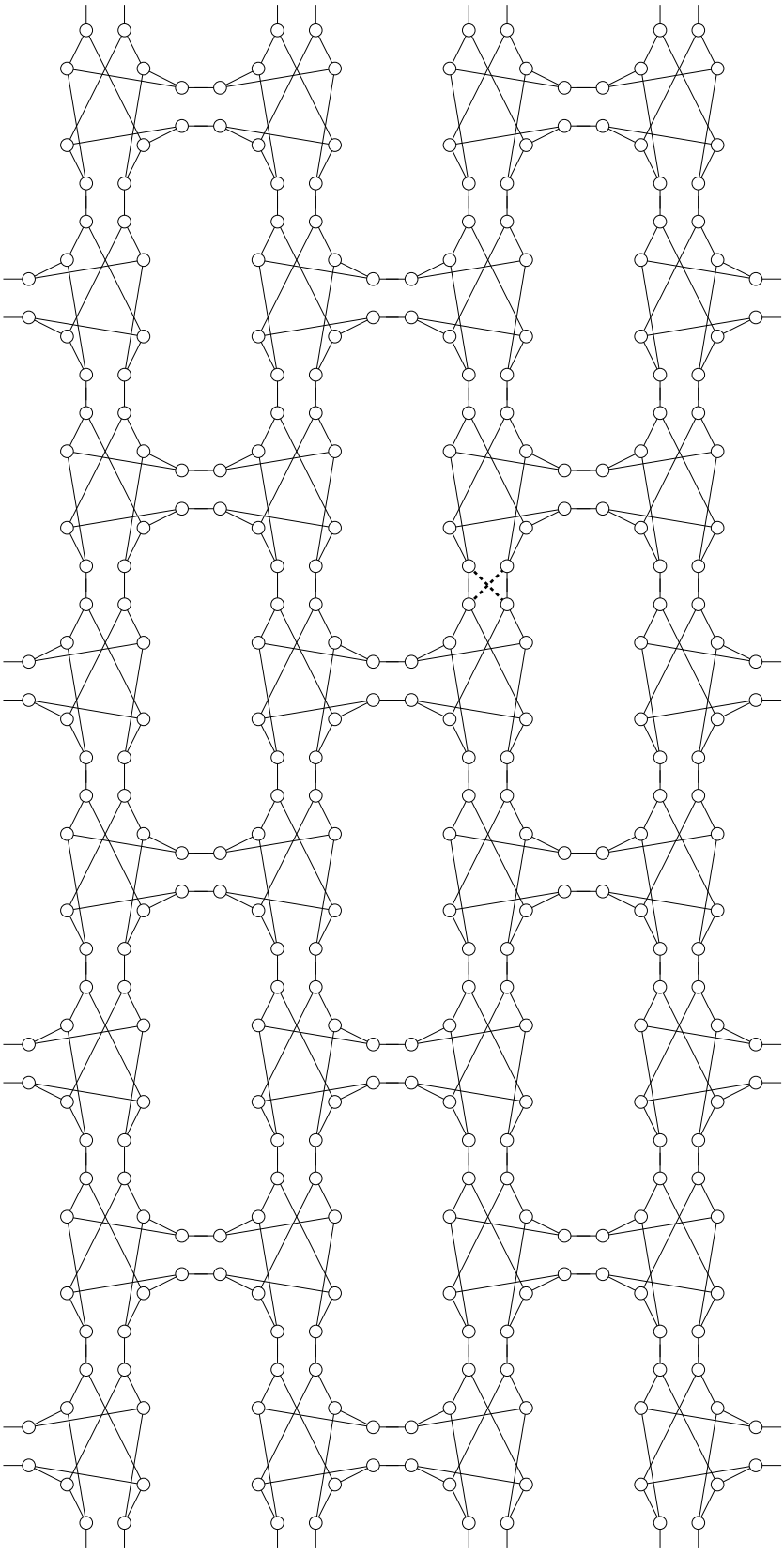


Figure 4.3: The graph F_4^1

4.5 Upper and Lower Bounds on the Number of Steps

A trivial upper bound on the number of steps for a k-dimensional stabilization procedure is n^k, since the number of colors is bounded by n^k and in every step at least one new color has to be introduced.

For the 1-dimensional case, examples which need $\Omega(n)$ steps can be easily constructed. We consider the path P_n on n nodes. In the first step, only the end vertices get colors different from the other nodes. In the next step, their neighbors obtain different colors, but all the others remain in the same color-class. After $\lceil \frac{n}{2} \rceil$ steps, the algorithm stops with a 1-stable coloring having $\lceil \frac{n}{2} \rceil$ colors.

4.5.1 A Lower Bound on the Number of Steps for $k = 2$

In the 2-dimensional case, it is not easy to determine good bounds. M. Fürer [31] has contructed a sequence of graphs ${\mathcal{F}^s}_n$ for which he could show that the Weisfeiler-Leman algorithm needs $\Omega(n)$ steps to compute a 2-stable coloring. His graphs consists of several copies of the simple basic graph depicted in **Figure 4.4** which are glued together as shown in **Figure 4.5(a)-(c)**.

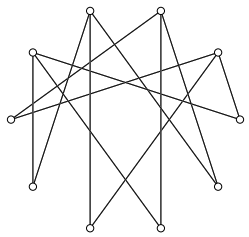
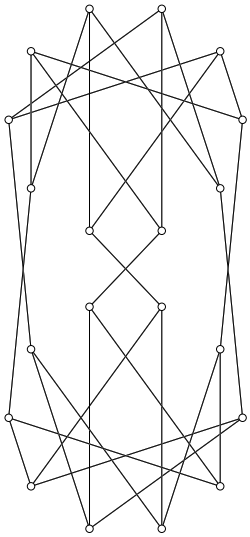
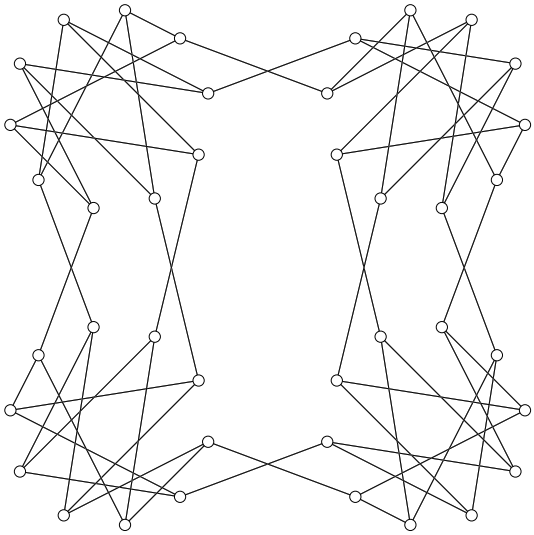


Figure 4.4: The component of which ${\mathcal{F}^s}_n$ is made of

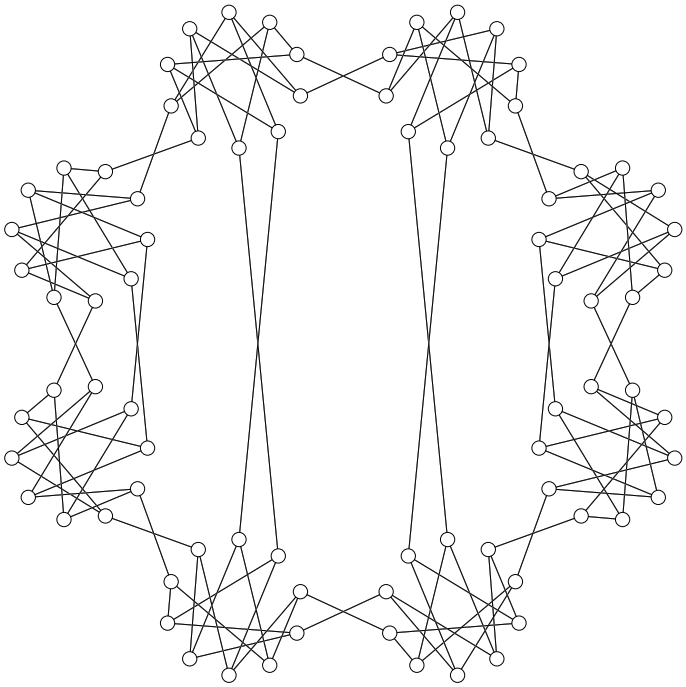
In Table 4.1 the number of steps needed to obtain a 2-stable coloring for ${\mathcal{F}^s}_n$ for small n and the computing times of **qWeil** on these instances are displayed.



(a) $\mathcal{F}^s{}_1$



(b) $\mathcal{F}^s{}_2$



(c) $\mathcal{F}^s{}_4$

Figure 4.5: Some instances $\mathcal{F}^s{}_n$

Vertices	Steps	time	Vertices	Steps	time
20	3	0	240	12	10.77
40	4	0.05	280	13	18.85
60	4	0.16	320	15	29.34
80	5	0.4	360	17	43.62
100	6	0.67	400	19	56.05
120	6	1.24	440	24	68.59
140	6	1.94	480	26	93.79
160	8	2.74	520	28	117.83
180	8	4.11	560	30	158.81
200	10	5.77	600	32	198.32

Table 4.1: The number of steps needed to obtain a 2-stable coloring for ${\mathcal{F}^s}_n$

4.6 Classical Invariants

In this section, we discuss two important invariants and their relations to k-dimensional stabilization procedures.

4.6.1 The Spectrum

As mentioned before, the adjacency matrix $A(G)$ of a graph G is not an invariant. However, the determinant $|A(G) - \lambda I|$ is not altered if the graph is relabeled (which is nothing but permuting the rows and columns of $A(G)$ in the same way), and hence is a graph invariant. Thus, the characteristic equation of $A(G)$ is also a graph invariant, so is the set of its roots (i.e., the *spectrum* of the graph).

Two graphs are *cospectral* if they have the same characteristic polynomial. Certainly, the cospectrality does not completely determine the isomorphism type of a graph. This was first pointed out by L. Collatz and U. Sinogowitz [16], by exhibiting two non-isomorphic cospectral trees (see [34, 36]).

The two graphs depicted in **Figure 4.6** have the same characteristic polynomial, namely

$$4\,\lambda - 2\,\lambda^2 - 22\,\lambda^3 + 25\,\lambda^5 - 9\,\lambda^7 + \lambda^9$$

(a) (b)

Figure 4.6: Two cospectral trees

Although the spectrum is frequently used in many algorithms for detecting the isomorphism type of graphs, it should be stressed that it is less meaningful than the set of 2-dimensional structure constants. In the above case even the structure constants of 1-dimensional stabilization suffice to fix the isomorphism type of the graphs.

In contrast to the 2-dimensional case, the set of structure constants of 1-dimensional stabilization is in general not a stronger invariant than the characteristic polynomial. Consider for example the graph G consisting of 2 cycles of length three and the graph G' being the cycle of length 6. For these graphs the set of 1-dimensional structure constants reduces to a single number, the degree of regularity, which is the same for both graphs, but they are not cospectral.

4.6.2 The Powers of the Adjacency Matrix

Other frequently used invariants are derived from the powers $A(G)^k$ of the adjacency matrix $A(G)$ of the colored graph G. It is well known, that the entry $A(G)^k_{ij}$ denotes the number of (not necessarily simple) paths of lengths k from v_i to v_j in G. Let $P = (v_0, v_1, \ldots, v_l)$ be a path in G. Then we define $\gamma(P) := (f(v_0, v_1), f(v_1, v_2), \ldots, f(v_{k-1}, v_k))$ and the multi-set

$$P^l(u, v) := \{\gamma(P) \mid P \text{ is a colored path of length } l \text{ from } u \text{ to } v\}.$$

Further define the set of multi-sets

$$\mathcal{P}_{(u,v)} := \{P^l(u, v) \mid l > 0\}.$$

An invariant of G is given by $\mathcal{L}^p(G) := \{\mathcal{P}_{(u,v)} \mid u, v \in V\}$. We will see that a edge partition based on $\mathcal{P}_{(u,v)}$ is coarser than a 2-stable partition.

Theorem 4.3 *[73] Let f be a 2-stable coloring of G. If $f(u, v) = f(u', v')$ then $P^l(u, v) = P^l(u', v')$ for all $l \geq 1$.*

Proof. The proof is by induction on l. For $l = 1$ there is nothing to prove and for $l = 2$ this is just the definition of 2-stable colorings. Let us assume that

$$\forall u, v, u', v' \in V : f(u, v) = f(u', v') \Rightarrow P^{l-1}(u, v) = P^{l-1}(u', v').$$

Let $\overset{multi}{\bigcup}$ denote the union of multi-sets. Consider now

$$P^l(u, v) = \overset{multi}{\bigcup_{w \in V}} \{(c^{l-1}, f(w, v)) \mid c^{l-1} \in P^{l-1}(u, w)\}.$$

By hypothesis, each of the multi-sets $P^{l-1}(u, w)$ depends only on the color of (u, w). Thus,

$$P^l(u, v) = P^l(u', v') \Leftrightarrow \bigcup_{w \in V} \{(f(u, w), f(w, v))\} = \bigcup_{w' \in V} \{(f(u', w'), f(w', v'))\}.$$

The right part of the condition is just the definition of a 2-stable coloring. □

Let d_v^i be the number of vertices with distance i from v in a graph G. Define $L^{dist}(v) := \{(i, d_v^i) | d_v^i \neq 0\}$. Use these lists to color the vertex set in the usual way. The resulting coloring is called *path stable*.

Corollary 4.4 *Every 2-stable coloring of G is path stable.*

Proof. This is clear due to Theorem 4.3. □

It is possible to generalize this approach by partitioning not only by the total number of vertices having certain distances but by the numbers of vertices of different colors at certain distances. Let $d_v^i(c)$ denote the number of vertices with color c at distance i from v in G.

Define $L^{dist}(v, c) := \{(i, c, d_v^i(c)) | d_v^i(c) \neq 0\}$. This list can be used in **Algorithm 1** instead of the former lists. We call the final coloring produced in this way *totally path stable*. Obviously, the total path partition is at least as fine as the total degree partition, since 1**-stab** only considers vertices with distance one and their colors.

But obviously the following corollary of Theorem 4.3 still holds.

Corollary 4.5 *Every 2-stable coloring of G is totally path stable.*

4.7 Pointed Graphs

The idea presented in this section is to improve the results of a stabilization procedure by starting with a certain finer initial coloring. This is done by first

constructing slightly altered graphs, applying a stabilization procedure to these graphs and then deducing an initial coloring from the results. One common way of alteration is to choose a vertex and to label it with a specific new color. Afterwards, the stabilization procedure is applied to this new colored graph. By comparing the results (structure constants) for different vertices the vertex set might be refined. Similarly, not only a single vertex but several vertices or edges might be labeled with new colors at a time. This idea was first introduced by B. J. Weisfeiler and A. A. Leman [73] and D. G. Corneil and C. C. Gotlieb [19]. A more general approach has been studied by S. Evdokimov, M. Karpinski and I. N. Ponomarenko [23] and has been elaborated by S. Evdokimov and I. N. Ponomarenko [24, 25].

To alter a colored graph $G_f = (V, E, f)$ in the described way, we define G_{f_v} as follows. Let $G_{f_v} := (V, E, f_v)$ and define f_v as

$$f_v(u) := f(u) \quad \forall u \neq v, \text{ and}$$
$$f_v(v) := r_f + 1.$$

The process of replacing f by f_v will be denoted as *pointing f at v*.

Let $\mathcal{L}_{f^k}(G)$ be the set of structure constants computed by the stabilization procedure under consideration.

Now, an improved algorithm can be described as follows.

Algorithm 21: pointed(k-**stab**,1)

foreach $v \in V$ **do**
  compute $\mathcal{L}_{f^k}(G_{f_v})$;
end
$\langle$recolor$\rangle$, i.e.,
$f(u) = f(v) :\Leftrightarrow f(u) = f(v)$ and $\mathcal{L}_{f^k}(G_{f_u}) = \mathcal{L}_{f^k}(G_{f_v})$, $u, v \in V$;
perform k-**stab** on this colored graph;

Algorithm 21 does the following. Given a graph $G_f = (V, E, f)$, for each $v \in V$ it computes G_{f_v}, applies k-**stab** to G_{f_v} and collects the invariants $\mathcal{L}_{f^k}(G_{f_v})$. Afterwards it recolors G according to $\mathcal{L}_{f^k}(G_{f_v})$ and applies k-**stab** once more to this intermediate coloring.

D. G. Corneil and C. C. Gotlieb [19] use this approach for the case $k = 1$ and an initial 1-stable coloring f.

For the following considerations, we need an auxiliary definition. A stabilization procedure is called *non-switching* if a color remains the same when its color-class is not split. W.l.o.g. we may assume that the stabilization procedures k-**stab** work in a non-switching mode and that the resulting colorings are canonical.

Let us take a look at the relations between the algorithms **pointed**(k-**stab**,1) and $(k+1)$-**stab**. Assume that we want to show that **pointed**(k-**stab**,1) is

weaker than k'-**stab**, $k' > k$. Let G_f be a given colored graph. We start with a k'-stable coloring of G_f and use it to construct a k-stable coloring of G_{f_v}.

First, we compare the cases **pointed**(1-**stab**,1) and 2-**stab**. Let f^2 be the coarsest 2-stable coloring of G_f and define

$$f_v{}^{2\to 1}(u) := f^2(v,u),\ u \in V.$$

Note that in the coloring $f_v{}^{2\to 1}$ the color of v is not used for any other vertex, since f^2 is a proper coloring.

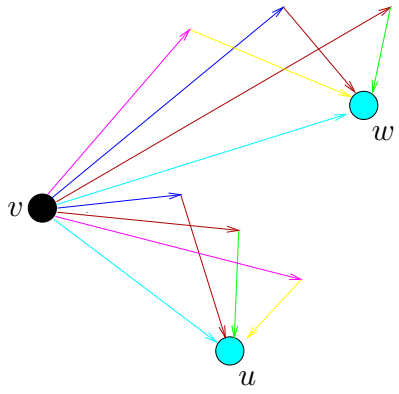


Figure 4.7: Two vertices with the same color (cyan) with respect to $f_v^{2\to 1}$ and with equally colored neighbors (blue, brown and magenta).

Lemma 4.6 *$f_v^{2\to 1}$ defines a 1-stable coloring on G.*

Proof. Since v is a color-class in $f_v^{2\to 1}$, the coloring is 1-stable with respect to v. Consider two equally colored vertices u and w, i.e., $f_v^{2\to 1}(u) = f_v^{2\to 1}(w)$. We have to show that $L^1_{f_v^{2\to 1}}(u) = L^1_{f_v^{2\to 1}}(w)$. Due to the definition of $f_v^{2\to 1}$, $f^2(v,u) = f^2(v,w)$ holds such that (v,u) and (v,w) have the same structure lists with respect to $f_v^{2\to 1}$. By inspection of these lists, we see that u and w have the same structure lists with respect to $f_v^{2\to 1}$. □

Lemma 4.7 *The vertex partition induced by $\{\mathcal{L}_{f_v^1}(G_{f_v}) \mid v \in V\}$ is coarser than the one induced by $\{\mathcal{L}_{f_v^{2\to 1}}(G_f) \mid v \in V\}$.*

Proof. Let f_v^1 be the coloring computed by 1-**stab** for G_{f_v}. Recall that f^1 is the coarsest 1-stable coloring of G_{f_v} and that the colorings $f_v^{2\to 1}$ are 1-stable on G_{f_v} for all $v \in V$. Since f_v^1 is the coarsest 1-stable partition of G_{f_v}, and $f_v^{2\to 1}$ is 1-stable and therefore at least as fine as f_v^1, the partition induced by $\{\mathcal{L}_{f_v^1}(G_{f_v}) \mid v \in V\}$ coarser than the one induced by $\{\mathcal{L}_{f^{2\to 1}}(v) \mid v \in V\}$.

□

Theorem 4.8 *The vertex partition computed by* **pointed***(*1**-stab***,1)** *is coarser than the one induced by* 2**-stab**.

Proof. This is evident due to the previous lemma. □

This result has been known before (see [68]) but no simple strictly graph theoretical proof can be found in the literature.

4.8 Coarsest Non-Trivial k-stable Partitions

An interesting question asked by M. E. Muzychuk [53] is whether every regular graph G has a coarsest non-trivial 1-stable partition and if yes, how to compute it. The question may be asked for higher dimensional cases as well.

The easiest case seems to be to decide whether a given graph has a 1-stable coloring with 2 colors only. Even for this case we were neither able to give a polynomial time algorithm nor to prove $\mathcal{NP}$-completeness.

4.9 Cayley Graphs

L. Babel [5] showed that coherent colorings of Cayley graphs can be computed considerably faster than for general graphs.

Let $\mathcal{G}$ be a group, $\mathcal{H} \subseteq \mathcal{G}$ with identity $1 \notin \mathcal{H}$. The *Cayley graph* $G(\mathcal{G}, \mathcal{H})$ is defined to be the graph with vertex set $\mathcal{G}$ and edge set $E = \{(g, h) : g^{-1}h \in \mathcal{H}\}$.

Theorem 4.9 *[5] The coarsest 2-stabel coloring of a Cayley graph G can be computed in time $O(n^2 log\, n)$.*

Chapter 5

Applications

5.1 Robinson Graphs

In this section, we briefly describe the notion of landscapes which was coined in recent publications of evolutionary biology [67, 69]. Afterwards, we investigate the structure of special graphs, namely Robinson graphs, which are useful in this context.

5.1.1 Landscapes

Graphs the vertices of which have an interior structure are often called *configuration graphs*. A landscape is a pair $(\mathcal{G}, w)$ of a configuration graph $\mathcal{G}$ and a function $w : \mathcal{V} \longrightarrow \mathbb{R}$ defined on the vertex set $\mathcal{V}$ of $\mathcal{G}$. Due to applications in biology, w is called *fitness function*. Landscapes are useful mathematical models for studying functions on a discrete set $\mathcal{V}$, the elements of which are structured objects. The configuration graph models a neighborhood relation on V, which defines how one is able to move within $\mathcal{V}$. Landscapes have a wide spectrum of applications.

Consider for example the Traveling Salesman Problem on a complete graph $\mathcal{G}$. In this case, $\mathcal{V}$ is the set of all tours in $\mathcal{G}$ and $w(T)$, $T \in \mathcal{V}$, is simply the length of T. The neighborhood of a tour T could be defined as the set of all tours which emerge from T by exchanging two adjacent cities. Of course, also more sophisticated neighborhoods can be defined and various local search procedures can be modeled by a configuration graph.

Landscapes can be described by their autocorrelation functions which are defined in terms of random walks on $\mathcal{G}$ and can be investigated by either using the eigenvalues and eigenspaces of $\mathcal{G}$ or via equitable partitions derived from its coherent algebra. See [68] for details.

In Section 2.2.1, we described how the eigenvalues of a graph G can be expressed in terms of the eigenvalues of graphs arising from pointed equitable partitions of G. As seen before in Section 4.7 equitable partitions f_v can be derived

from the coherent algebra of a graph very easily. Hence, the knowledge of the coherent algebra is of interest in this context as well.

5.1.2 Genetic Trees

The configuration graphs examined here were introduced by D. F. Robinson[59]. These graphs are of interest because the reconstruction of phylogenies can be modeled as an optimization problem on such graphs. A suitable way to state the problem is via defining a landscape, see [11] for details.

Although we are in general not able to determine the coherent algebra induced by a graph of so huge a size, we are at least able to determine the cell partition (see 2.2.1) of the coherent algebra of Robinson graphs, which is an important equitable partition.

Let T be a tree. A *leaf* of T is a vertex v of degree 1. All other vertices are called *inner vertices*. Edges joining inner vertices are called *inner edges*. A tree is called *genetic* if all its inner vertices have degree 3 and its leaves are colored with the colors $1, 2, \ldots, n$, whereas all inner vertices are uncolored.

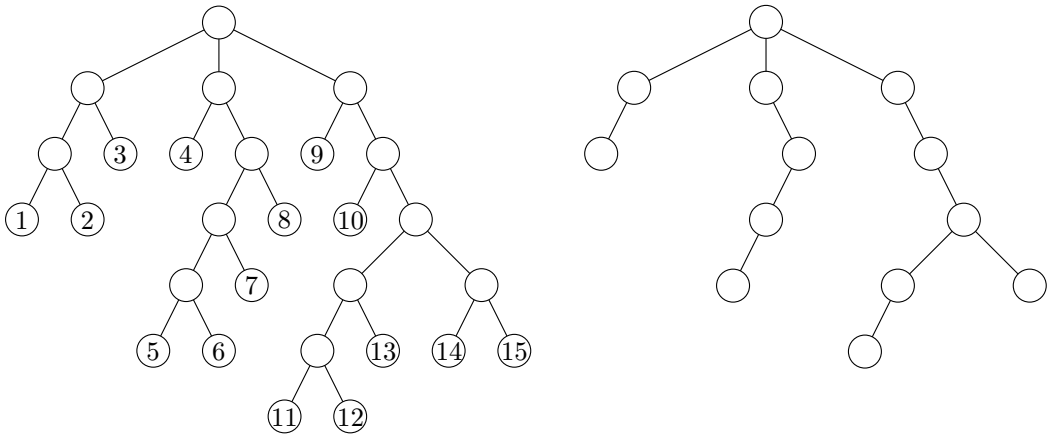


Figure 5.1: A genetic tree T (a) and its inner tree T° (b)

The set of genetic trees with n leaves will be denoted by $\mathcal{T}_n$. A member of $\mathcal{T}_n$ has $2n-3$ edges and $n-2$ inner vertices [59]. The *inner tree* T° of a genetic tree T is the subtree of T induced by the inner vertices of T. Two genetic trees are considered *equal* if and only if they are isomorphic as leaf colored trees. Observe that equal trees have isomorphic inner trees.

In **Figure 5.2** the inner trees in $\mathcal{T}_n$ for n up to 8 are depicted.

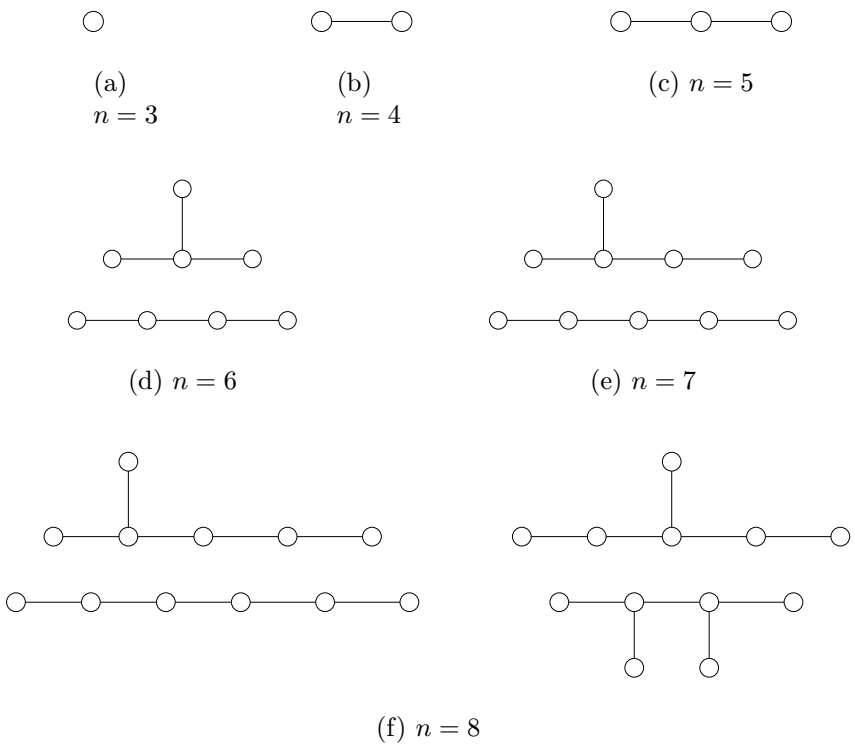


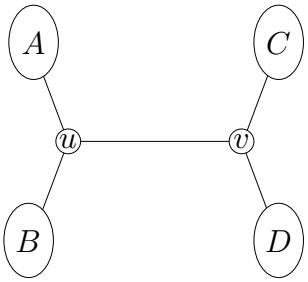
Figure 5.2: Inner trees of genetic trees

An inner vertex is called *s-vertex* if its degree with respect to the inner tree T° is s. The number of s-vertices will be denoted by n_s, $s = 1, 2, 3$. We call an edge of the inner tree an $(s : t)$-edge if the end vertices of the edge are an s- and a t-vertex.

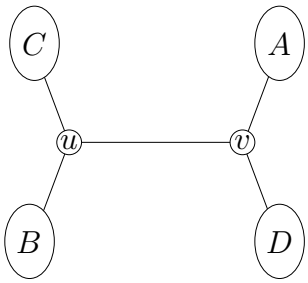
In many considerations, the coloring of the leaves is of no matter. In such cases, it is usually not mentioned.

With every inner edge $[u, v]$ of a genetic tree T, we associate four subtrees A, B, C, D as indicated in **Figure 5.3(a)**. The subtrees A, B, C, D are the four connected components which are obtained when deleting the edge $[u, v]$ and the vertices u and v from T. Note that each of these subgraphs may consist of a single vertex only.

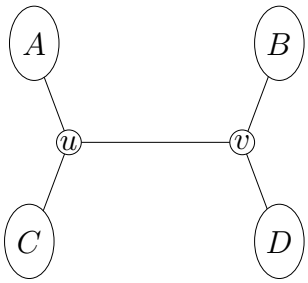
Definition 5.1 *The operations indicated in Figure 5.3(b) and (c) are called p(arallel)-crossover and d(iagonal)-crossover of T (on the inner edge $[u, v]$).*



(a) inner edge with subtrees



(b) p-crossover



(c) d-crossover

Figure 5.3: Crossovers

The type of the crossover is not a graph theoretical property. It depends on the drawing or the current ordering of the edges adjacent to some vertex. It is only introduced to simplify the argumentation at certain points.

A crossover on $[u, v]$ is called $(s : t)$-*crossover* if $[u, v]$ is an $(s : t)$-edge. We say that two trees are of the same *type* if and only if their inner trees are isomorphic.

Definition 5.2 *The configuration graph (Robinson graph)* Γ_n *has vertex set* $\mathcal{T}_n$ *and two trees* T, T' *are adjacent in* Γ_n *if and only if there exists an inner edge* $e \in T$ *such that* T' *results from* T *by a crossover on* e*. The vertices of* Γ_n *are called the trees of* Γ_n*.*

Observe that Γ_3 consists of 1 vertex only and that Γ_4 is the complete graph on 3 vertices.

Remark 5.3 The configuration graph Γ_n has $\prod_{i=0}^{n-3}(2i+1)$ vertices. It is $(2n-6)$-regular and the number of trees with distance two from a given tree equals $2n^2 - 10n + 4n_1$, i.e., it depends only on n and the number of 1-vertices of the inner tree. Furthermore, the numbers n_2 and n_3 depend only on n and n_1 in a simple way, namely $n_2 = n_1 - 2$ and $n_3 = n - 2n_1$. Proofs for these observations can be found in [59].

5.1.3 The Cell Partition of Γ_n

Remember that by definition the coherent configuration $\mathcal{A}_{\Gamma_n}$ generated by the graph Γ_n is the partition of $\mathcal{T}_n \times \mathcal{T}_n$ associated with the coarsest 2-stable coloring Γ_n starting with f_{int}. This partition contains a partition of $\mathcal{T}_n$, the cell partition $\mathcal{C}_{\Gamma_n}$ of Γ_n. In the following, we are going to determine the cell partition $\mathcal{C}_{\Gamma_n}$ by analyzing the structure of a 2-stable coloring of Γ_n.

Let $\{\tilde{T}_k \mid 1 \leq k \leq K\}$ denote the set of pairwise non-isomorphic inner trees with $n-2$ vertices. Let $[\tilde{T}_k]$ denote the set of genetic trees with inner tree isomorphic to $\tilde{T}_k$. Elements of $[\tilde{T}_k]$ differ only by the coloring of their leaves. Obviously, $\mathcal{C}_n := \{[\tilde{T}_k] \mid 1 \leq k \leq K\}$ is a partition of $\mathcal{T}_n$ as well.

Definition 5.4 *Let π be an arbitrary permutation of $\{1, 2, \ldots, n\}$. For a genetic tree $T \in \mathcal{T}_n$ define the tree $\pi(T)$ by replacing the colors $1, 2, \ldots, n$ on the leaves of T by the colors $\pi(1), \pi(2), \ldots, \pi(n)$, respectively.*

Let

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 4 & 3 & 9 & 2 & 15 & 8 & 11 & 14 & 6 & 13 & 5 & 7 & 12 & 10 & 1 \end{pmatrix}$$

and consider T as defined in **Figure 5.1**. The genetic tree $\pi(T)$ is depicted in **Figure 5.4**.

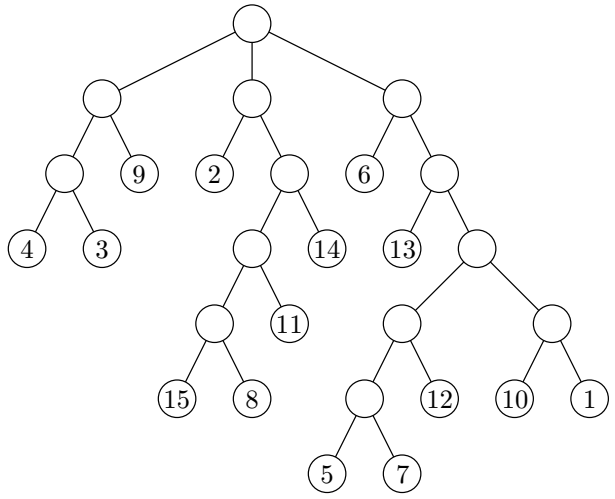


Figure 5.4: $\pi(T)$

Lemma 5.5 *The partition $\mathcal{C}_n$ is at least as fine as the cell partition $\mathcal{C}_{\Gamma_n}$.*

Proof. Let π be an arbitrary permutation of $\{1, 2, \ldots, n\}$. This permutation preserves the partition $\mathcal{C}_n$, i.e., $\pi([\tilde{T}_k]) = [\tilde{T}_k]$, $1 \leq k \leq K$. Furthermore, if T' is a neighbor of T due to a crossover on an inner edge $[u, v]$, then $\pi(T')$ is a neighbor of $\pi(T)$ due to the same crossover. Hence, π induces an automorphism of Γ_n.

Since for any two trees $T, T' \in [\tilde{T}_k]$, there is a π such that $\pi(T) = T'$, we obtain that $\mathcal{C}_n$ is at least as fine as the automorphism partition of Γ_n. Now, the claim follows by the fact that the cell partition of a graph is always coarser than its automorphism partition. □

Lemma 5.5 shows that all genetic trees in Γ_n having isomorphic inner trees belong to the same cell of $\mathcal{C}_{\Gamma_n}$. In the following, we want to prove that the other direction is true as well. In other words: trees contained in the same cell have isomorphic inner trees, and thus, $\mathcal{C}_n$, the automorphism partition of Γ_n, and the cell partition of $\mathcal{A}_{\Gamma_n}$ coincide. To prove this, we show that two trees having non-isomorphic inner trees lie in different cells of $\mathcal{C}_{\Gamma_n}$.

Let U be a cellular set (a union of cells) of $\mathcal{A}_{\Gamma_n}$. In the discussion which follows, we use the obvious fact that two vertices having a different number of neighbors in U, cannot belong to the same cell of $\mathcal{C}_{\Gamma_n}$.

We will start by showing that the sets defined in Definition 5.6 are cellular sets of $\mathcal{A}_{\Gamma_n}$.

Definition 5.6 *Let $\mathcal{T}_n^{n_k}(i)$, $1 \leq k \leq 3$, be the subset of $\mathcal{T}_n$ in which each element has i k-vertices and $\mathcal{T}_n^{dm}(i)$ the subset of $\mathcal{T}_n$ in which each element has diameter i.*

As mentioned above in Remark 5.3, the number of trees with distance 2 from a tree $T \in \mathcal{T}_n$ depends only on n and the number of 1-vertices. Due to Lemma 4.4, the following lemma holds.

Lemma 5.7 *The trees in a given cell of $\mathcal{C}_{\Gamma_n}$ have the same number of 1-vertices (n_1 is constant on each cell).*

Immediately, we get:

Lemma 5.8 *The trees in a given cell of $\mathcal{C}_{\Gamma_n}$ have the same number of 2-vertices and the same number of 3-vertices.*

Proof. Since $n_2 = n - 2n_1$ and $n_3 = n_1 - 2$, the claim holds. □

Using the Lemmas 5.7 and 5.8, we conclude:

Lemma 5.9 *Each $\mathcal{T}_n^{n_k}(i)$, $1 \leq k \leq 3$, defines a cellular set.*

We now examine crossovers and their potential to change the diameter, i.e., we examine how the diameter of the resulting tree differs from the diameter of the tree we start with.

Consider the tree T of **Figure 5.3** again. Define l_A and l_B to be the length of a longest path from u to a leaf of A and B, respectively, and l_C and l_D to be the length of a longest path from v to a leaf of C and D, respectively.

Lemma 5.10 *The diameters of a tree T and a tree T' which is obtained by a crossover on some inner edge of T can differ by at most one.*

Proof. Assume without loss of generality that $l_A \geq l_C$, $l_A \geq l_B$, and $l_C \geq l_D$. This situation can always be met by properly renaming the different parts of T.

The diameter of T is

$$\max\{l_A + l_B, l_A + 1 + l_C\}.$$

Consider now the trees T_p and T_d which are the result of a p- and d-crossover, respectively, of T on $[u, v]$. The diameters have the following values:

$$\begin{aligned}\operatorname{diam}(T_p) &= \max\{l_A + 1 + l_B, l_A + 1 + l_C\} \text{ and}\\ \operatorname{diam}(T_d) &= \max\{l_A + 1 + l_B, l_A + 1 + l_D, l_A + l_C\}.\end{aligned}$$

The p-crossover leaves the diameter untouched or enlarges it by at most one. Since $l_A + l_D \leq l_A + l_C$, $\operatorname{diam}(T_d)$ is at most $\operatorname{diam}(T) + 1$ and at least $l_A + l_C$ which is as least as large as $\operatorname{diam}(T) - 1$. □

An edge is *incident* with a path P if exactly one of the end vertices of the edge lies on the path. A path $P = (v_1, v_2, \ldots v_k)$ in a tree T, consisting of inner vertices only, is a *longest inner path* if and only if $k = \operatorname{diam}(T) - 1$. As an immediate consequence of Lemma 5.10, we obtain the following lemma.

Lemma 5.11 *A tree with a larger diameter is obtained if and only if a crossover is performed on an edge incident with a longest inner path.*

Proof. Recall the situation in the proof above. Consider the case where $\operatorname{diam}(T_p) = \operatorname{diam}(T) + 1$. By simply analyzing the formulas for $\operatorname{diam}(T_p)$ and $\operatorname{diam}(T)$, we see that this happens if and only if there is a longest inner path in T starting in A and ending in B. $[u, v]$ is incident to this path.

Now, assume that $\operatorname{diam}(T_d) = \operatorname{diam}(T) + 1$. This is true if and only if there is a longest inner path in T starting in A and ending in B. Again, $[u, v]$ is incident to this path.

□

Lemma 5.12 *The only way to obtain a tree with a smaller diameter by a crossover is to perform the crossover on an edge which is part of all longest inner paths.*

Proof. Revisit the proof of Lemma 5.10 again. The only possibility for reducing the diameter by a crossover on $[u, v]$ is that all longest inner paths in T go from leaves in A to leaves in C. □

We are now able to prove that all trees with equal diameter define a cellular set of $\mathcal{A}_{\Gamma_n}$. First, we will have a closer look at the trees of Γ_n having largest diameter. The inner trees with the largest diameter, namely $n-3$, are those isomorphic to the path on $n-2$ vertices. Note that the diameter of the inner tree T° of a tree T is exactly $\mathrm{diam}(T)-2$.

Lemma 5.13 *$\mathcal{T}_n^{n_1}(2) = \mathcal{T}_n^{dm}(n-1)$ is a cell of $\mathcal{C}_{\Gamma_n}$.*

Proof. $\mathcal{T}_n^{n_1}(2)$ is the set of trees whose inner trees are isomorphic to the path on $n-2$ vertices. Due to Lemma 5.5 and Lemma 5.7, $\mathcal{T}_n^{n_1}(2)$ is a cell of $\mathcal{C}_{\Gamma_n}$. □

Lemma 5.14 *All trees in a cell of $\mathcal{C}_{\Gamma_n}$ have equal diameter. Thus, $\mathcal{T}_n^{dm}(i)$ is a cellular set for all i.*

Proof. The proof is by downward induction on the diameter of the trees.

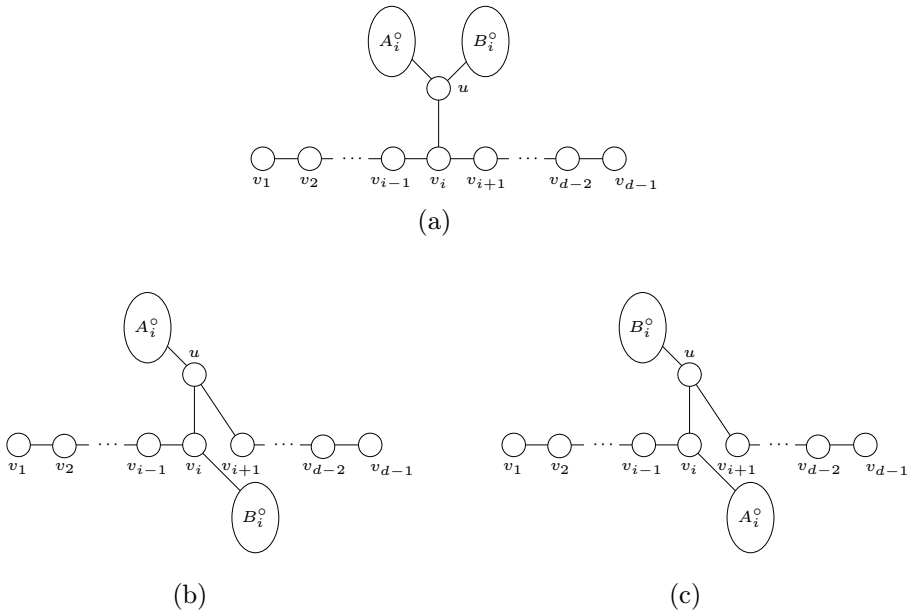


Figure 5.5: T° and the inner trees of the neighbors of T which result from crossovers on $[v_i, u]$

We have shown already that $\mathcal{T}_n^{n_1}(2)$ is a cell of $\mathcal{C}_{\Gamma_n}$. Assume that two trees with different diameters greater than d lie in different cells of $\mathcal{C}_{\Gamma_n}$.

Let T be a tree with diameter $d < n-1$. We will show that T has neighbors with diameter $d+1$. Observe that due to Lemma 5.10, the diameter can increase by at most 1 after executing one crossover and thus, trees with diameter d are the only candidates for having neighbors with diameter $d+1$.

Since $T \notin \mathcal{T}_n^{n_1}(2)$, each longest inner path contains at least one 3-vertex. Let $(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_{d-1})$ be such a longest inner path, v_i a 3-vertex, and denote the third neighbor of v_i by u (see **Figure 5.5(a)**). Applying the two possible crossovers on the edge $[v_i, u]$ results in two trees with diameter $d+1$ (see **Figure 5.5(b),(c)**), realized by a new longest path $P' = (v_1, v_2, \ldots v_i, u, v_{i+1}, \ldots, v_{d-1})$.

Hence, T has neighbors with diameter $d+1$. This completes the proof. □

Consider some longest inner path $P = (v_1, v_2, \ldots v_{d-1})$ in an inner tree T°. Assume that the 3-vertices on P are $\{v_{t_1}, v_{t_2}, \ldots, v_{t_k}\}$ with $\mathrm{dist}(v_1, v_{t_i}) < \mathrm{dist}(v_1, v_{t_j})$, $\forall i < j$, holds. Let u_i, $i \in \{1, 2, \ldots, k\}$, be the vertex not on P which is adjacent to v_{t_i}. If we perform both crossovers on $[v_{t_i}, u_i]$, a d-crossover and a p-crossover, we obtain two different trees. Observe that although the inner trees might be isomorphic, the resulting trees are different since they differ by the coloring of their leaves.

We will now, for each tree, identify "largest" neighbors among all neighbors with greater diameter. For this aim we are going to introduce an appropriate code for genetic trees. This task requires some preliminaries.

First, define the *code* $c_v(T)$ of a tree and an inner vertex v of this tree as the pair consisting of the length of a longest inner path from v to a leaf of T and of some complete invariant of T, for example the norm-code of T (see Section 5.2.1). We assume that codes can be compared lexicographically.

Next, define a function $c_T(P)$ on the set of inner paths P of a tree T. Let $P = (v_1, v_2, \ldots v_l)$ be an inner path between two 1-vertices v_1 and v_l. Assume that the vertices $\{v_{t_1}, v_{t_2}, \ldots, v_{t_k}\}$ are the 3-vertices on P. The subtree attached to v_{t_j} is denoted by Υ_j, and we assume $\mathrm{dist}(v_1, v_{t_j}) < \mathrm{dist}(v_1, v_{t_{j'}})$ if $j < j'$. Furthermore, each Υ_j consists of a node u_j adjacent to v_{t_j}, and the two subtrees A_j and B_j adjacent to u_j. The vertices in A_j and B_j adjacent to u_j are denoted by a_j and b_j, respectively. The situation is depicted in **Figure 5.6**.

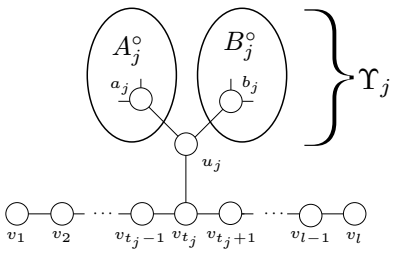


Figure 5.6: The path P in T° and the subtree Υ_j.

Let w.l.o.g. $c_{a_j}(A_j) \geq c_{b_j}(B_j)$. Define

$$c_T(P) := (l, ((\mathrm{dist}(v_{t_j}, v_k), c_{a_j}(A_j), c_{b_j}(B_j)) \mid j \in \{k, k-1, \ldots, 1\})).$$

Now we are able to introduce a suitable code for our trees. Define

$$c(T) = \max_{P \in \mathcal{P}(T)} \{c_P(T)\},$$

where

$$\mathcal{P}(T) := \{P \mid P \text{ is an inner path in } T\}$$

and where by "max" we mean the lexicographically largest value. We say that P is *responsible* for the code of T if $c(T) = c_T(P)$. Observe that if P is responsible for the code then it is a longest inner path in T. Obviously, given $c(T)$, we are able to reconstruct T in a unique way. A tree T is *larger* than another tree T' if $c(T)$ is lexicographically larger than $c(T')$.

Now we examine the situation with respect to the number of 3-vertices in more detail.

Lemma 5.15 *A* $(3:3)$*-crossover and a* $(3:2)$*-crossover leave the number of* 3*-vertices unchanged whereas a* $(3:1)$*-crossover reduces the number of* 3*-vertices by one.*

Proof. This is easy to verify. In **Figure 5.7(a)**, $[u, v]$ is a $(3:1)$-edge and the results of the two possible crossovers are shown in **Figure 5.7(b)** and **(c)**.

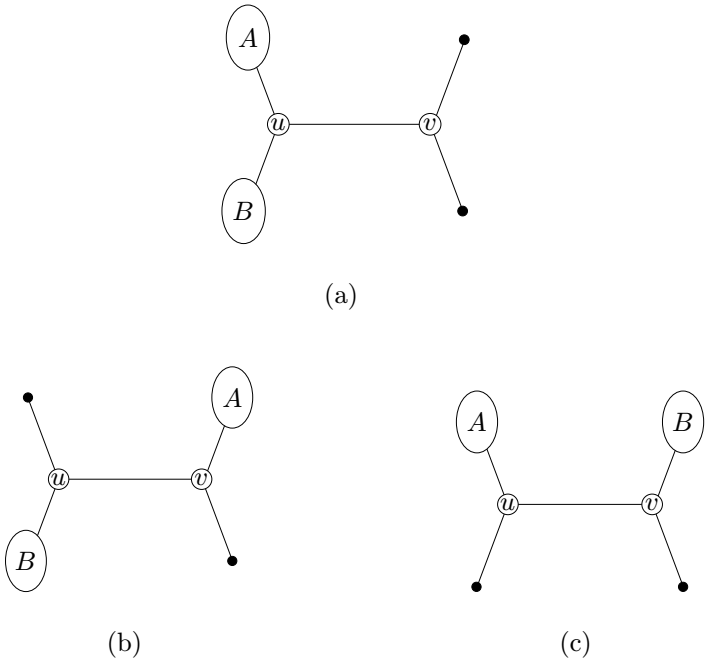


Figure 5.7: A tree and the two possible crossovers on a $(3:1)$-edge $[u, v]$

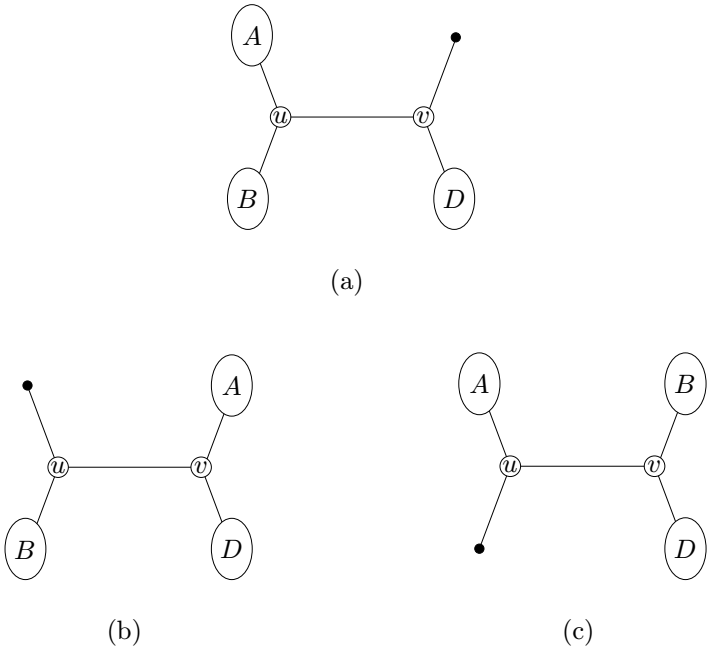


Figure 5.8: A tree and the two possible crossovers on a $(3:2)$-edge $[u,v]$

The edge $[u,v]$ becomes a $(2:2)$-edge. Note that the resulting inner trees are isomorphic.

In **Figure 5.8(a)**, $[u,v]$ is a $(3:2)$-edge and the results of the two possible crossovers are shown in **Figure 5.8(b)** and **(c)**. The edge $[u,v]$ remains a $(3:2)$-edge.

Obviously, a $(3:3)$-edge remains a $(3:3)$-edge. □

A neighbor of a tree is called *longer neighbor* if it has a larger diameter. A tree is called an *s-path* if its inner tree is a caterpillar with s legs, i.e., is composed of a longest inner path Q with s inner edges incident to it. Q is not necessarily unique. However, we assume that given an s-path, one of the possible longest inner paths is selected as Q.

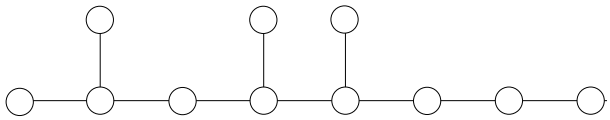


Figure 5.9: The inner tree of a 3-path with two choices for Q

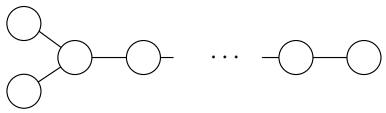
We first consider the set of all 1-paths and the set of all 2-paths, respectively, and show that they form cellular sets of $\mathcal{A}_{\Gamma_n}$. Afterwards, we turn to more general trees.

Let $Q = (v_1, v_2, \ldots, v_{d-1})$ be the selected longest inner path of an s-path, v_i the first and v_j the last 3-vertex on Q. Thus Q is the concatenation of three subpath Q_1, Q_2, Q_3, where Q_1 connects the 1-vertex v_1 to the first 3-vertex v_i, Q_3 connects the last 3-vertex v_j to the 1-vertex v_{d-1}. All other vertices of Q_1 and Q_3, if any, are 2-vertices. The subpaths Q_1 and Q_3 are called the *tails* of Q of length $i-1$ and $d-1-j$, respectively.

Lemma 5.16 *The 1-paths build a cellular set and are distinguished in $\mathcal{A}_{\Gamma_n}$ if they have non-isomorphic inner trees.*

Proof. The 1-paths build a cellular set since they are the only trees with diameter $n-2$ and one 3-vertex and the intersection $\mathcal{T}_n^{n_3}(1) \cap \mathcal{T}_n^{dm}(n-2)$ of two cellular sets is obviously a cellular set.

The inner tree of a 1-path with only one tail having length greater than one is isomorphic to the graph depicted below.



Trees having this inner tree are distinguished from the other 1-paths since they are the only ones having four neighbors in $\mathcal{T}^{dm}(n-1)$. All other 1-trees have only two such neighbors.

Assume now that the trees in question have two tails of lengths l_1 and l_2, and that w.l.o.g. $l_1 \leq l_2$ and $l_1 \leq \frac{d-1}{2}$ holds.

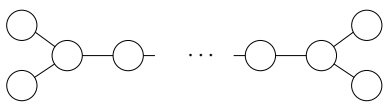
The proof is by induction on l_1. The proof for $l_1 = 1$ just has been given. Assume that the 1-paths with l_1 less than l are distinguished if they have non-isomorphic inner trees.

Consider now trees with $l_1 = l$. They are the only ones with $l_1 \geq l$ which have neighbors having a shortest tail of length $l-1$. This completes the proof. □

Lemma 5.17 *The 2-paths build a cellular set and will be distinguished in $\mathcal{A}_{\Gamma_n}$ if they have non-isomorphic inner trees.*

Proof. The 2-paths build a cellular set since they are the only trees with diameter $n-3$ and two 3-vertices, i.e., the only trees in $\mathcal{T}_n^{n_3}(2) \cap \mathcal{T}_n^{dm}(n-3)$. We define l_1 and l_2 as before.

The only 2-paths which have eight longer neighbors, which obviously are 1-paths, are trees the inner tree of which is isomorphic to the one depicted below ($l_1 = l_2 = 1$).



The length l_1 of the shortest tail of a 2-path T is determined by the code $c(T')$ of its largest neighbor T', which is a 1-path. By Lemma 5.16, 2-paths with different values of l_1 belong to different cells.

The remaining part of the proof is by induction on $l_1 + l_2$. The case when $l_1 + l_2 = 2$ has been considered already. The 2-paths with $l_1 + l_2 \geq l$ having neighbors where the sum of the tails is shorter than l are graphs with $l_1 + l_2 = l$.

□

So far, we have proven that trees the inner trees of which are caterpillars with at most two legs, belong to the same cell of $\mathcal{C}_{\Gamma_n}$ if and only if their inner trees are isomorphic.

Now, we treat more general classes of trees. Let T have diameter d. Assume that there exist neighbors of T having greater diameter than T. Let T_l be a largest (with respect to the code) neighbor among those neighbors. Assume that the crossover on T to obtain T_l has been performed on $[v_i, u]$. Since the diameter of T_l is greater than the diameter of T, each longest path in T_l must contain the edge $[v_i, u]$. Let $P_l = (v_1, v_2, \ldots, v_{i-1}, u, v_i, v_{i+1}, \ldots, v_{d-1})$ be an inner path responsible for the code of T_l. Then $P = (v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_{d-1})$ is a longest inner path in T. Obviously, v_i is the rightmost 3-vertex of P. Otherwise, P_l could not be a largest neighbor.

Lemma 5.18 *If the number of* 3*-vertices of* T *and* T_l *(as defined above) is equal, then* T° *is determined by* T_l°*.*

Proof. If T and T_l have the same number of 3-vertices, then the edge $[v_i, u]$ on which the crossover is performed is either a $(3:3)$- or a $(3:2)$-edge.

If $[v_i, u]$ is a $(3:3)$-edge, then subtrees isomorphic to A or B are attached to the rightmost 3-vertices (see Figure 5.10), namely u and v_i, on all paths in P_l' responsible for $c(T_l)$.

If the crossover has been performed on a $(3:2)$-edge, i.e., if in **Figure 5.10** B is a single vertex, then a subtree isomorphic to A is attached to the rightmost 3-vertex on all longest paths responsible for the code $c(T_l)$.

In both cases, the edge on which the crossover from T to T_l has been performed, is determined, namely the rightmost $(3:3)$-edge or $(2:3)$-edge, respectively, on a path responsible for the code $c(T_l)$.

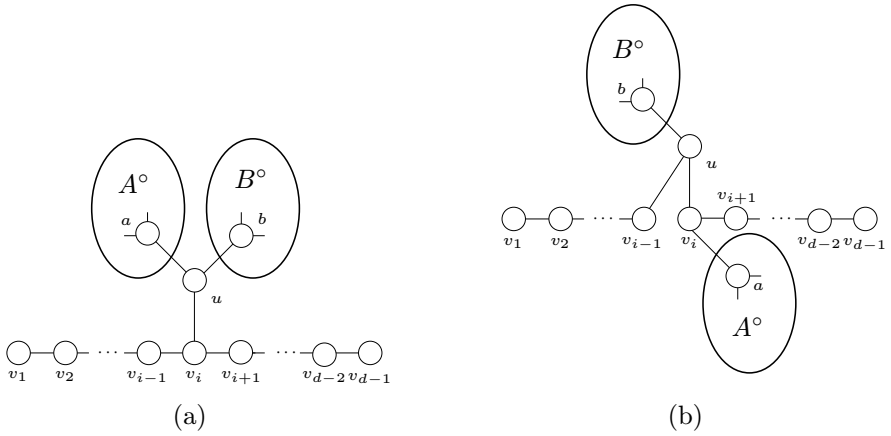


Figure 5.10:

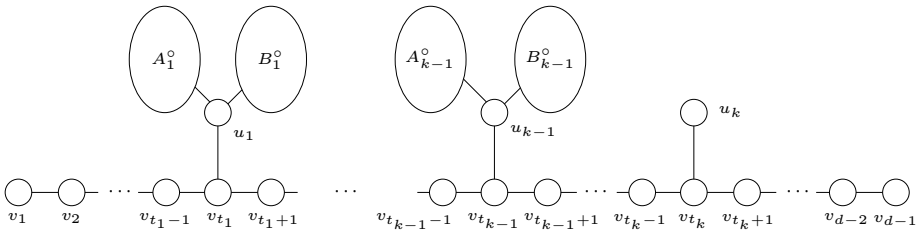
Therefore, we are able to reconstruct T° by only considering T_l°. □

Note that for finding a responsible path P, we need the whole tree rather than T° only, however, the coloring of the leaves in T is of no matter. Thus, what we need is the isomorphism class of T which is uniquely defined by T°.

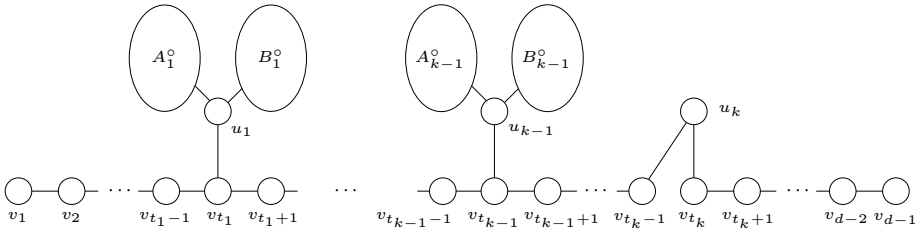
Now, let us consider the case where the largest neighbor T_l of T has fewer 3-vertices than T, i.e., the crossover leading from T to T_l is performed on a $(3:1)$-edge.

If this happens then clearly T° looks like in **Figure 5.11(a)** and T_l° like in **Figure 5.11(b)** where the path $P_l = (v_1, v_2, \ldots v_{t_k-1}, u_{k-1}, v_{t_k}, \ldots, v_{d-1})$ is responsible for the code of T_l. The 3-vertices of this path are $\{v_{t_1}, v_{t_2}, \ldots, v_{t_k-1}\}$.

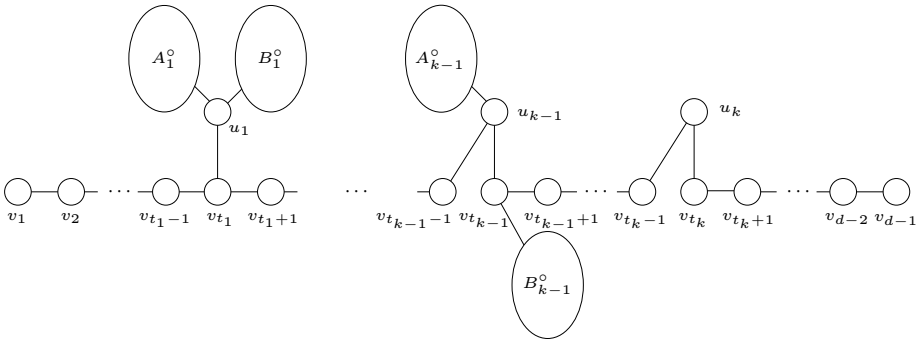
Consider now T_{xl}, the largest neighbor of T_l. It is clear that the crossover transforming T_l into T_{xl} has been made on the rightmost 3-vertex of P_l, namely $v_{t_{k-1}}$. This is because all paths in T_l which are responsible for the code contain the path from $v_{t_{k-1}}$ to v_{d-1}, since this is the only part of T_l where the length of a path has been increased with respect to T.



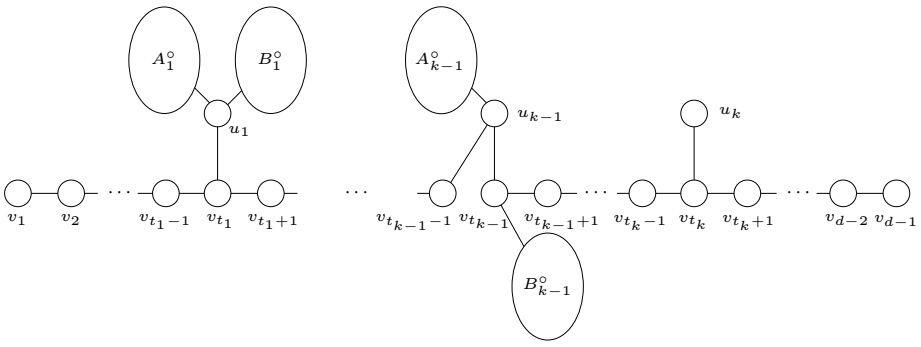
(a) T°



(b) $T^{2\circ}$



(c) $T^{4\circ}$



(d) $T^{5\circ}$

Figure 5.11: T and larger neighbors

For the same reason, this path is the right tail of all largest paths in T_l. It cannot be the left tail, since $\text{dist}(v_{d-1}, v_{t_{k-1}}) > \text{dist}(v_1, v_{t_1})$.

There is exactly one other path in Γ_n of length 2 from T to T_{xl} (by reversing the order of the two crossovers). Denote the tree on this path by T_x. Obviously, T_{xl} only exists if there are at least two 3-vertices on P. The situation is depicted in **Figure 5.11**.

Lemma 5.19 *If all largest neighbors of T have less 3-vertices than T, T° is determined by T_l° and T_{xl}°.*

Proof. As we have seen, there is the unique tree T_x. Since T_l has less 3-vertices than T, Υ_k has only one inner vertex, namely u_k (see **Figure 5.6**). Observe that T° is determined up to the position of v_{t_k} by T_l°.

Assume now that there are either at least two 3-vertices on P_l or a subtree Υ_j, $j < k$, has more than one inner vertex. Otherwise, the tree T would be a caterpillar with at most 2 legs, for which the result is already clear due to Lemma 5.16 and Lemma 5.17.

Let P_x be a path in T_x responsible for the code $c(T_x)$. Since the distance from the beginning of P_x to the first 3-vertex in P_x and of the rightmost 3-vertex on P_x are exactly as in P, a path responsible for the code $c(T)$, the position of v_{t_k} is determined by T_x°. □

Theorem 5.20 *Each $[\tilde{T}_k]$ defines a cell of $\mathcal{C}_{\Gamma_n}$.*

Proof. The proof is done by a similar induction as in Lemma 5.14. From Lemma 5.13 we know that $\mathcal{T}^{dm}(n-1)$ defines a cell. In fact, we even have proved that the sets $[\tilde{T}_k]$ are cells if $\tilde{T}_k$ is a 1-path or a 2-path. This result has already been used in the proof of Lemma 5.19.

Assume that the trees with diameter larger than d lying in one cell have isomorphic inner trees. Consider a tree T with diameter d. As we have seen before, T° can be determined by considering only the inner trees of some longer trees with distance one or two of T. To be more precise, let us consider the situation in Γ_n as depicted in **Figure 5.12**.

If a largest neighbor of T has the same number of 3-vertices as T (and thus all largest neighbors have this property), then T° is determined only by T_l° (see Lemma 5.18). Since by induction hypothesis the set $[T_l^\circ]$ is a cell, trees having a largest neighbor not in $[T_l^\circ]$ are distinguished from T. Hence, trees having a largest neighbor with the same number of 3-vertices lie in different cells of $\mathcal{C}_{\Gamma_n}$ if and only if their inner trees are isomorphic.

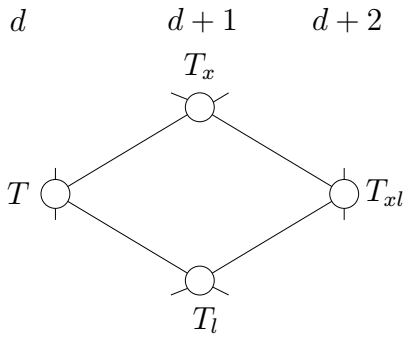


Figure 5.12: The crucial part of Γ_n

The case where all largest neighbors of T have less 3-vertices than T is more involved. As proved before, we need to consider T_x° together with T_l° to determine T° (see Lemma 5.19). In $\mathcal{A}_{\Gamma_n}$, the color of the edge (T, T_{xl}) represents the set of colored paths from T to T_{xl} (see Theorem 4.3). Hence, the color of (T, T_{xl}) depends on the colors of T_l and T_x as well. Thus, T° is determined by the color of the edge (T, T_{xl}) in $\mathcal{A}_{\Gamma_n}$.

Therefore, all trees with diameter d lie in the same cell only if they have isomorphic inner trees. □

5.1.4 Eigenvalues of the Laplacian of the Robinson Graph

Consider Γ_n. We were able to compute the pointed 1-stable partition for one representative vertex of each cell in Γ_n for n up to nine in reasonable time (see Table 5.1). **qStab** would be able to compute the pointed 1-stable partitions for larger instances of Γ_n but we are currently not able to create instances for $n > 9$.[1]

Using these results, we were able to compute the eigenvalues of the Robinson graphs for n up to 8 very quickly. For details on the methods we used for computing the eigenvalues and the computation times compared to a brute force approach, see the joint paper of O. Bastert, D. Rockmore, P. F. Stadler, and G. Tinhofer [11].

[1] See Section 6.1.2 for further computational results of **qStab** and Section 6.1.1 for some notes on the programs we used.

n	$\lvert\mathcal{T}_n\rvert$	Vertex pointed	Number of cells	Time in secs **qStab**	**nauty**
4	3	0	2	0.00	0.00
5	15	0	4	0.00	0.00
6	105	2	8	0.00	0.00
6	105	0	23	0.00	0.00
7	945	2	90	0.01	0.07
7	945	0	153	0.02	0.12
8	10395	277	158	0.23	17.61
8	10395	5979	880	0.24	99.29
8	10395	3663	888	0.26	100.66
8	10395	10371	1606	0.25	182.36
9	135135	15813	1534	5.31	-
9	135135	60357	3610	5.91	-
9	135135	33292	5901	5.51	-
9	135135	94791	10815	5.84	-
9	135135	135111	19698	6.31	-
9	135135	67089	21252	6.00	-

Table 5.1: Pointed 1-stable partitions of $\mathcal{T}_n$

Consider now the coloring $f^1_\circ$ computed by **pointed**(1-**stab**,1) on Γ_n. Obviously, the number $n_{f^1_\circ}$ of colors of $f^1_\circ$ is a lower bound on the dimension of the coherent algebra generated by Γ_n. We were able to show that $n_{f^1_\circ}$ equals this dimension for $n \in \{4,5,6,7\}$ using **qWeil** and **qStab**.

n	$\lvert\mathcal{T}_n\rvert$	Number of cells	Number of colors	Time in secs **qWeil**
4	3	1	2	0.00
5	15	1	4	0.00
6	105	2	31	0.78
7	945	2	243	1832.32

Table 5.2: Data of coherent algebras of Γ_n

In a first version of the proof of Theorem 5.20, the proof was only based on analyzing the neighborhood and the lengths and numbers of certain paths in Γ_n. Thus, the cells of $\mathcal{A}_{\Gamma_n}$ are determined by $f^1_\circ$ already. This fact and the computational results allow us to make the following conjecture.

Conjecture 5.21 $|f^1_\circ(\Gamma_n)| = \dim(\mathcal{A}_{\Gamma_n})$ *holds for* $n \geq 4$.

Concluding from the computational results for the eigenvalues, P. F. Stadler claims the following:

Conjecture 5.22 *The largest eigenvalue of the Laplacian of* Γ_n *equals* $3(n-3)$.

5.2 Graph Isomorphism, Automorphism and Canonical Labeling

In this section, we present frameworks of algorithms used for solving graph isomorphism, automorphism and canonical labeling problems. It will be seen that fast algorithms for computing strong invariants are crucial for the speed of these algorithms. In practice, most of the algorithms applicable to general graphs start with procedures for computing equitable partitions. This is because most graphs are easy problem instances in the sense that the total degree partition equals the automorphism partition. It can even be shown that for almost all graphs the canonical labeling problem and isomorphism problem can be solved in linear time[4, 3]. However, the set of graphs for which this true does not include the class of regular graphs or graphs with a small number of different degrees, which for example frequently appear in chemistry.

Nevertheless, algorithms for the above problems for all graphs are needed.

It can be shown that deciding graph isomorphism and finding a set of generators for the automorphism group are polynomially time equivalent problems and that a polynomial time algorithm for the canonical labeling problem would imply one for the isomorphism problem. Up to now, nobody was able to prove $\mathcal{NP}$-completeness of any one these three problems. Although no polynomial time algorithm has been found for graph isomorphism there are strong indications that is not $\mathcal{NP}$-complete [65].

5.2.1 Canonical Labeling

We now turn to the discussion of an algorithm for the computation of canonical labelings. A similar algorithm can be used for computing the automorphism partition or finding a set of generators for the automorphism group. See [48] for details. Consider a coloring f of a graph G. A color c is called *singular* if $|f^{-1}(c)| = 1$. f is *discrete* if all colors are singular.

If a coloring f is discrete, it defines a vertex-labeling of the graph, namely

$$l : V \to \{1, 2, \ldots, n\} \tag{5.1}$$

$$v \mapsto f(v). \tag{5.2}$$

In this way, for a colored graph G_f, a discrete canonical coloring determines a canonical labeling.

In our context, the *norm* of an adjacency matrix is defined as

$$||A(G)|| := \sum_{i,j \in \{1,2,\ldots,n\}} 2^{n(i-1)+(j-1)} (A(G))_{ij}.$$

Let f be a discrete vertex coloring. We define

$$(P_f)_{ij} = \begin{cases} 1 & \text{if } f(i) = j \\ 0 & \text{otherwise} \end{cases} \qquad \text{and} \tag{5.3}$$

$$A_f(G) = P_f A(G) P_f^t. \tag{5.4}$$

To obtain a complete invariant for graphs, consider the following. Every graph G can be represented by many adjacency matrices. Take an arbitrary discrete vertex-coloring of G, then $A_f(G)$ is the adjacency matrix of a graph isomorphic to G. It is well known that choosing the adjacency matrix with the smallest norm defines a complete invariant and a canonical representation of G. This value, namely

$$nc(G) := \min_{f \text{ is a discrete coloring}} ||A_f(G)||,$$

is called the *norm-code* of G.

In principle, one could compute all possible adjacency matrices of a graph and keep the one with the smallest norm to obtain the norm-code of G. A more intelligent approach uses a stabilization procedure to reduce the search space. We restrict our discussion to k-**stab**.

The approach discussed here uses pointed canonical colorings and starts with an arbitrary labeling which defines an initial discrete coloring f_{label}. Below, we present a framework for the computation of a discrete canonical vertex coloring for a given colored graph G_f. We refine f according to the chosen stabilization procedure. If the refined coloring is discrete, we compare it with the currently best coloring f_{label} and update f_{label} if necessary. If the coloring is not discrete, we assign to one of the vertices v_i in $PossVert(i)$ a new color and refine again. This is repeated until we obtain a discrete coloring which is again compared to f_{label}. Then we apply some backward steps, if necessary, choose other vertices in $PossVert(i)$ and proceed in the same manner.

Algorithm 22: Compute labeling

Data : $G_f = (V, E, f)$

Result: A discrete canonical coloring f_{label} of the graph

$i := 1$;
$f_1 \equiv k$-**stab**(f);
$f_{\text{label}} := f_1$;
do
  if f_i *is discrete* **then**
    if $||A_{f_{label}}(G)|| > ||A_{f_i}(G)||$ **then** $f_{\text{label}} := f_i$;
  else
    c=smallest non-singular color of f_i;
    $PossVert(i) := f_i^{-1}(c)$;
  end
  while $PossVert(i) = \emptyset$ **do**
    $i := i - 1$;
    if i=*0* **then** stop(f_{label});
  end
  choose $v_i \in PossVert(i)$;
  $PossVert(i) := PossVert(i) \setminus \{v_i\}$;
  $f_{i+1} \equiv k$-**stab**$((f_i)_{v_i})$;
  $i := i + 1$;
loop;

The speed of **Algorithm 22** depends on the chosen k-**stab** procedure and can be improved by techniques for recognizing colorings which do not have to be considered or refined anymore. See [48] for details.

5.2.2 Isomorphism Testing

We describe an algorithm for deciding whether two graphs G_f and $G'_{f'}$ are isomorphic or not.

A simple solution for solving graph isomorphism problems would be to enumerate the possible bijections from vertices of G_f to vertices of $G'_{f'}$, i.e., bijections $V \to V'$, and check whether one of them describes an isomorphism or not.

In a more efficient version, one would try to exclude some of the bijections by using stabilization procedures. The framework described in the following can be used with every canonical stabilization procedure.

The algorithm works as follows. Instead of checking a bijection from all vertices of the first graph onto the vertices of the second, it starts with no mapping at all and computes the structure constants with respect to the chosen k-**stab** procedure for both graphs G_f and $G'_{f'}$. If they are not equal, the algorithm

terminates. In this case, the two graphs cannot be isomorphic. Otherwise, the algorithm selects two vertices $v_i \in V$ and $v'_i \in V'$ having the same color and assigns v'_i to v_i. Afterwards, the current colorings of G_f and $G'_{f'}$ will be modified by pointing at v_i and v'_i, respectively. In these pointed colorings, v_i and v'_i have the same color. Now, another run of k-**stab** is performed.

We have to consider three cases: the algorithm observes that the assignments made so far cannot be extended to an isomorphism (by comparing the structure constants), the colorings computed so far are discrete (line 6), or another pair of vertices is selected and a the next assignment is made. In the first case, a backward step is performed, i.e., the last assignment is revoked and other assignments are tried out.

To make things work, we store lists of assignments which proved false and name them $ForbiddenPerms(i)$. In another list, called $PossiblePerms$, we store assignments which still have to be tested.

Algorithm 23: Test isomorphism

Data : G_f, $G'_{f'}$

Result: "yes" if $G \simeq G'$ and "no" otherwise

```
1:  ∀_{i∈{1,2,...,n}} : ForbiddenPerms := ∅;
2:  i := 1;
3:  f_i := f;
4:  do
5:      if k-stab(G_f) = k-stab(G'_{f'}) then
6:          if ∀c ∈ F : f_i^{-1}(c) = 1 then
7:              stop("yes");
            end
        else
8:          backwardStep;
        end
9:      Choose some color c with |f_i^{-1}(c)| > 1;
10:     PossiblePerms := ∅;
11:     while PossiblePerms = ∅ do
12:         PossiblePerms := {(u, u') | u ∈ f_i^{-1}(c) and u' ∈ f'_i^{-1}(c)} \
                ⋃_{j∈{1,2,...,i}} ForbiddenPerms(j);
13:         if PossiblePerms = ∅ then
14:             backwardStep;
            end
        end
15:     forwardStep;
    loop;
```

Procedure 24: backwardStep

```
1: if i=1 then
2:     stop("no");
   else
3:     ForbiddenPerms(i) := ∅;
4:     i − −;
5:     ForbiddenPerms(i) := ForbiddenPerms(i) ∪ (v_i, v'_i);
6:     f_i(v_i) := f'_i(v'_i) := c_i;
   end
```

Procedure 25: forwardStep

```
1: Select (v_i, v'_i) ∈ PossiblePerms;
2: c_i := f_i(v_i);
3: f_i(v_i) := f'_i(v'_i) := n_{f_i} + 1;
4: i + +;
```

The above framework for testing isomorphism of graphs is well known among experts. Obviously, the algorithm has at most exponential running time. Depending on the chosen k-**stab** procedure and the graph class, the algorithm may run in polynomial time. Examples will be given in Section 5.3.1. If no k-**stab** procedure is chosen, the above algorithm performs complete enumeration. In this case, once a complete assignment is made, one has to check explicitely if this assignment is a graph isomorphism.

5.2.3 Computations

As seen before, fast and strong k-**stab** procedures are of great importance. The certainly most prominent code for solving isomorphism, automorphism and labeling problems is the package **nauty** (**n**o **aut**omorphism, **y**es?) by B. D. McKay [49] which uses a 1-dimensional stabilization procedure in the above framework. Since **nauty** allows to implement user defined refinement procedures, it was relatively simple to test the algorithm for computing equitable partitions implemented in **nauty** against our implementation. The version of **nauty** using the implementation **qStab** of the algorithm presented in Section 3.3 for computing the equitable partitions is denoted by **qNauty**.

B. D. McKay maintains a list of interesting instances for the graph automorphism problem [50]. For our comparison, we have chosen a subset of instances out of this list, thereby selecting only instances which could be solved in less than one hour.

Since most part of the running time of **nauty** is needed for computing the equitable partitions ($\gg 99\%$), in the tables below only this part of the running time is reported.

Name	n	Orbits	**qNauty**	**nauty**	#EqPart
10cube	1024	1	1.79	4.29	45
11cube	2048	1	6.70	36.99	57
a52	52	4	4.11	0.13	1118
a60	60	8	25.73	0.34	4793
a72	72	36	155.56	2.70	21460
b171_1	171	5	103.88	23.56	12626
b52	52	8	12.97	0.21	3412
b60	60	18	51.51	0.81	9647
b72	72	36	155.64	2.59	21460
c72	72	36	155.95	3.12	21460
cub100_cifa	1000	400	22.75	3.15	1123
cub100_cifb	1000	400	22.87	3.26	1123
cub200_cifa	2000	800	198.98	47.71	3342
cub200_cifb	2000	800	189.03	43.05	3342
cub300	300	300	2.99	2.97	301
cub400	400	400	5.98	7.20	401
cub500	500	500	10.38	15.00	501
cub600	600	600	16.08	27.19	601
cub700	700	700	23.67	44.86	701
cub800	800	800	32.01	69.80	801
d72	72	36	155.20	3.15	21460
had112_1	112	6	384.87	33.04	34033
had112_2	112	6	189.92	17.49	17122
had96_1	96	2	298.22	18.15	38088
hanani	525	3	72.75	3.70	282
lk30	435	1	23.22	1.21	462
mols	271	17	127.94	49.20	9905
nice_7_3_3	1029	1	10.78	8.15	151
prim_5_11	275	1	4.02	0.32	550
rees	2312	28	8.28	17.90	23
sts117_1	117	117	62.23	2.28	4330
sts155_1	155	155	147.73	5.70	6666
tasty_125_5	625	1	148.94	3.85	5251
tasty_81_3	243	1	11.29	0.12	1594
trouble	174	11	10.27	0.06	258
trouble2	174	11	10.38	0.07	263
trouble3	174	94	7.12	0.03	171

Table 5.3: Results on interesting graphs for computing automorphisms

Table 5.3 shows that in most cases **nauty** is faster than **qNauty** however,

there are instances on which **qNauty** is faster. This seems to happen in particular on sparse and on large graphs. To obtain more evidence for this phenomenon, we tested the algorithms on some instances reported by D. Mathon [47] and on the instances considered by M. Fürer[31]. D. Mathon presents graphs which are difficult for the isomorphism testing and describes an operation called *doubling* to obtain larger graphs having similar properties.

Table 5.4 shows results for one of these graphs on 25 vertices and for graphs derived from it by doubling operations. The graphs obtained by this procedure are dense with about half of the possible edges present.

n	Degree	Orbits	#EqPart	**qNauty**	**nauty**
25	4,6	4	16	0.00	0.00
52	25	5	30	0.06	0.00
106	52	6	41	0.28	0.02
214	106	7	54	1.49	0.08
430	214	8	69	7.03	0.64
862	430	9	86	34.11	4.76
1726	862	10	105	160.52	36.78
3454	1726	11	126	752.07	314.90
6910	3454	12	149	3372.08	3406.66
13822	6910	13	174	19273.87	30230.10

Table 5.4: Results for graphs obtained by doubling from a25

The instances by M. Fürer are described in Section 4.5.1 are sparse (3-regular).

n	Orbits	**qNauty**	**nauty**	#EqPart
100	10	0.07	0.04	62
200	19	0.47	0.33	147
400	39	2.43	5.27	392
800	79	16.23	72.38	1182
1000	99	28.27	184.38	1727
1200	119	46.37	388.51	2372

Table 5.5: Results for the instances step

In a private discussion, B. D. McKay conjectured that the reason for the better running time of **qStab** on several instances can be explained by the different representations of the graph structure [50]. **nauty** stores edges as bit arrays while **qStab** uses adjacency lists. He concludes that **qStab** is more advantages on large and sparse graphs. B. D. McKay proved a worst-case time bound of

only $O(n^3)$, for his code for computing the equitable partitions. On the other hand, for our code we have proved the time bound of $O(m \log^2(n))$. Since this is a bound involving the number of edges rather than the number of vertices only, it may help to explain the fact that **qStab** is in fact faster for sparse graphs. Further, the better theoretical time bound, as it is often the case, pays off not for small but only for large instances.

5.3 Polynomial Time Solvable Cases

In this section, we give examples of graph classes for which the recognition problem or the isomorphism problem is solvable in polynomial time using stabilization procedures.

5.3.1 Compact and Weakly Compact Graphs

We start by defining the graph automorphism problem with the help of a system of equalities.

Let A be the adjacency matrix of some graph G and Π_n the set of all permutation matrices of degree n. A matrix $P \in \Pi_n$ represents an automorphism of G if and only if P commutes with A, i.e.,

$$PA = AP.$$

Using this, we can represent the automorphism group of G by

$$Aut(A) = \{P \in \Pi_n | PA = AP\}. \tag{5.5}$$

This representation can be used to formulate a polyhedral approach to the graph automorphism problem. Let Σ_n denote the set of all doubly stochastic matrices S of degree n. Each $S \in \Sigma_n$ can be considered as a point in the space $\mathbb{R}^{n \times n}$. It is well known that $\Sigma_n = conv(\Pi_n)$, where $conv(\Pi_n)$ denotes the convex hull of Π_n. Hence, Σ_n is a polytope.

Denote by $\mathcal{DS}(A)$ the set of doubly stochastic matrices commuting with A, i.e.,

$$\mathcal{DS}(A) := \{S \in \Sigma_n | SA = AS\}.$$

Note that $\mathcal{DS}(A)$ is equal to the set of solutions of ($\mathbf{e} = (1, 1, \ldots, 1)^T$):

$$\begin{aligned} SA - AS &= 0 \\ S\mathbf{e} &= \mathbf{e} \\ S^t\mathbf{e} &= \mathbf{e} \\ S &\geq 0 \end{aligned}$$

The automorphisms of G are exactly the integral solution of the above equality-system. Clearly, they are vertices (extreme points) of the polytope $\mathcal{DS}(A)$. In some favorable cases, all vertices of this polytope are integral and hence automorphisms of G. In such cases, we are able to find non-trivial automorphisms in polynomial time [70, 71]. Graphs for which $\mathcal{DS}(A)$ is integral are called *compact*, i.e, G is *compact* if and only if

$$\mathcal{DS}(A) = conv(Aut(A)).$$

The notion of compactness has been introduced by G. Tinhofer in [70], in order to create a tool for studying the complexity of a simpler variant of **Algorithm 23**. In that paper, compact graphs have been called *Birkhoff graphs*. The notation *compact* has been introduced by R. A. Brualdi in [13]. The importance of the notion of compactness is based on the fact that for two graphs G and G', if at least one of them is compact, **Algorithm 23** with 1**-stab** as k**-stab** does not make any backward steps and hence correctly solves the isomorphism problem for G and G' in polynomial time.

Instead of considering the adjacency matrix, we could also use the information of the coherent algebra $\mathcal{A}$ generated by A.

A permutation matrix P is called a *strong automorphism* of $\mathcal{A}$ if and only if P commutes with every matrix in $\mathcal{A}$, i.e.,

$$B \in \mathcal{A} \Longrightarrow PB = BP.$$

Let $Aut(\mathcal{A})$ be the group of strong automorphisms of $\mathcal{A}$. It is well known that $Aut(\mathcal{A}) = Aut(A)$.

Analogous to the definition of $\mathcal{DS}(A)$, let $\mathcal{DS}(\mathcal{A})$ be the polytope of doubly stochastic matrices which commute with every matrix in $\mathcal{A}$. In general, $\mathcal{DS}(\mathcal{A})$ is a subset of $\mathcal{DS}(A)$.

The algebra $\mathcal{A}$ is called *compact* if and only if

$$\mathcal{DS}(\mathcal{A}) = conv(Aut(\mathcal{A})).$$

G is called *weakly compact* if its algebra $\mathcal{A}$ is compact.

Compactness of a graph implies its weak compactness, the converse statement is not always true [22].

Note that if a graph is compact then 1**-stab** computes the automorphism partition and likewise, if it is weakly compact, then 2**-stab** computes the automorphism partition.

Weak compactness has been introduced by S. Evdokimov, M. Karpinski and I. N. Ponomarenko in [22]. For the purpose of efficiently testing graph isomorphism, weak compactness is not worse than compactness. It can be shown that isomorphism testing of two graphs can be done in polynomial time if at least one of them is weakly compact [71, 22]. Again, this is done by showing that **Algorithm 23** does not make any backward steps if at least one of two given graphs is compact and 2**-stab** is chosen as k**-stab**.

The up to now largest class of weakly compact graphs is the class of algebraic forests [26] which includes forests, cographs and tree-cographs, rooted-directed-path graphs, and interval graphs.

5.3.2 Higher Dimensions

Very recently, M. Grohe [35] has proven that for every graph G there is a $k = k(G)$ which is linear in the genus of G such that $\mathbf{il}(k)$ decides isomorphism of G and

any other candidate G'.

5.3.3 Recognition

In certain cases, the membership problem for graph classes can be tested in polynomial time using the coherent algebra approach. One example is given by the class of algebraic forests [26] which allow a membership test in $O(n^3 \log(n))$. The time needed for computing the coherent algebra is the dominating factor. If the coherent algebra is already known, the membership can be tested in $O(n^2 \log(n))$. Another graph class which allows polynomial time membership testing using coherent algebras are circulant graphs of prime order [55], and as it has been recently shown, so-called geometric circulants [54].

5.4 Stabilization Procedures in Chemistry

There are several applications in chemistry which are related to the graph automorphism and the canonical labeling problem.

To retrieve chemical structures in a database or to automatically assign names (reference codes) to them, it is necessary to obtain a canonical form of a molecule, that is to obtain a canonical labeling.

The coding of a molecule M with n atoms $\{a_1, a_2, \ldots, a_n\}$ as a colored graph $G = (\{v_1, v_2, \ldots, v_n\}, E, f)$ works as follows. The atoms correspond to colored vertices and the bonds to colored edges of G. Edges between two vertices v_i and v_j are present if and only if there is a chemical bond between the atoms a_i and a_j. Colors of vertices represent different types of atoms. Colors of edges represent different types of bonds. Two vertices v_i and v_j have the same color if the corresponding atoms a_i and a_j are of the same type and two edges (v_i, v_j) and $(v_{i'}, v_{j'})$ have the same color if they correspond to the same type of bond. The resulting colored graph is called a *molecular graph*.

Chemists are interested in classifying atoms according to their function within the molecule, roughly speaking, two atoms play identical roles if they can be mapped onto each other by a certain automorphism of the molecular graph. For this reason the automorphism partition of molecular graphs is of interest. Moreover, also bonds have to be classified in this way, therefore, the notion of 2-orbits of the automorphism group of the molecular graph becomes important. The 2-orbits of $V \times V$ with respect to a group acting on V are the orbits of this group considered as a permutation group of $V \times V$.

Stabilization algorithms are widely used in chemistry to solve the tasks described above. Below, we give some examples.

T. Laidboeur, D. Cabrol-Bass and O. Ivanciuc [45] incorporate not only the information of the molecular graph but also the lengths of bonds into their approach to the structure analysis to obtain quantitative geometrical symmetry information. They perform a geometrical distance partition of the vertex set, i.e, for each atom they count the numbers of atoms at certain distances in space. All this information is encoded by an initial coloring. Additionally, they perform a 1-dimensional stabilization to split the set of vertices further and compute a canonical labeling of the graph by enumeration. Their method is a slim version of **Algorithm 22**, as far as they apply a stabilization procedure in the beginning and then continue by complete enumeration. This labeling distinguishes isomers of chemical compounds even if the underlying molecular graphs are isomorphic.

J. Faulon in [27] presents an algorithm implementing a decomposition method for planar molecular graphs which is based on 1-dimensional stabilization. He obtains a new algorithm for canonically labeling planar graphs.

In addition, the spectra of graphs are widely used to determine a vertex partition, see for example [46]. However, as discussed in Section 4.6.1, this criterion is not sufficient to distinguish non-isomorphic graphs and further, **2-stab** yields

stronger results. In addition, since stabilization methods work with small integers only, no numerical problems can occur as it may happen with the eigenvalue approach.

5.4.1 The Algorithm of G. Rücker and Ch. Rücker

Here an algorithm is discussed in more detail which uses an approach similar to 2-dimensional stabilization and simultaneous pointing of colorings at several vertices.

It is one of the algorithms presented in a series of papers [61, 62, 60]. We start by stating the algorithm presented in [61] and add improvements in the following paragraphs.

As for graphs, we can define colors, structure values and structure lists for a matrix A. Define the colors $\mathcal{F}(A)$ as

$$\mathcal{F}(A) := \{c \in \mathbb{N} \mid \exists i, j : A_{i,j} = c\}$$

and define structure values

$$p_v^c(A) := |\{w \in V \mid A_{(v,w)} = c\}|,$$

$$\overline{p}_v^c(A) := |\{w \in V \mid A_{(v,w)} = c, (v,w) \in E\}|$$

and structure lists

$$L(v, A) := \{(c, p_v^c(A)) \mid c \in \mathcal{F}(A), p_v^c(A) \neq 0\} \text{ and}$$

$$\overline{L}(v, A) := \{(c, \overline{p}_v^c(A)) \mid c \in \mathcal{F}(A), \overline{p}_v^c(A) \neq 0\}.$$

Two vertices v and w are *equivalent* or *neighborhood equivalent* with respect to A if and only if $L(v, A) = L(w, A)$ or $\overline{L}(v, A) = \overline{L}(w, A)$, respectively. Two edges (v, w) and (v', w') are equivalent if and only if $\{f(v), f(w)\} = \{f(v'), f(w')\}$.

G. Rücker and Ch. Rücker start with a coloring in which all edges are equally colored.

Algorithm 26: Improved Degree Partition **rücker**(0)

Data : A colored graph $G_f = (V, E, f)$

Result: A Rücker coloring of the graph

1: $A := A(G_f)$;
2: steps:= 0;
3: **repeat**
4: steps++;
5: perform an out-degree partition on the vertices and recolor;
recolor the vertex set: $f(v) = f(w) :\Leftrightarrow L(v, A^{steps}) = L(w, A^{steps})$, $\overline{L}(v, A^{steps}) = \overline{L}(w, A^{steps})$, and $f(v) = f(w)$, $v, w \in V$;
6: recolor the edge set: $f(v, w) = f(v', w') :\Leftrightarrow A^{steps}_{v,w} = A^{steps}_{v',w'}$ and $f(v, w) = f(v', w')$, $v, w \in V$;
until n_f *did not change and at least* diam(G_f) *steps are performed*;
7: $f(v, w) = f(v', w') :\Leftrightarrow \{v, w\}$ and $\{v', w'\}$ are equivalent and $f(v, w) = f(v', w')$, $(v, w), (v', w') \in E$;

The running time of the implementation of **Algorithm 26** given by G. Rücker and Ch. Rücker can roughly be estimated as follows. The most important factor is the calculation of the powers of A and the comparison of the structure values. The computations of the powers of A take time $O(n^3)$ and the comparison between the structure lists is done by a simple algorithm taking time $O(n^4)$. This, of course, could be done using bucketsort in time $O(n^2)$. Since up to n^2 steps have to be repeated, the overall running time obtained is $O(n^6)$. This could be lowered to $O(n^{4.376})$ since matrix multiplication can be implemented to take time $O(n^{2.376})$ [17] only. As we have seen in Section 4.6, it is more useful to perform 2-dimensional stabilization than to consider only the powers of the adjacency matrix.

We immediately obtain.

Theorem 5.23 rücker*(0) is weaker than 2-dimensional stabilization.*

In an improved version, the algorithm of G. Rücker and Ch. Rücker chooses a pair of different vertices and introduces a new (possibly additional) edge between them (**rücker**(2)). Then **Algorithm 26** is applied to this altered graph and the number of colors is stored. This is done for every pair of vertices and the result is used to obtain an initial partition of the vertex set and of the edge set. Then **Algorithm 26** is applied to this pre-colored graph. Instead of a pair, choosing a triple of pairwise different vertices and introducing edges between them is also considered (**rücker**(3)). This approach is similar to the one introduced in Section 4.7.

For the description of **rücker**(2) and **rücker**(3) the definition of more general structure values is needed. For $k \in \mathbb{N}$, let

$$C = (C_{i_1,i_2,\ldots,i_k})_{i_1,i_2,\ldots,i_k \in \{1,2,\ldots,n\}}, \quad C_{i_1,i_2,\ldots,i_k} \in \mathbb{N}$$

and

$$p^c_{(u,w)}(C) := |\{v^{k-2} \in V^{k-2} \mid C_{(u,w,v^{k-2})} = c\}|$$

and define the structure lists in the usual way. Then the algorithm **rücker**(k), $k \geq 2$, reads as follows.

Algorithm 27: rücker(k), $k \geq 2$

Data : $G_f = (V, E, f)$
Result: A k-Rücker coloring of the graph
foreach *k-tuple v^k of pairwise different vertices* **do**
    $G'_{f'} = G_f$;
    add to $G'_{f'}$ edges between each v_i and v_j, $i, j \in \{1, 2, \ldots, k\}$, $i \neq j$;
    perform **rücker**(0) on $G'_{f'}$ and store the number of colors in $C_{v_1,v_2,\ldots,v_k}$;
end
Split the colors of vertices and the edges set of the original graph according to the structure lists regarding C;
Perform **rücker**(0) on the resulting colored graph;

Analogously, we define **rücker**(1) where we add loops (we point colors at vertices) to obtain the graph $G'_{f'}$.

The implementations of algorithm **rücker**(k), $k = 2, 3$, presented by G. Rücker and Ch. Rücker run in time $O(n^{k+5})$.

Corollary 5.24 **rücker***(1) is weaker than* **pointed***(2,1).*

Proof. This follows immediately from the theorem above. □

5.4.2 Computational Experiments

Computational experiments support our theorem that 2-dimensional stabilization, and therefore of course also **pointed**(2,1), is stronger than **rücker**(0). For this reason, **qWeil** is preferable and could be used in **rücker**(2) and **rücker**(3) as refinement procedure as well. **qWeil** and **pointed**(2,1) often yield finer refinements of the edge partition than **rücker**(2) and **rücker**(3). But no examples have been found on which they yield finer refinements of the vertex set. The vertex partition computed by **rücker**(2) and **rücker**(3), at the cost of much longer running time, is often much finer than the one computed by **qWeil** and **pointed**(2,1).

The computational results given in Tables 5.6 and 5.7 show impressively the power of pointing colorings at vertices and on sets of vertices. Again, we choose for testing purposes the test instances used by B. D. McKay. Their names are listed in the first column. n is the instance size, s_f denotes the number of different

vertex colors, r_f the total number of different colors and the entries in the columns for algorithms represent the CPU time in seconds.

Name	n	**qWeil**	s_f	r_f	**pointed**(2,1)	s_f	r_f	**rücker**(0)	s_f	r_f	**rücker**(2)	s_f	r_f
C3903	58	0.45	30	1684	24.53	30	1684	0.86	30	871	1186.06	30	871
C3905	58	0.28	10	323	37.07	10	323	0.60	10	181	1186.75	10	181
b25	25	0.01	2	9	1.06	5	43	0.03	2	7	6.89	5	29
b251	25	0	1	3	1.79	1	3	0.02	1	3	5.77	25	325
b291	29	0.01	1	3	3.23	1	3	0.03	1	3	12.64	29	435
b35	35	0.01	1	3	6.19	1	3	0.06	1	3	33.42	35	630
b50	50	0.04	2	9	22.69	2	9	0.22	2	7	355.47	50	1275
b52	25	0	1	3	1.79	1	3	0.02	1	3	5.77	25	325
b60	29	0.01	1	3	3.23	1	3	0.03	1	3	12.65	29	435
b72	35	0.01	1	3	6.19	1	3	0.06	1	3	33.38	35	630
cage_1	58	0.36	20	882	28.94	20	882	0.70	20	470	1183.37	20	470
cage_2	58	0.45	31	1690	24.68	31	1690	0.87	31	874	1184.49	31	874
cage_3	58	0.25	5	150	35.63	5	150	0.56	5	88	1190.35	5	88
cage_4	58	0.6	16	846	26.77	16	846	0.75	16	452	1182.31	16	452
cheryl	57	0.11	1	12	21.38	1	12	0.22	1	8	405.50	1	8
cuddly_2	16	0	1	7	0.15	1	7	0.01	1	7	0.62	1	7
cuddly_3	81	0.43	1	45	82.65	1	45	0.84	1	18	3236.96	1	23
cute_3_13	78	0.36	1	42	57.21	1	42	1.01	1	22	4672.84	1	22
cute_3_7	42	0.06	1	15	7.19	1	15	0.14	1	11	156.23	1	11
d35	35	0.01	1	3	6.45	1	3	0.05	1	3	36.27	35	630
d72	35	0.01	1	3	6.46	1	3	0.06	1	3	36.31	35	630
danwell	70	0.24	2	26	41.42	2	26	0.83	2	19	3533.82	2	19
j4j7	27	0	1	3	0.48	1	3	0.03	1	3	9.42	1	3
latin	28	0.06	18	424	1.53	18	424	0.09	18	226	20.86	18	226
latin1	28	0.05	18	424	1.52	18	424	0.09	18	226	20.79	18	226
nc3	18	0	1	8	0.28	1	8	0.01	1	7	1.05	1	7
nc4	32	0.03	1	18	2.68	1	18	0.06	1	15	27.83	1	15
nc42_1	42	0.06	1	9	5.91	1	9	0.18	1	8	189.24	1	8
nc5	50	0.11	1	26	20.67	1	26	0.22	1	20	308.32	1	20
nc6	72	0.33	1	40	84.6	1	40	0.83	1	31	2874.89	1	31
nc7	98	0.81	1	50	295.81	1	50	1.82	1	38	15010.73	1	38
r147	18	0	1	10	0.4	1	10	0.01	1	8	0.96	1	8
simple	8	0	2	11	0.01	2	11	0.00	2	9	0.03	2	9
sr81a	81	0.11	1	3	134.23	9	321	0.67	1	3	3538.02	9	185
sr81b	81	0.11	1	3	132.5	9	321	0.68	1	3	3569.58	9	185
youden	29	0.05	13	297	1.92	13	297	0.06	13	155	12.19	13	155

Table 5.6: Comparison of **qWeil**, **pointed**(2,1), **rücker**(0), and **rücker**(2)

Name	n	**qWeil**	s_f	r_f	**pointed**(2,1)	s_f	r_f	**rücker**(3)	s_f	r_f
had96_10	96	0.21	1	5	34.07	1	5	409484.07	8	165
had96_11	96	0.21	1	5	34.24	1	5	414051.91	6	84
had96_6	96	0.21	1	5	34.09	1	5	420123.59	2	8
had96_7	96	0.22	1	5	34.6	1	5	409314.11	2	7
had96_8	96	0.22	1	5	34.19	1	5	410756.48	5	65
had96_9	96	0.22	1	5	34.29	1	5	412873.92	5	77
nc6	72	0.33	1	40	84.6	1	40	64372.87	1	31
nc7	98	0.81	1	50	295.81	1	50	447997.33	1	38
nice_19_1_3	57	0.11	3	36	20.77	3	36	5968.26	3	24
prim_2_3	12	0	1	5	0.06	1	5	0.77	1	5
prim_2_4	16	0	1	5	0.12	1	5	5.41	1	5
prim_3_11	99	0.3	1	9	50	1	9	467345.29	1	8
prim_3_5	45	0.03	1	6	3.42	1	6	2896.52	1	6
prim_3_7	63	0.09	1	7	10.46	1	7	27109.53	1	7
prim_5_3	75	0.12	1	5	17.51	1	5	67235.18	1	5
r147	18	0	1	10	0.4	1	10	6.19	1	8
ramsey3_8_26	26	0.04	10	228	1.76	10	228	71.99	10	119
ramsey_36a	17	0.01	9	145	0.23	9	145	5.79	9	81
ramsey_36b	17	0.01	9	145	0.23	9	145	5.77	9	81
ramsey_36c	17	0.01	10	149	0.23	10	149	5.73	10	83
ramsey_36d	17	0.01	9	145	0.23	9	145	5.54	9	81
ramsey_36e	17	0.01	17	289	36.19	17	289	5.51	17	153
ramsey_36f	17	0.01	4	47	0.34	4	47	4.21	4	28
ramsey_36g	17	0.01	9	145	0.23	9	145	4.23	9	81
simple	8	0	2	11	0.01	2	11	0.09	2	9
sr81a	81	0.11	1	3	134.23	9	321	79939.74	9	185
sr81b	81	0.11	1	3	132.5	9	321	80159.68	9	185

Table 5.7: Comparison of **qWeil**, **pointed**(2,1) and **rücker**(3)

Chapter 6

Implementation Details and Computations

First, some ideas will be discussed which can be used to decrease the running time of the stabilization procedures considerably.

Instead of refining the coloring by considering all necessary colors c and then recoloring only once per step, we refine colorings permanently by considering single colors and recoloring immediately. This results in much more $\langle$recolor$\rangle$ operations and blasts the theoretical time bound but reduces the sizes of the color classes very quickly. Applying such a strategy leads to a practically much more efficient algorithm. Note that this method still colors canonically.

In practice it turns out that many structure lists of colors $L(c^k)$ do not yield a refinement of the coloring. It is easy to check whether a color class $\mathcal{C}(b)$ will be split by a structure list $L(c^k)$ or not. $\mathcal{C}(b)$ will not be split by $L(c^k)$ if and only if $p_{v^k}^{c^k}$ is constant on $v^k \in \mathcal{C}(b)$. This is the case when

$$\max\{p_{v^k}^{c^k} \mid v^k \in \mathcal{C}(b)\} \cdot |\mathcal{C}(b)| = \sum_{v^k \in \mathcal{C}(b)} p_{v^k}^{c^k},$$

holds. The values $\max\{p_{v^k}^{c^k} \mid v^k \in \mathcal{C}(b)\}$ and $\sum_{v^k \in \mathcal{C}(b)} p_{v^k}^{c^k}$ can be easily updated during the execution of $\langle$splitcolor$\rangle$. A structure list which refines the coloring is called *useful*. If we check this condition after having computed the structure list and then reduce the structure list if possible, the running time decreases.

6.1 Computations

6.1.1 2-stable Colorings

Due to the above considerations, it does not make sense to maintain the list $\mathcal{L}(c)$ throughout the execution of $\langle$step$\rangle$. $\mathcal{L}(c)$ may change after every recoloring. In

our implementation, we just store colors in $\mathcal{L}(c)$ and do not do any updating with $\mathcal{L}(c)$. This may cause the calculation of triangles which do not exist and thus enlarge the theoretical running time but it proves to be efficient in practice. For a better understanding, the implemented procedure is given below (see **Procedure 28**).

Procedure 28: step

```
   Examine triangles whose first non-basis edge was recently
   colored;
1: foreach c ∈ 𝒩 do
2:     foreach (u, w) ∈ colums(c) do
3:         for v ∈ {1, 2, ..., n} do
4:             append f(w, v) to the list ℒ(c);
           end
       end
5:     sort ℒ(c) and delete duplicates;
6:     foreach d ∈ ℒ(c) do
7:         splitcolor(c, d);
8:         recolor;
       end
9:     delete ℒ(c);
   end
   Treat the case were the second color is new analogously;
```

Algorithm 2-**stab** with the above extensions is implemented in a program called **qWeil**. This program uses **Procedure 28** as implementation of ⟨step⟩ and **Procedure 11** as implementation of ⟨splitcolor⟩. The remaining procedures are implemented as discussed in Section 3.4. The implementation of 2-**stab** described in Chapter 3, which has been proven to be time and space optimal, is denoted by **ts**. Both implementations have been tested on typical test instances and have been compared to the implementations **stabil**, **CC** and **stabcol**. Due to memory restrictions, we have not been able to test **stabcol** on instances with more than 200 nodes. The results of comprehensive tests are reported in Tables 6.1-6.5.

Each line of a table represents a test instance. It consists of the name of the instance and its number of vertices, the number of colors of the resulting 2-stable coloring and the running times of the different algorithms. The times are given in seconds and the bold entries mark the fastest algorithm for each instance. By ratio, we mean the running time compared to the running time of the leftmost algorithm. For a description of the test instances and the test environment see Section 6.2 and 6.3, respectively.

Table 6.1 shows that **qWeil** is by far the fastest canonical implementation.

On all instances, including the small ones, **qWeil** is faster than **CC** and **stabcol**. Table 6.2 shows that **ts** is faster than **stabcol** and thus the fastest implementation of an algorithm for computing coherent algebras with a theoretical time bound of $O(n^3 \log(n))$ up to our knowledge.

We have implemented also a non-canonical version of **qWeil** by just skipping the sorting at certain points in the algorithm. We denote this variant by **¬canonical**. On small instances both, **qWeil** and **¬canonical**, are often slower than **stabil**. However, on the large instances, **¬canonical** and even **qWeil** prove to be much faster than **stabil** (see Table 6.2). Observe that the differences between the running times of **qWeil** and **stabil** increase with increasing values of $\frac{\#\text{ colors}}{n}$.

In Table 6.3, some more variants of **qWeil** are listed. Algorithm **¬ir** denotes the algorithm which recolors only once per step, algorithm **¬useful** does not delete useless structure lists. Table 6.3 shows that the ideas presented at the beginning of this section are responsible for the good practical running time of **qWeil**. Observe that if we refrain from immediate recoloring, **qWeil** computes in each step the same number of new colors as the Weisfeiler-Leman algorithm and thus needs the same number of steps.

In Table 6.4, we compare the number of computed structure lists with the number of useful structure lists. It turns out that only very few structure lists are useful. This observation insinuates the design of algorithms which compute only few structure lists per step. Since we do not know which structure lists are useful, we implemented a randomized version of our algorithm which computes only few randomly chosen structure lists. Although, we have only little experience in this direction, the few results obtained so far are very promising. However, the major problem is that up to now we are not able to check reasonably fast whether the resulting coloring is stable or not. That means checking a given coloring for being 2-stable needs about the time as constructing the coloring from scratch.

Good results have been obtained with the two following randomized step procedures. The number of iterations of $\langle \text{step} \rangle$ is denoted by i. The function $\langle \text{randint}(k) \rangle$ chooses uniformly distributed an integer value out of the interval $[1, k]$.

Procedure 29: random step 1

```
1: f̄ ≡ f;
2: if first time in this procedure then
3: |   max_tries := n;
4: |   tries := 5;
   end
5: foreach k ∈ {1, 2, ..., tries} do
6: |   splitcolor(randint(r_f),randint(r_f));
   end
7: tries := min{r_f · i + 3, max_tries};
8: recolor;
```

Procedure 30: random step 2

```
1: f̄ ≡ f;
2: if first time in this procedure then
3: |   max_tries := n · (⌊√n⌋ + 1);
4: |   tries := min{r_f · 2 + 10, max_tries};
   end
5: foreach k ∈ {1, 2, ..., tries} do
6: |   foreach h ∈ {1, 2, ..., n} do
   |   |   splitcolor(f(randint(n),h), f(h,randint(n)));
   |   end
   end
7: tries := min{⌈r_f/n + n⌉, max_tries};
8: recolor;
```

The versions of **qWeil** using **Procedure 29** and **Procedure 30** instead of the standard step procedure are denoted by **random 1** and **random 2**, respectively.

In Table 6.5, these two step procedures are compared with the standard step procedure. **random 1** does not compute the correct results for the instances `dynkin`. In most of the cases, the randomized step procedure speed up the computation of a coloring. But there exist examples where the versions presented here slow the algorithm down quite a bit.

Name	n	Colors	Cells	**qWeil**	**CC**		**stabcol**		**ts**	
				time	time	ratio	time	ratio	time	ratio
benzene	18	36	3	**0**	0.01	-	0.02	-	0.05	-
benzene	36	144	6	**0.06**	0.09	1.500	0.23	3.833	0.56	9.333
benzene	54	324	9	**0.18**	0.28	1.556	0.87	4.833	5.39	29.944
benzene	72	576	12	**0.43**	0.74	1.721	2.62	6.093	14.84	34.512
benzene	90	900	15	**0.85**	1.45	1.706	5.59	6.576	31.78	37.388
benzene	108	1296	18	**1.4**	2.58	1.843	12.13	8.664	58.11	41.507
benzene	126	1764	21	**2.25**	4.26	1.893	20.10	8.933	97.96	43.538
benzene	144	2304	24	**3.29**	6.95	2.112	30.51	9.274	153.35	46.611
benzene	162	2916	27	**4.62**	9.64	2.087	45.20	9.784	224.03	48.491
benzene	180	3600	30	**6.59**	13.65	2.071	64.47	9.783	312.25	47.382
benzene	198	4356	33	**8.92**	18.97	2.127	92.98	10.424	448.82	50.316
benzene	204	4624	34	**9.97**	21.17	2.123	104.77	10.509		
benzene	240	6400	40	**18.55**	38.75	2.089	219.42	11.829		
benzene	276	8464	46	**30.59**	67.42	2.204	449.63	14.699		
benzene	312	10816	52	**45.71**	112.78	2.467	712.31	15.583		
benzene	348	13456	58	**66.04**	167.74	2.540	1061.62	16.075		
benzene	384	16384	64	**92.39**	245.72	2.660	1426.21	15.437		
benzene	402	17956	67	**102.09**	277.87	2.722	1625.96	15.927		
benzene	420	19600	70	**120.14**	329.2	2.740	1877.93	15.631		
benzene	456	23104	76	**146.73**	427.63	2.914	2390.23	16.290		
benzene	480	25600	80	**173.42**	531.74	3.066	2938.09	16.942		
benzene	492	26896	82	**191.3**	564.01	2.948	3295.12	17.225		
benzene	528	30976	88	**268.83**	751.64	2.796	4692.07	17.454		
benzene	558	34596	93	**316.9**	861.22	2.718	6524.80	20.589		
benzene	564	35344	94	**308.62**	868.15	2.813	7025.28	22.764		
dynkin	20	362	19	**0.03**	**0.03**	**1.000**	**0.03**	**1.000**	0.08	2.667
dynkin	40	1522	39	**0.23**	0.27	1.174	0.35	1.522	0.98	4.261
dynkin	60	3482	59	**0.75**	0.94	1.253	1.48	1.973	7.61	10.147
dynkin	80	6242	79	**1.95**	2.4	1.231	4.98	2.554	21.00	10.769
dynkin	100	9802	99	**3.8**	5.02	1.321	11.94	3.142	45.66	12.016
dynkin	120	14162	119	**7.04**	9.41	1.337	22.99	3.266	82.02	11.651
dynkin	140	19322	139	**12.54**	15.12	1.206	42.71	3.406	134.90	10.758
dynkin	160	25282	159	**18.97**	25.08	1.322	90.87	4.790	214.98	11.333
dynkin	180	32042	179	**29.18**	38.59	1.322	205.74	7.051	312.77	10.719
dynkin	200	39602	199	**42.69**	56.25	1.318	377.42	8.841	473.09	11.082
dynkin	240	57122	239	**77.43**	110.03	1.421	829.46	10.712		
dynkin	280	77842	279	**123.57**	194.86	1.577	1526.92	12.357		
dynkin	320	101762	319	**214.72**	332.43	1.548	2675.14	12.459		
dynkin	360	128882	359	**334.43**	533.28	1.595	4977.54	14.884		
dynkin	400	159202	399	**430.75**	759.08	1.762	9972.96	23.153		
dynkin	440	192722	439	**708.88**	1168.42	1.648	20523.56	28.952		
dynkin	480	229442	479	**889.61**	1405.94	1.580	41066.38	46.162		
dynkin	520	269362	519	**1278.53**	1819.9	1.423	76118.69	59.536		
dynkin	560	312482	559	**1218.98**	2572.28	2.110	105627.68	86.653		

continued on next page

continued from previous page

Name	n	Colors	Cells	**qWeil**	**ts**		**CC**		**stabcol**	
				time	time	ratio	time	ratio	time	ratio
moebius	20	11	1	**0**	0.01	-	0.03	-	0.06	-
moebius	40	21	1	**0.05**	0.09	1.800	0.39	7.800	0.78	15.600
moebius	60	31	1	**0.16**	0.31	1.938	1.41	8.812	5.54	34.625
moebius	80	41	1	**0.39**	0.83	2.128	4.26	10.923	16.57	42.487
moebius	100	51	1	**0.73**	1.63	2.233	8.62	11.808	34.98	47.918
moebius	120	61	1	**1.23**	2.88	2.341	15.21	12.366	76.39	62.106
moebius	140	71	1	**2.01**	4.8	2.388	28.99	14.423	102.49	50.990
moebius	160	81	1	**2.98**	7.67	2.574	44.95	15.084	170.20	57.114
moebius	180	91	1	**5.05**	11.79	2.335	65.52	12.974	237.80	47.089
moebius	200	101	1	**7.95**	18.16	2.284	90.49	11.382	332.25	41.792
moebius	240	121	1	**14.95**	37.13	2.484	160.15	10.712		
moebius	280	141	1	**30.8**	71.71	2.328	299.77	9.733		
moebius	320	161	1	**53.05**	140.95	2.657	465.53	8.775		
moebius	360	181	1	**78.46**	192.7	2.456	676.78	8.626		
moebius	400	201	1	**111.63**	280.15	2.510	934.81	8.374		
moebius	440	221	1	**158.66**	392.95	2.477	1258.45	7.932		
moebius	480	241	1	**225.71**	500.15	2.216	1651.00	7.315		
moebius	520	261	1	**371.14**	688.26	1.854	2385.57	6.428		
moebius	560	281	1	**450.86**			3057.44	6.781		
step	20	18	2	**0**	0.01	-	0.03	-	0.08	-
step	40	62	3	**0.05**	0.09	1.800	0.38	7.600	0.85	17.000
step	60	154	6	**0.17**	0.27	1.588	1.40	8.235	6.15	36.176
step	80	262	7	**0.41**	0.7	1.707	3.76	9.171	18.17	44.317
step	100	418	10	**0.76**	1.4	1.842	9.96	13.105	40.26	52.974
step	120	590	11	**1.29**	2.47	1.915	19.10	14.806	73.70	57.132
step	140	810	14	**2.05**	3.79	1.849	32.02	15.620	136.53	66.600
step	160	1046	15	**2.89**	5.68	1.965	55.13	19.076	208.88	72.277
step	180	1330	18	**4.29**	8.44	1.967	79.83	18.608	288.13	67.163
step	200	1630	19	**5.71**	12.21	2.138	122.65	21.480	404.97	70.923
step	240	2342	23	**10.79**	24.17	2.240	237.74	22.033		
step	280	3182	27	**18.62**	46.5	2.497	423.44	22.741		
step	320	4150	31	**30.08**	77.07	2.562	711.77	23.663		
step	360	5246	35	**45.02**	120.3	2.672	1138.03	25.278		
step	400	6470	39	**58.2**	151.4	2.601	1735.95	29.827		
step	440	7822	43	**75.81**	204.87	2.702	2554.46	33.696		
step	480	9302	47	**110.98**	301.18	2.714	3613.48	32.560		
step	520	10910	51	**130.73**	417.94	3.197	4969.97	38.017		
step	560	12646	55	**184.77**	548.76	2.970	6759.82	36.585		

Table 6.1: Canonical Implementations

Name	Vertices	Colors	Cells	¬**canonical**	**stabil**		**qWeil**	
				time	time	ratio	time	ratio
benzene	18	36	3	**0**	**0.00**	**-**	**0**	**-**
benzene	36	144	6	**0.05**	**0.05**	**1.000**	0.06	1.200
benzene	54	324	9	**0.15**	0.22	1.467	0.18	1.200
benzene	72	576	12	**0.37**	0.63	1.703	0.43	1.162
benzene	90	900	15	**0.75**	1.38	1.840	0.85	1.133
benzene	108	1296	18	**1.22**	2.72	2.230	1.4	1.148
benzene	126	1764	21	**1.88**	4.80	2.553	2.25	1.197
benzene	144	2304	24	**2.87**	7.95	2.770	3.29	1.146
benzene	162	2916	27	**4.02**	12.14	3.020	4.62	1.149
benzene	180	3600	30	**5.46**	18.65	3.416	6.59	1.207
benzene	198	4356	33	**7.98**	26.12	3.273	8.92	1.118
benzene	204	4624	34	**8.62**	29.78	3.455	9.97	1.157
benzene	240	6400	40	**15.66**	55.68	3.556	18.55	1.185
benzene	276	8464	46	**25.45**	95.57	3.755	30.59	1.202
benzene	312	10816	52	**38.05**	153.96	4.046	45.71	1.201
benzene	348	13456	58	**55.5**	236.15	4.255	66.04	1.190
benzene	384	16384	64	**80.02**	348.99	4.361	92.39	1.155
benzene	402	17956	67	**90.21**	411.88	4.566	102.09	1.132
benzene	420	19600	70	**106.02**	501.89	4.734	120.14	1.133
benzene	456	23104	76	**137.35**	693.30	5.048	146.73	1.068
benzene	480	25600	80	**162.76**	854.67	5.251	173.42	1.065
benzene	492	26896	82	**178.99**	927.07	5.179	191.3	1.069
benzene	528	30976	88	**221.69**	1237.62	5.583	268.83	1.213
benzene	558	34596	93	**260.08**	1503.83	5.782	316.9	1.218
benzene	564	35344	94	**257.42**	1593.04	6.188	308.62	1.199
dynkin	20	362	19	0.02	**0.01**	**0.500**	0.03	1.500
dynkin	40	1522	39	0.13	**0.09**	**0.692**	0.23	1.769
dynkin	60	3482	59	0.43	**0.41**	**0.953**	0.75	1.744
dynkin	80	6242	79	**1.02**	1.20	1.176	1.95	1.912
dynkin	100	9802	99	**1.99**	2.73	1.372	3.8	1.910
dynkin	120	14162	119	**4.47**	5.61	1.255	7.04	1.575
dynkin	140	19322	139	**7.73**	11.37	1.471	12.54	1.622
dynkin	160	25282	159	**13.45**	21.94	1.631	18.97	1.410
dynkin	180	32042	179	**19.68**	40.61	2.064	29.18	1.483
dynkin	200	39602	199	**29.85**	71.88	2.408	42.69	1.430
dynkin	240	57122	239	**58.3**	203.19	3.485	77.43	1.328
dynkin	280	77842	279	**78.37**	555.00	7.082	123.57	1.577
dynkin	320	101762	319	**126.74**	1526.98	12.048	214.72	1.694
dynkin	360	128882	359	**193.61**	3953.94	20.422	334.43	1.727
dynkin	400	159202	399	**271.38**	9337.60	34.408	430.75	1.587
dynkin	440	192722	439	**370.03**	20478.01	55.341	708.88	1.916
dynkin	480	229442	479	**481.42**			889.61	1.848
dynkin	520	269362	519	**642.68**			1278.53	1.989
dynkin	560	312482	559	**833.66**			1218.98	1.462

continued on next page

continued from previous page

Name	Vertices	Colors	Cells	¬**canonical**	**stabil**		**qWeil**	
				time	time	ratio	time	ratio
moebius	20	11	1	**0**	**0.00**	**-**	**0**	**-**
moebius	40	21	1	**0.05**	0.06	1.200	**0.05**	**1.000**
moebius	60	31	1	**0.15**	0.22	1.467	0.16	1.067
moebius	80	41	1	**0.36**	0.57	1.583	0.39	1.083
moebius	100	51	1	**0.68**	1.18	1.735	0.73	1.074
moebius	120	61	1	**1.17**	2.12	1.812	1.23	1.051
moebius	140	71	1	**1.91**	3.51	1.838	2.01	1.052
moebius	160	81	1	**2.89**	5.46	1.889	2.98	1.031
moebius	180	91	1	**4.88**	8.43	1.727	5.05	1.035
moebius	200	101	1	**7.25**	11.52	1.589	7.95	1.097
moebius	240	121	1	**14.62**	21.19	1.449	14.95	1.023
moebius	280	141	1	**29.28**	35.57	1.215	30.8	1.052
moebius	320	161	1	54.1	56.80	1.050	**53.05**	**0.981**
moebius	360	181	1	80.31	84.50	1.052	**78.46**	**0.977**
moebius	400	201	1	**103.98**	122.96	1.183	111.63	1.074
moebius	440	221	1	**145.06**	172.89	1.192	158.66	1.094
moebius	480	241	1	**211.02**	234.10	1.109	225.71	1.070
moebius	520	261	1	**244.81**	305.05	1.246	371.14	1.516
moebius	560	281	1	**280.32**	396.25	1.414	450.86	1.608
step	20	18	2	**0**	**0.00**	**-**	**0**	**-**
step	40	62	3	**0.05**	0.06	1.200	**0.05**	**1.000**
step	60	154	6	**0.16**	0.23	1.438	0.17	1.062
step	80	262	7	**0.4**	0.59	1.475	0.41	1.025
step	100	418	10	**0.67**	1.24	1.851	0.76	1.134
step	120	590	11	**1.24**	2.28	1.839	1.29	1.040
step	140	810	14	**1.94**	3.92	2.021	2.05	1.057
step	160	1046	15	**2.74**	6.18	2.255	2.89	1.055
step	180	1330	18	**4.11**	9.66	2.350	4.29	1.044
step	200	1630	19	5.77	16.62	2.880	**5.71**	**0.990**
step	240	2342	23	**10.77**	37.85	3.514	10.79	1.002
step	280	3182	27	18.85	66.01	3.502	**18.62**	**0.988**
step	320	4150	31	**29.34**	126.42	4.309	30.08	1.025
step	360	5246	35	**43.62**	222.46	5.100	45.02	1.032
step	400	6470	39	**56.05**	374.60	6.683	58.2	1.038
step	440	7822	43	**68.59**	597.37	8.709	75.81	1.105
step	480	9302	47	**93.79**	917.41	9.782	110.98	1.183
step	520	10910	51	**117.83**	1380.94	11.720	130.73	1.109
step	560	12646	55	**158.81**	1982.39	12.483	184.77	1.163

Table 6.2: Non Canonical Implementations

Name	n	Colors	Cells	**qWeil**	**¬useful**		**¬ir**		**¬ir ¬useful**	
				time	time	ratio	time	ratio	time	ratio
benzene	18	36	3	**0**	0.01	-	0.01	-	0.02	-
benzene	36	144	6	**0.06**	0.14	2.333	0.09	1.500	0.19	3.167
benzene	54	324	9	**0.18**	0.44	2.444	0.28	1.556	0.64	3.556
benzene	72	576	12	**0.43**	1.03	2.395	0.74	1.721	1.67	3.884
benzene	90	900	15	**0.85**	2.08	2.447	1.45	1.706	3.33	3.918
benzene	108	1296	18	**1.4**	3.45	2.464	2.58	1.843	5.85	4.179
benzene	126	1764	21	**2.25**	5.48	2.436	4.26	1.893	9.73	4.324
benzene	144	2304	24	**3.29**	8.03	2.441	6.95	2.112	15.41	4.684
benzene	162	2916	27	**4.62**	11.45	2.478	9.64	2.087	21.73	4.703
benzene	180	3600	30	**6.59**	15.98	2.425	13.65	2.071	30.63	4.648
benzene	198	4356	33	**8.92**	21.49	2.409	18.97	2.127	42.5	4.765
benzene	204	4624	34	**9.97**	23.81	2.388	21.17	2.123	46.63	4.677
benzene	240	6400	40	**18.55**	43.41	2.340	38.75	2.089	85.38	4.603
benzene	276	8464	46	**30.59**	70.46	2.303	67.42	2.204	138.32	4.522
benzene	312	10816	52	**45.71**	103.49	2.264	112.78	2.467	220.41	4.822
benzene	348	13456	58	**66.04**	147.84	2.239	167.74	2.540	329.43	4.988
benzene	384	16384	64	**92.39**	197.12	2.134	245.72	2.660	479.6	5.191
benzene	402	17956	67	**102.09**	225.09	2.205	277.87	2.722	538.24	5.272
benzene	420	19600	70	**120.14**	258.76	2.154	329.2	2.740	632.96	5.269
benzene	456	23104	76	**146.73**	322.48	2.198	427.63	2.914	835.54	5.694
benzene	480	25600	80	**173.42**	376.35	2.170	531.74	3.066	989.37	5.705
benzene	492	26896	82	**191.3**	408.34	2.135	564.01	2.948	1113.59	5.821
benzene	528	30976	88	**268.83**	573.3	2.133	751.64	2.796	1367.24	5.086
benzene	558	34596	93	**316.9**	650.51	2.053	861.22	2.718	1649.49	5.205
benzene	564	35344	94	**308.62**	667.28	2.162	868.15	2.813	1713.07	5.551
dynkin	20	362	19	**0.03**	0.05	1.667	**0.03**	**1.000**	0.06	2.000
dynkin	40	1522	39	**0.23**	0.42	1.826	0.27	1.174	0.54	2.348
dynkin	60	3482	59	**0.75**	1.41	1.880	0.94	1.253	1.92	2.560
dynkin	80	6242	79	**1.95**	3.63	1.862	2.4	1.231	4.69	2.405
dynkin	100	9802	99	**3.8**	7.37	1.939	5.02	1.321	9.91	2.608
dynkin	120	14162	119	**7.04**	13.32	1.892	9.41	1.337	17.65	2.507
dynkin	140	19322	139	**12.54**	23.55	1.878	15.12	1.206	28.3	2.257
dynkin	160	25282	159	**18.97**	36.05	1.900	25.08	1.322	45.75	2.412
dynkin	180	32042	179	**29.18**	53.68	1.840	38.59	1.322	70.76	2.425
dynkin	200	39602	199	**42.69**	80.44	1.884	56.25	1.318	104.25	2.442
dynkin	240	57122	239	**77.43**	137.5	1.776	110.03	1.421	195.37	2.523
dynkin	280	77842	279	**123.57**	199.5	1.614	194.86	1.577	331.71	2.684
dynkin	320	101762	319	**214.72**	407.25	1.897	332.43	1.548	569.92	2.654
dynkin	360	128882	359	**334.43**	581.46	1.739	533.28	1.595	888.35	2.656
dynkin	400	159202	399	**430.75**	759.13	1.762	759.08	1.762	1266.08	2.939
dynkin	440	192722	439	**708.88**	1200.59	1.694	1168.42	1.648	1828.28	2.579
dynkin	480	229442	479	**889.61**	1414.33	1.590	1405.94	1.580	2278.56	2.561
dynkin	520	269362	519	**1278.53**	2145.43	1.678	1819.9	1.423	2938.52	2.298
dynkin	560	312482	559	**1218.98**	1971.99	1.618	2572.28	2.110	3964.8	3.253

continued on next page

continued from previous page

Name	n	Colors	Cells	**qWeil**	**¬useful**		**¬ir**		**¬ir ¬useful**	
				time	time	ratio	time	ratio	time	ratio
moebius	20	11	1	**0**	0.01	-	0.01	-	0.02	-
moebius	40	21	1	**0.05**	0.13	2.600	0.09	1.800	0.24	4.800
moebius	60	31	1	**0.16**	0.45	2.812	0.31	1.938	0.84	5.250
moebius	80	41	1	**0.39**	1.14	2.923	0.83	2.128	2.24	5.744
moebius	100	51	1	**0.73**	2.25	3.082	1.63	2.233	4.5	6.164
moebius	120	61	1	**1.23**	4	3.252	2.88	2.341	7.71	6.268
moebius	140	71	1	**2.01**	6.65	3.308	4.8	2.388	13	6.468
moebius	160	81	1	**2.98**	9.91	3.326	7.67	2.574	21.65	7.265
moebius	180	91	1	**5.05**	15.29	3.028	11.79	2.335	32.46	6.428
moebius	200	101	1	**7.95**	27.01	3.397	18.16	2.284	52.56	6.611
moebius	240	121	1	**14.95**	52.05	3.482	37.13	2.484	97.79	6.541
moebius	280	141	1	**30.8**	90.51	2.939	71.71	2.328	191.04	6.203
moebius	320	161	1	**53.05**	165.82	3.126	140.95	2.657	284.97	5.372
moebius	360	181	1	**78.46**	242.58	3.092	192.7	2.456	447.23	5.700
moebius	400	201	1	**111.63**	340.48	3.050	280.15	2.510	685.23	6.138
moebius	440	221	1	**158.66**	502.07	3.164	392.95	2.477	919.7	5.797
moebius	480	241	1	**225.71**	816.58	3.618	500.15	2.216	1469.16	6.509
moebius	520	261	1	**371.14**	1001.24	2.698	688.26	1.854	1903.12	5.128
step	20	18	2	**0**	0.01	-	0.01	-	0.02	-
step	40	62	3	**0.05**	0.11	2.200	0.09	1.800	0.2	4.000
step	60	154	6	**0.17**	0.4	2.353	0.27	1.588	0.63	3.706
step	80	262	7	**0.41**	0.94	2.293	0.7	1.707	1.53	3.732
step	100	418	10	**0.76**	1.61	2.118	1.4	1.842	2.99	3.934
step	120	590	11	**1.29**	2.88	2.233	2.47	1.915	5.23	4.054
step	140	810	14	**2.05**	4.47	2.180	3.79	1.849	7.73	3.771
step	160	1046	15	**2.89**	6.37	2.204	5.68	1.965	11.47	3.969
step	180	1330	18	**4.29**	9.15	2.133	8.44	1.967	16.76	3.907
step	200	1630	19	**5.71**	12.29	2.152	12.21	2.138	24.52	4.294
step	240	2342	23	**10.79**	22.3	2.067	24.17	2.240	46.55	4.314
step	280	3182	27	**18.62**	36.89	1.981	46.5	2.497	85.18	4.575
step	320	4150	31	**30.08**	60.3	2.005	77.07	2.562	136.44	4.536
step	360	5246	35	**45.02**	86.98	1.932	120.3	2.672	186.9	4.151
step	400	6470	39	**58.2**	115.32	1.981	151.4	2.601	268.55	4.614
step	440	7822	43	**75.81**	148.01	1.952	204.87	2.702	362.92	4.787
step	480	9302	47	**110.98**	217.03	1.956	301.18	2.714	539.93	4.865
step	520	10910	51	**130.73**	259.77	1.987	417.94	3.197	693.9	5.308
step	560	12646	55	**184.77**	363.79	1.969	548.76	2.970	957.16	5.180

Table 6.3: Variants of **qWeil**

Name	Vertices	Colors	Structure Lists		
			overall	useful	%
benzene	18	36	511	21	4.110
benzene	60	400	18754	287	1.530
benzene	96	1024	75868	757	0.998
benzene	198	4356	649837	3112	0.479
benzene	300	10000	2477605	7560	0.305
benzene	396	17424	5375700	14242	0.265
dynkin	20	362	8808	266	3.020
dynkin	60	3482	253185	3145	1.242
dynkin	100	9802	1215970	9129	0.751
dynkin	200	39602	9795498	37832	0.386
dynkin	300	89402	33462760	83709	0.250
dynkin	400	159202	75681945	152072	0.201
moebius	20	11	120	6	5.000
moebius	60	31	960	26	2.708
moebius	100	51	2600	46	1.769
moebius	200	101	10200	95	0.931
moebius	300	151	22800	141	0.618
moebius	400	201	40400	187	0.463
path	20	200	4858	160	3.294
path	60	1800	129139	1254	0.971
path	100	5000	582057	4076	0.700
path	200	20000	4520545	18322	0.405
path	300	45000	16399733	35839	0.219
path	400	80000	36016087	67137	0.186
step	20	18	323	10	3.096
step	60	154	5084	97	1.908
step	100	418	27342	287	1.050
step	200	1630	163646	1336	0.816
step	300	3658	534460	2969	0.556
step	400	6470	1281129	5666	0.442

Table 6.4: Structure Lists Statistics

Name	Vertices	Colors	Cells	**qWeil**	**random 2**	
				ratio	time	ratio
dynkin	20	362	19	0.02	**0.01**	**0.500**
dynkin	40	1522	39	0.13	**0.07**	**0.538**
dynkin	60	3482	59	0.43	**0.21**	**0.488**
dynkin	80	6242	79	0.99	**0.47**	**0.475**
dynkin	100	9802	99	1.96	**0.98**	**0.500**
dynkin	120	14162	119	4.35	**1.92**	**0.441**
dynkin	140	19322	139	7.57	**3.06**	**0.404**
dynkin	160	25282	159	13.8	**4.57**	**0.331**
dynkin	180	32042	179	21.22	**7.85**	**0.370**
dynkin	200	39602	199	32.37	**10.58**	**0.327**
dynkin	240	57122	239	62.6	**18.35**	**0.293**
dynkin	280	77842	279	84.93	**31.04**	**0.365**
dynkin	320	101762	319	121.38	**51.54**	**0.425**
dynkin	360	128882	359	189.6	**103.72**	**0.547**
dynkin	400	159202	399	271.65	**136.7**	**0.503**
dynkin	440	192722	439	351.13	**172.02**	**0.490**
dynkin	480	229442	479	518.1	**237.41**	**0.458**
dynkin	520	269362	519	780.15	**325.83**	**0.418**
dynkin	560	312482	559	954.54	**437.37**	**0.458**

Name	Vertices	Colors	Cells	**qWeil**	**random 1**		**random 2**	
				time	time	ratio	time	ratio
moebius	20	11	1	**0**	**0**	**-**	0.02	-
moebius	40	21	1	0.05	**0.03**	**0.600**	0.21	4.200
moebius	60	31	1	0.15	**0.06**	**0.400**	0.7	4.667
moebius	80	41	1	0.36	**0.15**	**0.417**	1.79	4.972
moebius	100	51	1	0.69	**0.26**	**0.377**	3.2	4.638
moebius	120	61	1	1.18	**0.39**	**0.331**	5.64	4.780
moebius	140	71	1	1.92	**0.56**	**0.292**	9.16	4.771
moebius	160	81	1	2.89	**0.86**	**0.298**	14.25	4.931
moebius	180	91	1	4.86	**1.44**	**0.296**	34.65	7.130
moebius	200	101	1	8.16	**1.73**	**0.212**	39.63	4.857
moebius	240	121	1	17.63	**3.92**	**0.222**	128.59	7.294
moebius	280	141	1	27.01	**4.97**	**0.184**	178.25	6.599
moebius	320	161	1	46.48	**7.64**	**0.164**	357.91	7.700
moebius	360	181	1	86.77	**16.74**	**0.193**	700.75	8.076
moebius	400	201	1	120.55	**16.09**	**0.133**	772.51	6.408
moebius	440	221	1	206.04	**26.88**	**0.130**	1415.15	6.868
moebius	480	241	1	212.37	**22.9**	**0.108**	1402.52	6.604
moebius	520	261	1	390.4	**38.81**	**0.099**	1964.23	5.031
moebius	560	281	1	524.02	**38.05**	**0.073**	2764.23	5.275

Table 6.5: Random step procedures

6.1.2 1-stable Colorings

To give an idea of the performance of the implementation of 1-**stab**, denoted by **qStab**, we have tested it on several graph classes which have been already used as test objects above. **Table 6.6** shows the number of colors of the coarsest 1-stable partition for each instance and compares the running times of the non-canonical and canonical algorithms implemented in **qStab**. The CPU times are given in seconds.

			qStab		
Name	Vertices	Colors	¬ **canonical**	**canonical**	
			time	time	ratio
benzene	1602	267	**0.01**	0.02	2.000
benzene	2400	400	**0.01**	0.02	2.000
benzene	3198	533	**0.01**	0.02	2.000
benzene	3996	666	**0.02**	0.03	1.500
benzene	4002	667	**0.02**	0.03	1.500
benzene	12000	2000	**0.06**	0.14	2.333
benzene	19998	3333	**0.11**	0.37	3.364
benzene	27996	4666	**0.15**	0.66	4.400
benzene	40002	6667	**0.22**	1.25	5.682
benzene	120000	20000	**0.73**	9.77	13.384
benzene	199998	33333	**1.25**	25.97	20.776
dynkin	1600	1599	**0.01**	0.06	6.000
dynkin	2400	2399	**0.01**	0.13	13.000
dynkin	3200	3199	**0.02**	0.23	11.500
dynkin	4000	3999	**0.03**	0.37	12.333
dynkin	12000	11999	**0.09**	3.25	36.111
dynkin	20000	19999	**0.15**	8.93	59.533
dynkin	28000	27999	**0.2**	17.41	87.050
dynkin	36000	35999	**0.26**	28.72	110.462
dynkin	40000	39999	**0.3**	35.42	118.067
dynkin	120000	119999	**0.96**	395.06	411.521
dynkin	200000	199999	**1.6**	2225.59	1390.994

Table 6.6: Comparison of running times of a canonical and non-canonical version of **qStab**

Besides on the number of vertices, the computation time depends highly on the number of colors of the resulting partitions. Again the canonical implementation is slower than the non-canonical one.

For a comparison of **qStab** to the algorithm implemented in **nauty**, see Sections 5.2.3 and 5.1.4. **nauty** cannot handle graphs with vertex number larger than about 100000.

6.2 The Test Instances

The instances `benzene`, `moebius` and `dynkin` have been suggested in [7] and are described below. `step` denotes the instances given by Fürer [31] for which **2-stab** requires a number of steps proportional to n and which have been described in Section 4.5.1. `path` and `cycle` denote the instances which are paths and cycles, respectively.

Path and Cycle

$\mathbf{P}_n$ is the path on n vertices, i.e,

$$\mathbf{P}_n = (V_n, \{[v_i, v_{i+1}] \mid i \in \{1, 2, \ldots, n-1\}\})$$

and $\mathbf{C}_n$ is the simple cycle on n vertices, i.e.,

$$\mathbf{C}_n = (V_n, \{[v_i, v_{(i+1 \mod n)}] \mid i \in \{1, 2, \ldots, n-1\}\} \cup \{[v_1, v_n]\}).$$

Dynkin Graphs

$\mathbf{D}_n$ is the so called Dynkin graph on n vertices (see **Figure 6.1**), i.e.,

$$\mathbf{D}_n = (V_n, \{[v_i, v_{i+1}] \mid i \in \{1, 2, \ldots, n-2\}\} \cup \{[v_{n-2}, v_n]\}).$$

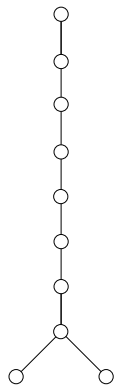


Figure 6.1: The dynkin graph with 10 vertices

Benzene Stacks

$\mathbf{B}_n$ is the benzene stack on n vertices (see **Figure 6.2**). $\mathbf{B}_n$ is only defined for n divisible by 6. Let $C_j = [v_{i+j}, v_{(i+(j+1 \bmod 6))}]$, $L_j = [v_j, v_{j+6}]$, and $c = \frac{n}{6}$.

$$\mathbf{B}_n = (V_n, \bigcup_{j \in \{0,6,12,\ldots,n-6\}} C_j \cup \bigcup_{j \in \{1,2,\ldots,n-6\}} L_j)$$

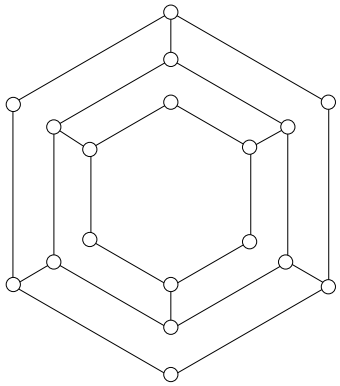
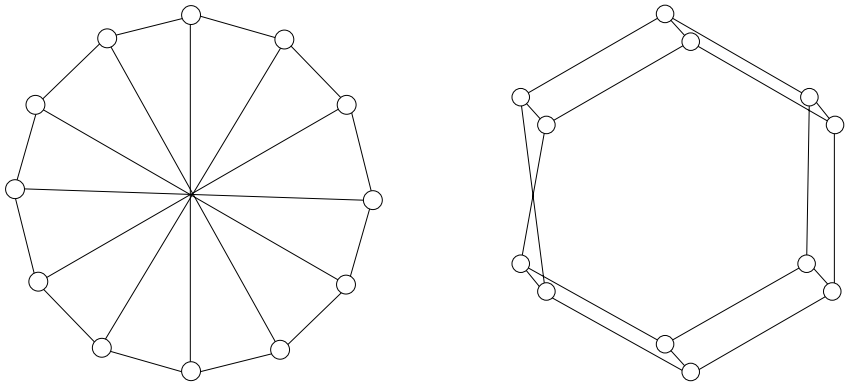


Figure 6.2: The benzene stack with 18 vertices

Möbius

We denote by $\mathbf{M}_n$ the Möbius graph on n vertices (see **Figure 6.3**). $\mathbf{M}_n$ is only defined for even numbers.

$$\mathbf{M}_n = (V_n, \{\{v_i, v_j\} \mid j - i \equiv k \,(\mathrm{mod}\, n), k \in \{1, n/2, n-1\}\})$$



(a) (b)

Figure 6.3: Two drawings of the Möbius ladder with 12 vertices

6.3 Testing Environment

I am grateful to all those who provided me with their programs for the aims of comparison to my programs. Furthermore, I used some libraries and compilers from the open source community. I thank the respective authors for writing great

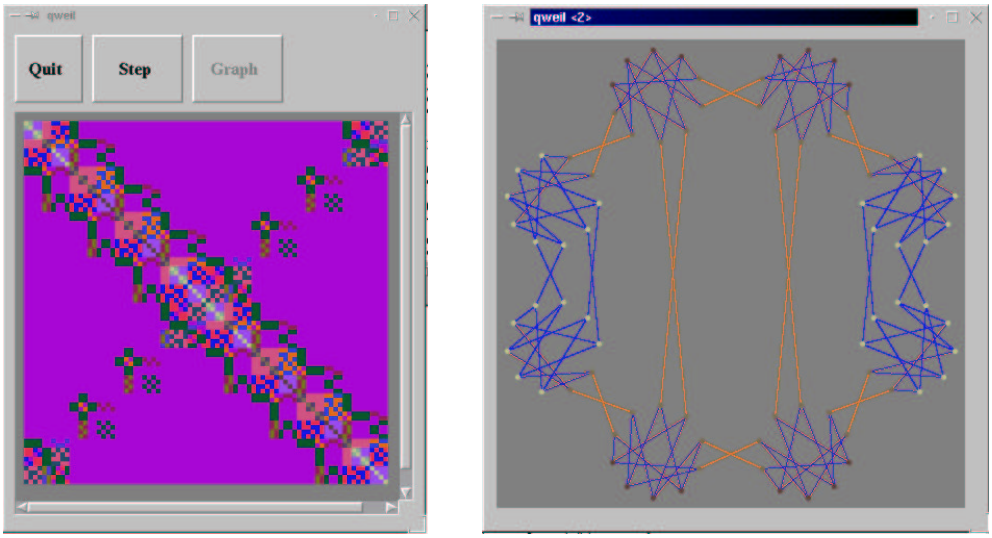
software. Some of the packages mentioned in this section are available directly in the world wide web. If applicable, we give the address.

The results have been obtained on an SGI IP27, 1 GB memory using the NCC compiler version 7.3.1.1m with optimization switched on, except for the program **rücker** by G. Rücker and Ch. Rücker where we used g77, version 2.7.2.2.

6.3.1 qStab and qWeil

The algorithms presented in this thesis and implemented in the programs **qStab** and **qWeil** are written in C++. The programs are available at `http://www-m9.mathematik.tu-muenchen.de/~bastert`.

All parameters of **qStab** are defined via command line parameters. To visualize the results, an elementary graphical user interface was written making use of the Qt library (`http://www.troll.no`) by troll tech.



(a) The colored matrix (b) and the colored graph

Figure 6.4: The colored instance `step 80` after 3 iterations of **qWeil**

6.3.2 Other Programs

The previously mentioned programs **stabcol** and **stabil** [6, 7] are available at `http://www-m9.mathematik.tu-muenchen.de/~bastert/wl.html`. The program **CC** [58] can be obtained directly from the author (`inp@pdmi.ras.ru`)

The program **rücker** [61, 62] is available from the authors (`cruecker@orgmail.chemie.uni-freiburg.de`) on request.

nauty is a very impressive program package for solving graph isomorphism, automorphism and canonical labeling problems (`http://cs.anu.edu.au/people/bdm/nauty/`), written by B. D. McKay [49].

During the development of the algorithms, we made use of LEDA [51] (`http://www.mpi-sb.mpg.de/LEDA/`), a library of efficient data structures. In the current versions, LEDA is not used anymore. All the development work has been made on a computer powered by Linux. The pictures are drawn in xfig and the typesetting is made in LaTeX2e.

Chapter 7

Conclusions

We have presented a new canonical algorithm for computing coherent colorings which is the first one to satisfy the best known time and space bound simultaneously. Furthermore, we have shown that our implementation is by far the fastest known for computing coherent colorings. In addition, the algorithm for computing the coarsest 1-stable partition is competitive and is in particular fast on large instances. We have introduced new k-dimensional stabilization algorithms which can be implemented more efficiently than the known ones.

The algorithm for computing the coarsest 1-stable partition was designed to tackle eigenvalue problems of large graphs. It was used in a framework for computing the eigenvalues of Robinson graphs which play an important role in the reconstruction of phylogenies.

As a contribution to the structural analysis of Robinson graphs, we determined the cells of the coherent algebra generated by Robinson graphs.

We discussed several variations of stabilization procedures. In particular, we examined one algorithm used in chemistry in more detail.

From an algorithmic point of view, two open questions are of particular interest. M. E. Muzychuk raised the question how to compute coarsest non-trivial equitable partitions of regular graphs. There is a lower bound on the maximum number of iterations in **2-stab** of $\Omega(n)$ and no upper bound, except the trivial one of $O(n^2)$. It would be interesting to close this gap.

Appendix A

The Complete Algorithm

In this section, we state the space optimal algorithm for 2-dimensional stabilization with time bound $O(n^3 \log(n))$ (**ts**) presented in Section 3.4 as a whole.

Algorithm 31: 2-**stab** (**ts**)

Data : $G_f = (V, E, f)$, f proper

Result: The coarsest 2-stable coloring f^2 of G_f

```
1:  f² ≡ f;
2:  repeat
3:      f̄ ≡ f²;
4:      foreach c ∈ 𝒩 do
5:          ℒ(c) = ∅;
6:          foreach (u, w) ∈ columns(c) do
7:              for (v = 1; v ≤ vertices; v++) do
8:                  if (w, v) is first of its color-class in the row of M corresponding to w then
9:                      append ((u, w), (w, v)) to the list ℒ(c);
                    end
                end
            end
10:         sort ℒ(c) by the colors of the second edges;
11:         foreach d ∈ ℒ(c) do
12:             splitcolor(c, d);
            end
13:         delete ℒ(c);
        end
14:     foreach d ∈ 𝒩 do
15:         ℒ(d) = ∅;
16:         foreach (w, v) ∈ rows(c) do
17:             for (u = 1; v ≤ vertices; u++) do
18:                 if (u, w) is first of its color-class in the row of M corresponding to w and f(u, w) ∈ 𝒪 then
19:                     append ((u, w), (w, v)) to the list ℒ(d);
                    end
                end
            end
20:         sort ℒ(d) by the colors of the first edges;
21:         foreach c ∈ ℒ(d) do
22:             splitcolor(c, d);
            end
23:         delete ℒ(d);
        end
24:     recolor;
    until r_{f²} did not change;
```

Procedure 32: splitcolor(c, d)

```
1:  compute L(c,d);
2:  sort L(c,d) by the values p_e^{c,d};
3:  foreach e with e first edge in L(c,d) with color f̄(e) do
4:      b := f̄(e);
5:      if C(b).hit < C(b).size then
6:          C(b).current_p := 0;
        else
7:          C(b).current_p := p_e^{c,d};
        end
8:      C(b).current_color := b;
9:      C(b).hit := 0;
    end
10: foreach e ∈ L(c,d) do
11:     if C(f̄(e)).current_p ≠ p_e^{c,d} then
12:         C(f̄(e)).current_p := p_e^{c,d};
13:         C(f̄(e)).current_color := n_{f̄} + 1;
        end
14:     C(f̄(e)).size − −;
15:     delete e from its parent's children list;
16:     append e to the children list of C(f̄(e));
17:     parent(e) := C(f̄(e));
18:     f̄(e) := C(f̄(e)).current_color;
19:     C(f̄(e)).size + +;
    end
```

Procedure 33: compute $L(c, d)$

```
1:  foreach w ∈ columnindices(c) ∩ rowindices(d) do
2:      foreach e = (u,v) with (u,w) ∈ C(c) and (w,v) ∈ C(d) do
3:          if p_e^{c,d} = 0 then
4:              C(f̄(e)).hit + +;
5:              append (e, p_e^{c,d}) to L(c,d);
            end
6:          p_e^{c,d} + +;
        end
    end
```

Procedure 34: recolor

Let L be the list of all edges which got a new pseudo color;
sort L by the old colors f;
introduce a dummy root r;
foreach $e \in L, f(parent(e)) \leq n_f$ **do**
    assign e as child of r;
    assign r as parent of e;
end
Let $L' := \{f(v) | v \in L\}$;
`the parent-children relationship defines a tree` T `on the color-classes in` L.;
assign new colors $n_f + 1, \ldots, n_{\overline{f}}$ to the color-classes in T and to the edges in L by walking through T in post order (or some other well defined order);
foreach $e \in L$ *(pass though L in the described order)* **do**
    delete e from its color-class $\mathcal{C}(f(e))$;
    append e to $\mathcal{C}(\overline{f}(e))$;
end
sort L by the tuples $(rowindex(e), columnindex(e))$ and generate with the help of this ordering the row encodings of the new color-classes;
sort L by the tuples $(columnindex(e), rowindex(e))$ and generate with the help of this ordering the column encodings of the new color-classes;
foreach $c \in L'$ **do**
    find $d \in \mathcal{N}_c$ with $|\mathcal{C}(d)| = \max_{d' \in \mathcal{N}_c} |\mathcal{C}(d')|$;
    if $|\mathcal{C}(d)| > |\mathcal{C}(c)|$ **then**
        exchange the colors of the color-classes $\mathcal{C}(c)$ and $\mathcal{C}(d)$;
    end
end
$f^2 \equiv \overline{f}$;

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] M. Ajtai. Recursive construction for 3-regular expanders. *Combinatorica*, 14(4):379–416, 1994.

[3] L. Babai, P. Erdös, and S. M. Selkow. Random graph isomorphism. *Journal on Computing*, 9:628–635, 1980.

[4] L. Babai and L. Kŭcera. Canonical labeling of graphs in linear average time. In *20th IEEE Symposium on Foundations of Computer Science*, volume 20, pages 39–46, 1979.

[5] L. Babel. Computing coherent algebras. submitted, 1995.

[6] L. Babel, S. Baumann, M. Lüdecke, and G. Tinhofer. STABCOL: Graph isomorphism testing based on the Weisfeiler-Leman Algorithm. Technical Report TUM-M9702, Technische Universität München, 1997.

[7] L. Babel, I. V. Chuvaeva, M. Klin, and D. V. Pasechnik. Algebraic combinatorics in mathematical chemistry. Methods and algorithms. II. Program implementation of the Weisfeiler-Leman Algorithm. Technical Report TUM-M9701, Technische Universität München, 1997.

[8] E. Bannai and T. Ito. *Algebraic combinatorics I: Association Schemes*. Benjamin, 1984.

[9] O. Bastert. New ideas for canonically computing graph algebras. Technical Report TUM-M9803, Technische Universität München, 1998.

[10] O. Bastert. Computing equitable partitions. *Match*, 40:265–272, 1999.

[11] O. Bastert, D. Rockmore, P. Stadler, and G. Tinhofer. Landscapes on spaces of trees. in preparation, 2000.

[12] A. E. Brouwer and W. H. Haemers. Association schemes. In R. L. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of Combinatorics*, volume 2, pages 747–771. Elsevier (North-Holland), 1995.

[13] R. A. Brualdi. Some applications of doubly stochastic matrices. *Linear Algebra and Its Applications*, 107:77–100, 1988.

[14] J.-Y. Cai, M. Fürer, and N. Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

[15] A. Chan and C. D. Godsil. Symmetry and eigenvectors. In G. Hahn and G. Sabidussi, editors, *Graph Symmetry: Algebraic Methods and Applications*, volume 497 of *NATO ASI Series C*, pages 75–106. Kluwer, 1997.

[16] Collatz, Lothar and Sinogowitz, Ulrich. Spektren endlicher Grafen. *Abh. Math. Sem. Univ. Hamburg*, 21:63–77, 1957.

[17] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression. *Journal of Symbolic Computation*, 9:251–280, 1990.

[18] D. Cvetković, P. Rowlinson, and S. Simić. *Eigenspaces of Graphs*. Encyclopedia of Mathematics and Its Applications. Cambridge University Press, 1997.

[19] Derek G. Corneil and Calvin C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the Association of Computing Machinery*, 17:51–64, 1970.

[20] R. Diestel. *Graph Theory*. Springer, 1997.

[21] A. Ehrenfeucht. An application of games to the completeness problem for formalized theories. *Fundamenta Math.*, 49:129–141, 1961.

[22] S. Evdokimov, M. Karpinski, and I. Ponomarenko. Compact cellular algebras and permutation groups. Technical Report 85154-CS, Universität Bonn, 1996.

[23] S. Evdokimov, M. Karpinski, and I. Ponomarenko. On a new high dimensional Weisfeiler-Lehman algorithm. *Journal of Algebraic Combinatorics*, 10:29–45, 1999.

[24] S. Evdokimov and I. Ponomarenko. On highly closed cellular algebras and highly closed isomorphisms. *Electronic Journal of Combinatorics*, 6:#R18, 1999.

[25] S. Evdokimov and I. Ponomarenko. Separability number and schurity number of coherent configurations. *Electronic Journal of Combinatorics*, 7:#R31, 2000.

[26] S. Evdokimov, I. Ponomarenko, and G. Tinhofer. On a new class of weakly compact graphs. *Discrete Mathematics*, 225:149–172, 2000.

[27] J.-L. Faulon. Isomorphism, automorphism partitioning, and canonical labeling can be solved in polynomial-time for molecular graphs. *Journal of Chem. Inf. Comput. Sci.*, 38:432–444, 1998.

[28] R. Fraissé. Sur quelques classifications des systemes de relations. *Publ. Sci. Univ. Alger. Ser. A*, 1:35–182, 1955.

[29] S. Friedland. Coherent algebras and the graph isomorphism problem. *Discrete Applied Mathematics*, 25:73–98, 1989.

[30] M. Fürer. A counterexample in graph isomorphism testing. Technical report, Computer Science Department, Pennsylvania State University, 1987.

[31] M. Fürer. Private communication, 1998-2000.

[32] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, 1979.

[33] C. D. Godsil. *Algebraic Combinatorics*. Chapman & Hall, 1993.

[34] C. D. Godsil and B. D. McKay. Constructing cospectral graphs. *Aequationes Mathematicae*, 25:257–268, 1983.

[35] M. Grohe. Isomorphism testing for embeddable graphs through definability. In *32nd ACM Symposium on Theory of Computing*, 2000.

[36] F. Harary, C. King, A. Mowshowitz, and R. C. Read. Cospectral graphs and digraphs. *Bull. Lond. Math. Soc.*, 3:321–328, 1971.

[37] D. G. Higman. Coherent configurations. *I. Rend. Sem. Mat. Univ. Padova*, 44:1–25, 1972.

[38] D. G. Higman. Coherent algebras. *Linear Algebra and its Applications*, 93:209–239, 1987.

[39] J. Hinteregger and G. Tinhofer. Zerlegung der Knotenmengen von Graphen zum Nachweis der Isomorphie. *Computing*, 18:351–359, 1977.

[40] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*. Academic Press, New York, 1971.

[41] J. E. Hopcroft and R. Tarjan. Isomorphism of planar graphs. In R. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 131–152. Plenum Press, 1972.

[42] N. Immerman and E. Lander. Describing graphs: A first-order approach to graph canonization. In *Complexity Theory Retrospective*, pages 59–81. Springer-Verlag, 1990.

[43] M. Klin, C. Rücker, G. Rücker, and G. Tinhofer. Algebraic combinatorics in mathematical chemistry. Methods and algorithms. I. Permutation groups and coherent (cellular) algebras. *Match*, 40:7–138, 1999.

[44] M. Klin and G. Tinhofer. Algebraic combinatorics in mathematical chemistry. Methods and algorithms. III. graph invariants and stabilization methods. Technical Report TUM-M9902, Technische Universität München, 1999.

[45] T. Laidboeur, D. Cabrol-Bass, and O. Ivanciuc. Determination of topogeometrical equivalence classes of atoms. *Journal of Comp. Inf. Comput. Sci*, 37:87–91, 1997.

[46] X. Liu and D. J. Klein. The graph isomorphism problem. *Journal of Comput. Chem.*, 12:1243–1251, 1991.

[47] R. Mathon. Sample graphs for isomorphism testing. In *9th. Southeastern Conference on Combinatorics, Graph Theory and Computing*, 1978.

[48] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[49] B. D. McKay. Nauty users guide (version 1.5). Technical Report TR-CS-90-02, Computer Science Department, Australian National University, 1990.

[50] B. D. McKay. Private communication, 2000.

[51] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[52] H. L. Morgan. The generation of a unique machine description for chemical structures – a technique developed at chemical abstracts service. *J. Chem. Doc.*, 5:107–113, 1965.

[53] M. E. Muzychuk. Private communication, 2000.

[54] M. E. Muzychuk and G. Tinhofer. Recognizing circulant graphs: An application of association schemes. submitted.

[55] M. E. Muzychuk and G. Tinhofer. Recognizing circulant graphs of prime order in polynomial time. *Electronic Journal of Combinatorics*, 5(1):28, 1998.

[56] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[57] I. N. Ponomarenko. On computation complexity problems concerning relation algebras. *Zapiski Nauchnykh Seminarov POMI*, 202:116–134, 1993.

[58] I. N. Ponomarenko. *CC User's Guide*, Feb 1997.

[59] D. F. Robinson. Comparison of labeled trees with valency three. *Journal of Combinatorial Theory B*, 11:105–119, 1971.

[60] C. Rücker. Private communication, 1999.

[61] C. Rücker and G. Rücker. Computer perception of constitutional (topological) symmetry: TOPSYM, a fast algorithm for partitioning atoms and pairwise relations among atoms into equivalence classes. *Journal of Chemical Information and Computer Sciences*, 30:187–191, 1990.

[62] C. Rücker and G. Rücker. On using the adjacency matrix power method for perception of symmetries and for isomorphism testing of highly intricate graphs. *Journal of Chemical Information and Computer Sciences*, 31:123–126, 1991.

[63] H. Sachs. Über Teiler, Faktoren und charakteristische Polynome von Graphen I. *Wiss. Z.*, 12:7–12, 1966.

[64] H. Sachs. Über Teiler, Faktoren und charakteristische Polynome von Graphen II. *Wiss. Z.*, 13:405–412, 1967.

[65] U. Schöning. Graph isomorphism in the low hierarchy. *Journal of Computer and System Sciences*, 37:312–323, 1988.

[66] A. J. Schwenk. Computing the characteristic polynomial of a graph. In R. A. Bari and F. Harary, editors, *Graphs and Combinatorics*, pages 153–172. Springer, Berlin, 1974.

[67] P. F. Stadler. Landscape of correlation functions. *Journal of Mathematical Chemistry*, 20:1–45, 1996.

[68] P. F. Stadler and G. Tinhofer. Equitable partitions, coherent algebras and random walks: Applications to the correlation structure of landscapes. *Match*, 40:215–261, 1999.

[69] P. F. Stadler and G. P. Wagner. Algebraic theory of recombination spaces. *Evolutionary Computation*, 5(3):241–275, 1998.

[70] G. Tinhofer. Graph isomorphism and theorems of Birkhoff type. *Computing*, 36:285–300, 1986.

[71] G. Tinhofer. A note on compact graphs. *Discrete Applied Mathematics*, 30:253–264, 1991.

[72] B. J. Weisfeiler. *On Construction and Identification of Graphs.* Springer, Berlin, 1976.

[73] B. J. Weisfeiler and A. A. Leman. Reduction of a graph to a canonical form and an algebra arising during this reduction. *Naucho - Technicksagy Informatsia*, 9(2):12–16, 1968. Russian.

Index

1-dimensional stabilization, 16
1-dimensional structure constant, 11
1-dimensional structure list, 11
1-dimensional structure value, 11
1-stable coloring, 11
2-dimensional structure constant, 14
2-dimensional structure list, 13
2-dimensional structure value, 13
2-stable coloring, 11, 13

$A(G)$, 6
adjacency matrix, 6
adjacent, 5
$\mathcal{A}_{\Gamma_n}$, 69
algorithm, 22
association scheme, 13
automorphism, 8
automorphism group, 1, 8
automorphism partition, 1, 8
automorphism partition problem, 1

basis edge, 13
Birkhoff graphs, 93

canonical, 17
canonical label, 21
canonical labeling, 21
canonical way, 17
canonically labeling, 1
Cayley graph, 63
cell, 12
cellular algebra, 13, 14
cellular set, 12
coarser, 7
coherent, 14
coherent algebra, 2, 5, 11, 13, 14
coherent algebra corresponding to f, 15
coherent algebras, 11
coherent coloring, 11
coherent configuration, 15
color matrix, 9
color-class, 9
colored graph, 6
  isomorphic, 8
coloring, 6, 7
  k-stable, 11, 21
  1-stable, 11
  2-stable, 11
  coarser, 7
  coherent, 11
  complete, 7
  edge, 7
  equitable, 11
  equivalent, 7
  finer, 7
  vertex-, 7
compact, 92, 93
complete, 22
complete coloring, 7
complete sequence of partitions, 12
configuration graph, 65, 68
connected, 6
  strongly, 6
corresponding, 50
cospectral, 58
cover, 47
$\mathcal{CP}(L)$, 45
crossover
  $(s:t)$-, 68
$c_T(P)$, 73

$c_v(T)$, 73
cycle, 6

$\Delta(G)$, 12
d(iagonal)-crossover, 67
degree, 5
diameter, 6
diam(G), 6
directed, 5
discrete, 84
dist(u, v), 6
distinguished, 7
doubling, 90

E, 5
edge
  basis, 13
  non-basis, 13
  undirected, 5
edge set, 5
edge-coloring, 7
edges
  non-, 7
eigenvalue, 12
encoding length, 22
equal, 66
equally colored, 16
equitable coloring, 11
equitable partition, 2, 11
equitable partitions, 5, 11
equivalent, 7, 15, 96

f, 7
finer, 7
fitness function, 65
forest, 6
$f_v{}^{2\to1}$, 62

G, 5
genetic, 66
G_{f_v}, 61
graph, 5
  k-stable, 21
  automorphism, 8
  colored, 6
  connected, 6
  diameter, 6
  equivalent, 15
  induced, 6
  isomorphism, 8
  labeled, 8
  order of a, 5
  regular, 6
  size of a, 5
  strongly connected, 6
  undirected, 5
graph automorphism, 1
graph isomorphism, 1

head, 5
hit, 29, 37

$\mathbf{il}(k)$, 50
incident, 5, 15, 44, 71
indeg(v), 5
indegree, 5
induced, 6
inner edge, 66
inner tree, 66
inner vertex, 66
instance, 22
invariant procedures, 22
isomorphic, 8
isomorphism, 8

k-coloring, 5, 16
k-configuration, 52
k-dimensional Weisfeiler-Leman algorithm, 49
k-dimensional stabilization, 16
k-dimensional structure list of v^k, 16
k-dimensional structure list ofc^k, 16
k-dimensional structure value, 16
k-roof, 47
k-stable, 16, 21
  loosely, 47
  strongly, 50
k-stable coloring, 11

k-stable graph, 21
k-stable partition, 11
k-stable coloring, 21
k-star, 44
k-starlet, 15
k-tuple, 15

L^1, 11
L^2, 13
labeled graph, 8
labeling, 8
Laplacian, 13
larger, 74
LCOC rule, 26
leaf, 66
length, 6
$\mathcal{L}_{f^k}(G)$, 61
l-invariant, 22
L^k, 16
longer neighbor, 75
longest inner path, 71
loosely k-stable, 47

m, 5
matrix representation, 14
m-move $\mathcal{P}_k$ game, 52
molecular graph, 95

n, 5
nauty, 88
necessary, 26
neighborhood equivalent, 96
neighbors, 5
new, 26
new colors, 26, 28
non-basis edge, 13
non-edges, 7
non-switching, 61
norm, 84
norm-code, 85
$\mathcal{NP}$-complete, 23
$\mathcal{NP}$-hard, 23

old color, 26
orbit partition, 1
orbits, 1
order, 5
outdeg(v), 5
outdegree, 5

p(arallel)-crossover, 67
part, 50
    set of, 50
partition
    automorphism, 8
    complete sequence of, 12
    equitable, 11
path, 6
    length, 6
    length of a, 6
    simple, 6
path stable, 60
permutation, 44
$\pi(T)$, 69
$\mathcal{P}_k$ game, 52
pointing f at v, 61
polynomial, 23
polynomially time equivalent, 23
problem, 22
proper, 9, 43
pseudo coloring, 28
p_v^c, 11

qStab, 28
quotient graph, 12
qWeil, 28

rank, 7
recently recolored, 26
reduced dimension, 16
reduced structure list, 26
reduction code, 50
regular, 6
responsible, 74
Robinson graph, 68
running time, 23

Schur-Hadamard product, 14

separator, 54
set of parts, 50
simple, 6
singular, 84
size, 5, 22
Solving, 22
s-path, 75
spectrum, 58
stabilization
    k-dimensional, 16
    1-dimensional, 16
stabilization procedure, 1
standard base, 15
$(s : t)$-crossover, 68
strong automorphism, 93
strongly k-stable, 50
strongly connected, 6
strongly regular, 19
structure constant, 15
    1-dimensional, 11
    2-dimensional, 14
structure constants, 11
structure list
    1-dimensional, 11
    2-dimensional, 13
structure value, 11
    k-dimensional, 16
    1-dimensional, 11
    2-dimensional, 13
subgraph, 6
s-vertex, 67

tail, 5, 76
time complexity function, 23
$\mathcal{T}_n$, 66
$\mathcal{T}_n^{dm}(i)$, 70
$\mathcal{T}_n^{n_k}(i)$, 70
total degree partition, 12
totally path stable, 60
tree, 6
trees of Γ_n, 68
type, 68

undirected edge, 5
undirected graph, 5
useful, 101

V, 5
vertex
    degree, 5
    indegree, 5
    outdegree, 5
vertex set, 5
vertex-coloring, 7
vertex-transitive, 8

weakly compact, 93
weakly isomorphic, 22
Weisfeiler-Leman algorithm, 19
    k-dimensional, 49
wins, 52