SAPIENZA
UNIVERSITÀ DI ROMA

DIPARTIMENTO DI SCIENZE STATISTICHE

DOTTORATO IN RICERCA OPERATIVA - XXV CICLO

Solving hard instances of maximum stable set problem by equitable partitions

Author:
Gianmaria LEO

Supervisor:
Prof. Gianpaolo ORIOLO

March 2013

Ai miei Nonni,
ai miei genitori Giovanni e Marilena,
a Ubi.

"When a man is born... there are nets flung at it to hold it back from flight.
You talk to me of nationality, language, religion.
I shall try to fly by those nets."

James Joyce, *A Portrait of the Artist as a Young Man*

Acknowledgements

I owe my sincere gratitude to my Advisor Professor Gianpaolo Oriolo for his precious and stimulating guidance, assistance and encouragements. His consistent and enlightening instruction, deepened knowledge and advices have gone a long way to this thesis and have also augmented my love for research.

I would also like to thank Professor Fabrizio Rossi and Professor Stefano Smriglio for their help in writing of this thesis. Their interesting starting points about computational topic and their observations have contributed to the development of my work, also have increased my interest.

I am very grateful to Professor Andrea Pacifici and Professor Veronica Piccialli for their contribution to my scientific training.

Finally, I thank my parents Giovanni and Marilena, my girlfriend Sara and my colleague, and friend, Marco, for the great support they have shown me.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Motivation

The Maximum Stable Set Problem (MSSP) is one of the most widely studied problems in Operations Research, as it has important applications in many different scientific areas. It is well-known that the problem is NP-hard and it is as well extremely hard to approximate [Hås96]. While MSSP is solvable in polynomial time for certain classes of graphs (e.g. line graphs), it is in practice very challenging, since even instances with a few hundred of vertices could be hard to solve. Moreover, it requires quite advanced techniques, mainly from integer programming.

In practice, the most effective algorithms for solving challenging instances of MSSP are based on the *branch&bound* paradigm, which is a basic standard method for solving integer programs. The solution space of an integer program is represented as a search tree, whose node are all possible solutions of the problem. Then, finding an optimal solution require exploring the search tree. A general branch&bound method basically performs two subsequent operations: it partitions the set of feasible solutions to obtain a set of more easily-solved subproblems; then solutions are bounded in order to prune subproblems that cannot contain optimal solution. Essentially, the bounding procedure computes relaxation of subproblems, or it is based on combinatorial algorithms. In practice, bounds from relaxations of integer programs are often either too weak and reasonably tractable (this is often the case with linear programs) or very strong, but requiring an unacceptable computational effort (this is often the case with semidefinite programs). Combinatorial bounding procedures are often successful in practice, even though they provide weak bound.

In particular, MSSP instances with properties of *symmetry* or, more in general, integer programs with symmetric feasible regions, often turn out to be challenging. The motivation is that branch&bond wastes a lot of computation time generating subproblems which contain symmetric solutions.

Consider for instance the graph G in Fig. 1.1: the violet vertex belongs to a clique of size four and it is adjacent to one vertex for each of the triangles (therefore, $|V(G)| = 4 + 3k$, $|E(G)| = 6 + 4k$). We can immediately see that $\alpha(G) = 1 + k$. However, there many equivalent maximum stable sets, in particular we have 2^k equivalent maximum stable sets which intersect the violet vertex and 3^{k+1} maximum stable sets that do not intersect it. The existence of an exponential number of optimal solutions amplifies the grow of brach&bound enumeration tree, since many subproblems containing equivalent solution are eventually explored.

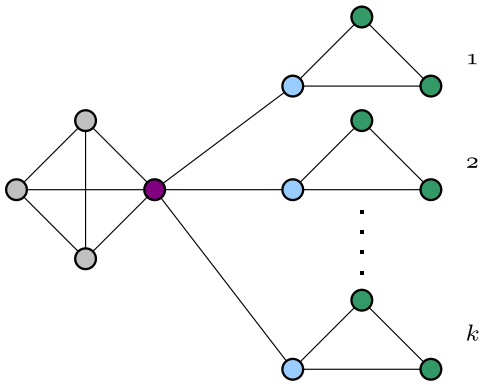


Figure 1.1: A graph with many symmetric maximum stable sets

One can observe that for the graph in Fig.1.1, there exist several mapping of the vertex set into itself that preserve adjacencies. This leads to the definition of the *orbits* of the graph, that are the class of vertices that are equivalent under such mappings (more formal definitions will be given later). If we look again at the graph in Fig.1.1, we can check that vertices within the same orbit have the same color. We point out that the orbits of a graph are uniquely defined, and they provide a *partition* of the vertices.

If we move to integer programs, we say that an integer program is *symmetric* if its variables can be permuted without changing the structure of the problem. Symmetric integer programs are popular in combinatorial optimization: they appear when classical problems are formulated, e.g. graph coloring, or they "naturally" arise from application. In the case of MSSP, several symmetric instances are reformulations of well-studied problems of coding theory, e.g. error correcting-code problems.

In literature, we find three classes of approaches dealing with symmetry in integer linear programming. The first two pursue an *off-line* strategy that removes symmetry in formulations of (ILP). The last one develops *on-line* techniques that exploit symmetry information during the branch&bound algorithm. Basic ideas are

discussed below.

The first category of symmetry-breaking methods attempts to remove symmetry by *reformulating the problem*. In [MDZ08], the authors propose a reformulation of the graph coloring problem which aims to remove some equivalent solutions obtained by color permutations. The authors investigate symmetry properties of coloring formulations and provide a method for generating valid inequalities that is implemented in a cutting plane algorithm. In general, reformulation techniques provide an effective way to avoid symmetry, but they show the disadvantage to be applicable only to certain classes of problem, with specific structure and property of symmetry.

A second class of approaches deal with general polyhedral properties which impact symmetry. In [KP08], the authors investigates *orbitopes*. Given a set of binary matrices $\mathcal{M}^{p\times q}$ and a symmetry group $\mathcal{G}$ acting on the columns of a matrix, a full orbitopes $O_{p\times q}(\mathcal{G})$ is a set of matrices that are lexicographically maximal within their orbits. The work focuses on partitioning and packing orbitopes, which correspond to feasible solutions respectively of set-partitioning problems and set-packing problems. The authors provide a full polyhedral description of partitioning and packing orbitopes and a polynomial separation algorithm for all inequalities. Oribitopes allow to avoid the symmetry in formulations of partitioning and packing problem simply requiring that feasible solutions have to be an element of the corresponding orbitope. This kind of approach efficiently succeeds in removing symmetry from integer linear formulations, but it has the disadvantage that many constraints could be added and the choice of a fundamental set of variables may conflict with branching strategies.

A last class of approaches exploit symmetry information during the branch&bound algorithm. In [Mar03], the author introduces *isomorphism pruning*, one of the most effective symmetry-breaking method: the basic idea follows. Given a subproblem P in the branch&bound enumeration tree, we can find a set $\mathcal{G}(P)$ of subproblems of the enumeration tree that are equivalent to a P by automorphisms of the feasible region of (ILP). Thus, solutions of subproblems in $\mathcal{G}(P)$ are symmetric to solutions of P: during the branch&bound, the method identifies all equivalent nodes to an incumbent subproblem P, then subproblems in $\mathcal{G}(P)$ are pruned by isomorphism. Essentially, the method has two cons: a significant computational effort is required either to compute automorphism groups of all current subproplems in the enumeration tree, either to compare them to each other; moreover, algebraic tools needed for the computation of automorphism group complicate the implementation.

A powerful symmetry-breaking method that exploits symmetry information during branch&bound is *orbital branching* [OLRS11a]. Orbital branching turn out to be one of the most effective method in practice [Ost09]. In this thesis we will exploit

orbital branching (see Section 4.2) for solving MSSP instances, and we therefore give a more detailed overview of this method in Section 3.1.

We can summarize the above discussion, by pointing out that the most part of state-of-art symmetry-breaking approaches exploits symmetry group information in order to avoid bad effects of symmetry during the optimization process. However, the complexity of computing generators of symmetry group and orbits, is not known to be polynomial. A consequence is that a significant computational effort could be required, even though there exist efficient algorithms that often perform efficiently in practice. *Furthermore, removing completely the symmetry implies that useful information gets lost.*

In the thesis, we aim at showing that, in some cases, symmetry produces regular structure that can be exploited by branch-and-bound algorithms. We explain our main idea by considering again the graph in Fig. 1.1. Recall that vertices within the same orbit have the same color. Interestingly, observe that the "grey" vertices are not adjacent to the "cyan" vertices, as well as the "purple" vertices are not adjacent to the "green" vertices. Such observations can be encoded in an auxiliary graphs that we depict in the Figure 1.2.

Figure 1.2: EP-graph associated to graph in Fig. 1.1

The rationale is the following: the vertices of this auxiliary graph correspond to the orbits, and two orbits are adjacent if there are edges joining some vertices of the two classes. It is somehow surprising that, as we show in the thesis, this auxiliary graph is simple for several hard instances of the stable set problem: in many cases it is even a tree!

However, there are two drawbacks. As we already pointed out, the complexity of computing the orbits of a graph is not known to be polynomial. Moreover, there is a *single* orbit partition of the graph, and so it might be easily the case that the auxiliary graph is not simple as above (even though this does not seem to be often the case with symmetric instances).

In order to deal with these problems, in the thesis, we consider some partitions of the vertices of a graph which arise from a relaxation of the orbit partition: *equitable partition*. An equitable partition of a graph is a partition of the vertices such that every two vertices belonging to a same class are adjacent to the same number of vertices of each cell. Loosely speaking, one can say that the orbit partition gives a description of the symmetry of a graph showing more informations than equitable partitions, since the vertices that share a same orbit are related by automorphisms.

In fact, while the orbit partition of a graph is an equitable partition, the converse does not hold.

Equitable partitions allow indeed to deal with the previous drawbacks. On the one hand the so-called *coarsest equitable partition*, that can be computed very efficiently. And in general, given any partition, we can compute in polynomial time its coarsest refinement. Therefore, we can easily compute *several* different partitions.

We may then associate to each equitable partition $\mathcal{P}$ of a graph an auxiliary graph, that we call the *EP-graph* associated with $\mathcal{P}$, along the same same lines as we did above for the orbit partition. As we show in the thesis, EP-graphs are a powerful tool, that allow to capture the structure of a symmetric graphs. Interestingly, in some cases. we detected equitable partitions that are not orbit partitions, but have a very simple EP-graph: that would not have been recognized dealing only with orbits.

We believe that EP-graphs introduce a new approaches to exploit symmetry information, for instance it can be used in combinatorial branch&bound schemes. In this thesis, we however focus on how to derive valid inequalities from the topology of EP-graph, in order to strengthen integer linear programming formulations of the maximum stable set problem.

The rational is the following. Suppose we are given an equitable partition $\mathcal{P}$ of a graph G. We consider subgraphs of G induced by some relevant structure of the EP-graph associated with $\mathcal{P}$, e.g. the subgraph induced by two classes V_i and V_j that are adjacent in EP-graph. Trivially, the rank inequality $\sum_{v \in V_i \cup V_j} x_v \leq \alpha(V_i \cup V_j)$ is valid for the stable set polytope of G and can be added to any integer linear programming formulation of the maximum stable set problem. The very good news is that, in many cases, $G[V_i \cup V_j]$ has a "small" size and a "simple" symmetry structure. For this reason, the computation of the right-hand side $\alpha(V_i \cup V_j)$ can be carried out efficiently. In particular it is often the case that, for this computation, orbital branching is effective, even if it is not effective for the *entire* graph. Moreover, each inequality derived from an equitable partition can be generated independently from other (e.g. because they involve different edges of the EP-graph), thus a remarkable advantage is the potentiality of performing the generation of equitable partition inequalities in parallel computing.

In the thesis, we then consider inequalities that are induced by vertices, edges, triangles and closed neighborhood of the EP graph associated with an equitable partition $\mathcal{P}$ of a graph G, that we call *equitable partition inequalities*. We add this inequalities to the standard clique relaxation of the stable set polytope, and this defines what we call the *equitable partition formulation*.

Given an equitable partition $\mathcal{P}$, we also present an *aggregate equitable partition formulation*. The idea is the following: we associate to each class $P \in \mathcal{P}$ an integer variable y_P, that is equal to the number of vertices that a stable set will pick in the class P. Then we consider edge or clique etc. constraints arising from the EP-graphs, e.g., if we refer again to the above example, we would write $y_{V_i} + y_{V_j} \leq \alpha(V_i \cup V_j)$.

In order to formalize our method, one major issue to be addressed is the generation of a families of equitable partitions, as well the corresponding inequalities. A first set of inequalities can be easily obtained from the coarsest equitable partition, since it can be computed in polynomial time. However, our computational experiments have proved that the potentiality of the method is restricted if we limit our choice uniquely to coarsest equitable partition, as they sometimes provide inequalities that either are weak, or require an unacceptable computational effort to be computed. Therefore we show how to break the coarsest equitable partition so as to produce new, finer, equitable partitions yielding more effective inequalities.

We have experienced that there is a trade-off between quality and fragmentation of equitable partitions. Usually, equitable partitions with few classes give rise to strong inequalities, while equitable partitions with many classes often lead to weak inequalities. Then, we have build up a set of empirical rules that attempt to generate, but limit, the fragmentation of any equitable partition into many classes.

Computational experience focuses on hard and symmetric instances of the maximum stable set problem, that we believe promising for our method. We have performed computational experiments on three classes of symmetric instances: 1zc, mann and keller. We have compared the performance of ILOG CPLEX 12.4 (with standard settings) solving the equitable partition formulation in the original space, with orbital branching, one of the most effective symmetry-breaking methods in practice, that we have suitably implemented using CPLEX callbacks.

The *1zc-instances* are a class of instances from the Sloane *Independent Set Challenge* [Slo00] known to be among the hardest instances for the maximum stable set problem. In fact, for most instances of this class the optimal solutions is still unknown and even estimating upper bounds for them turns out to be challenging. 1zc-graphs arise from an application of coding theory, in particular they are a maximum stable set reformulation of the problem of finding a largest binary asymmetric 1-error-correcting code. For the instance 1zc512, both CPLEX solving equitable partition formulation and orbital branching can achieve an optimal solution. In this case, we obtain a better performance than orbital branching. Unfortunately, both method are not able to solve 1zc1024 and 1zc2048. However, for these instances, aggregate equitable partition formulations have allowed us to improve best

upper bound known so far. Moreover, we have certified the optimality for 1zc1024, a graph of the class whose stability number has been unknown so far.

The second class of instances on which we have tested our method is given by *mann-graphs*. Mann-graphs belong to the Dimacs [Dim92] benchmark and have been representing a challenge for a long time. They correspond to stable set formulations of the Steiner Triple Problem, translated from the set covering formulations by Mannino code [MS95]. Consequently, all their optimal solutions are known, since the corresponding set covering instances have been solved. However, exact methods known in literature are able to solve only the graphs of the class that are strictly smaller than mann_a81. For all mann-graphs considered, the aggregate equitable partition formulation allows to obtain excellent upper bound: for mann_a9, mann_a27 and mann_a45 we reach an absolute gap of one unit from the optimal solution, while, for mann_a81, we obtain a relative gap of 0.64% from the optimal solution. For mann_a27 and mann_a45, CPLEX solving equitable partition formulations shows impressive results, outperforming orbital branching. Unfortunately, the equitable partition is not able to certify the optimality for mann_a81 within a time limit of one hour. However, the aggregate formulation associated to a "pretty" equitable partition allows us to prove the optimality for mann_a81.

The last class of instances of our experience refers to *Keller-graphs*. This class belongs to Dimacs benchmark set and arises from the Keller's cube-tiling conjectures. In 1990,the author of [CS90] stated that there is a counterexample for this conjecture if and only if the n-dimensional Keller-graph, properly defined, has a clique of size 2^n. All keller-graphs have been solved to optimality by exploiting their theoretical properties in suitable enumeration schemes. In our experiments, we focus on keller4 and keller5. For both graphs, we obtain excellent upper bounds: for keller4 the upper bound is tight and for keller5 we obtain an absolute gap of one unit from the optimal solution. In these cases, both CPLEX solving the equitable partition formulation and orbital branching achieve optimal solution. For this class of graph, we observe that the performance of our method drastically improve if we use orbital branching for generating inequalities.

1.2 Linear and Integer Linear Programming

In this section we briefly recall notations and basic notions of linear and integer linear programming. For an exhaustive exposition, the reader may refer to [Sch86].

A *halfspace* in $\mathbb{R}^n$ is a set of points of $\mathbb{R}^n$ which satisfies a linear inequality $ax \leq b_0$, where $a \in \mathbb{R}^n, b_0 \in \mathbb{R}$. A *polyhedron* P is the intersection of finitely many halfspaces: $P = \{x \in \mathbb{R}^n : Ax \leq b\}$, where $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^n$; moreover, a bounded polyhedron is called *polytope*. An inequality $a'x \leq b'_0$ is *valid* for a polyhedron P

if $\{x \in \mathbb{R}^n : a'x \leq b'_0\} \supseteq P$. Given a finite set of points $X = \{x_1, \ldots, x_p\}$ and non-negative coefficients $\lambda_1, \ldots, \lambda_p$ such that $\sum_{i=1}^{p} \lambda_i = 1$, point $y = \sum_{i=1}^{p} \lambda_i x_i$ is a *convex combination* of points in X. The *convex hull* of X, denoted by $\text{conv}(X)$, is the set of all convex combinations of points in X. An alternative definition characterizes a polytope as the convex hull of finite set of points.

A *linear program* (LP) is the problem of maximizing (or minimizing) a linear function over a polyhedron:

$$(LP) := \max_{x \in \mathbb{R}^n} \{cx : Ax \leq b\}$$

where $c \in \mathbb{R}^n$. The function cx is called *objective function* and the inequalities of system $Ax \leq b$ are known as *constraints*. The polyhedron $\{x \in \mathbb{R}^n : Ax \leq b\}$ is called *feasible region* of (LP) and a point is *feasible* if it is in the feasible region. If the feasible region is an empty set, (LP) is *infeasible*. If the objective function value can be made arbitrarily large, we say that (LP) is *unbounded*, otherwise it is *bounded*.

The *simplex method* is one of the most effective algorithms for solving linear programs. Whereas the simplex method has exponential time complexity, it is well-known that linear programs can be solved in polynomial time by the *ellipsoid method* [Kha79]. However, the ellipsoid method does not allow any practical implementation, but it gave rise to the development of *interior points methods* which have been successful in applications. Implementations of simplex and interior points method show good efficiency in practice, then linear programming solvers are based on both these methods.

An *integer linear program* (ILP) is the problem of maximizing a linear function subject to a set of linear constraints with the restriction that values of all variables should be integral:

$$(ILP) := \max_{x \in \mathbb{Z}^n} \{cx : Ax \leq b\}$$

Let us observe that removing the integrality constraints of (ILP), we obtain a linear program which is called *linear relaxation* of (ILP). Whereas the linear relaxation of (ILP) can be computed in polynomial time, integer linear programming problems are well-known to be $\mathcal{NP}$-hard. Most popular methods for solving (ILP) are based on *Branch&bound* algorithm. Basically, the branch&bound first partitions (*branching*) the feasible region of (ILP) by fixing the value of one or more variables, in order to subdivide the (ILP) into smaller subproblems: essentially, it build up an enumeration tree whose nodes are subproblems of (ILP). Then, it estimates an upper bound (*bounding*) on the optimal value of each generated subproblem, in order to prune nodes of the enumeration tree that cannot contain any optimal solution of

(ILP). The general branch&bound paradigm performs the bounding of subproblems by computing the optimal solution of relaxation of each subproblem, that can be a linear or a semidefinite relaxation. A branch&bound which implements the bounding procedure by combinatorial heuristic algorithms is called *combinatorial* branch&bound. In practice, it usually happens that relaxations of subproblems are either too weak at reasonable computational effort, either very strong but unacceptably time-consuming. On the other hand, combinatorial heuristics are very fast, but they provide weak bounds. However, combinatorial branch&bound are more successful in applications.

1.3 Graphs: basic notions

An *undirected graph* is an ordered pair $G := (V, E)$, where V is a set of *vertices* and E is a set of unordered pairs of vertices each of which is called *edge*. Alternatively, we denote respectively by $V(G)$ and $E(G)$ the vertex set and the edge set of G. With a slight abuse of notation, we denote by (u, v) or uv the edge corresponding to the unordered pair $\{u, v\}$. If $(u, v) \in E$, we say that u, v are the *extremes* of edge (u, v), and that u, v are *adjacent* or *connected by an edge* in G. For each vertex $u \in V$, the *neighborhood* of u, denoted by $N(u)$, is the subset of V which contains the vertices adjacent to u, i.e $N(u) := \{vinV : (u, v) \in E\}$. The *degree* of a vertex $u \in V$, denoted by $d(u)$ is the number of vertices connected to u by an edge, i.e. $|N(u)|$. Given $U \subset V$, let $N(U) = \bigcup_{v \in U} N(u)$. For a vertex v, we let $N[v]$ denote the *closed neighborhood* of v, that is $N[v] = \{v\} \cup N(v)$. Analogously, we let $N[Q] = Q \cup N(Q)$. A pair $\{u, v\}$ occurring more than once in E is called a *multiple edge*. A *loop* is an edge of G which connect a vertex $u \in V$ to itself. A *simple graph* is an undirected graph without multiple edges or loops. For the sake of convenience, we use the term *graph* to refer to simple graph through the thesis.

A *subgraph* of a graph $G(V, E)$ is a graph $G'(V', E')$ of G with $E' \subseteq E$ and $V' \subseteq V$ and $u, v \in V'$ for each $(u, v) \in E'$. G' is an *induced subgraph* of G if $(u, v) \in E$ if and only if $u, v \in V$ and $(u, v) \in E$. Thus an induced subgraph is uniquely identified by a set $V' \subseteq V$, and we denote it by $G[V']$. Given a graph G and an integer $k \in \mathbb{N}$, an ordered set of vertices $v_1, \ldots, v_k$ is a *walk* (of length $k-1$) if $(v_i, v_{i+1}) \in E$ for each $i \in [k-1]$. If $v1, \ldots, v_k$ are all distinct, the walk is called a path. If $P = v_1, \ldots, v_k$ is a path, the vertex v_1 is called the *starting vertex* or *first vertex* of P and the vertex v_k the *end vertex* or *last vertex* of P. Sometimes, both v_1 and v_k are called the *end vertices* or *extremes* or *ends* of P. A graph is *connected* if there exists a walk between any two vertices of G.

The *complement* of a graph G is the graph $\bar{G}(V, \bar{E})$ where $\bar{E}$ is the set of edges (u, v) with $u \neq v$ such that $(u, v) \notin E$. A graph G is *complete* (or a *complete graph*)

if its complement has no edge. Given a graph $G(V, E)$, we say that a set $U \subseteq V$ is a stable set of G if no two elements of its are joined by an edge in G, whereas it is a *clique* if each two elements of it are joined by an edge in G. We denote by $\alpha(G)$ the size of the *maximum stable set* in G, and we often refer to $\alpha(G)$ as to the *stability number* of G. We denote by $\omega(G)$ the size of the *maximum clique* in G, and we often refer to $\omega(G)$ as to the *clique number* of G. A *coloring* is a function $f : V \to C$ that assigns to each vertex $u \in V$ a color $c \in C$ such that two adjacent vertices of G are not given the same color. The *chromatic* number of G is the cardinality of the smallest set C of colors such that there exists a coloring of G that uses only colors from C.

A graph $G(V, E)$ is k-partite if V can be partitioned in k sets $V_1, \ldots, V_k$ and each edge of G has an endpoint in V_i and one in V_j with $i \neq j$ (i.e. V_i is a stable set for every $i = 1, \ldots, k$). In the special case $k = 2$, the graph is said to be *bipartite*.

Given a graph $G = (V, E)$, a *stable set* S of G is a subset of V such that the vertices of S are pairwise non-adjacent. In the literature, all terms: *independent set*, *vertex packing*, *co-clique* or *anticlique* refer to stable set.

The *maximum stable set problem* (MSSP) is the problem of finding a stable set of maximum cardinality in G. The *stability number* of G, denoted by $\alpha(G)$, is the cardinality of the maximum stable set in G. Given a weighted graph $G = (V, E, w)$ where function $w : V \to \mathbb{R}$ assigns a weight w_i to each vertex $u_i \in V$, the *maximum weighted stable set problem* looks for a stable set S which maximises $w(S)$. Clearly, MSSP corresponds to the maximum weighted stable set problem with $w_i = 1$ for all $u_i \in V$. In this thesis, we focus on MSSP, also simply called "stable set problem", omitting the weighted case.

It is well-known that MSSP is $\mathcal{NP}$-hard and, as well, extremely hard to approximate [Hås96].

1.4 The Maximum Stable Set Problem (MSSP)

This section discusses polyhedral descriptions of MSSP. Given a graph $G = (V, E)$ and a stable set S of G, we define *incidence vector* of S, denoted by χ^S, the binary vector of size $|V|$ whose i-th entry has value 1 if and only if vertex $i \in V$ belongs to S. An implicit polyhedral description of stable sets of G is given by:

$$STAB(G) := \text{conv}\left\{\chi^S : S \subseteq V \text{ is a stable set of } G\right\} \tag{1.1}$$

$STAB(G)$ is called *stable set polytope*. Explicit descriptions of $STAB(G)$ are provided by integer linear formulation of the problem.

For each node $i \in V$, we introduce a binary variable $x_i \in \{0, 1\}$ that has value

1 if vertex $i \in S$, 0 otherwise. Since every two vertices $i, j \in S$ are not joined by an edge of V, the following inequalities, called *edge-inequalities* are valid for $STAB(G)$:

$$x_i + x_j \leq 1 \qquad \forall \{i, j\} \in E \tag{1.2}$$

We can easily observe that each $x \in \{0,1\}^{|V|}$ that satisfies edge-inequalities is an incidence vector of some stable set S. Hence, edge-inequalities give an explicit representation of $STAB(G)$:

$$STAB(G) = \left\{ x \in \{0,1\}^{|V|} : \, x_i + x_j \leq 1 \, \forall \{i, j\} \in E \right\} \tag{1.3}$$

This formulation of $STAB(G)$ has $O(|V|)$ variables and $O(|E|)$ constraints, so it is compact. Now, let focus on the following linear relaxation of the previous integer program, called *edge polytope* of the stable set:

$$FRAC(G) := \left\{ x \in \mathbb{R}^{|V|} : \, x_i + x_j \leq 1 \, \forall \{i, j\} \in E, \, 0 \leq x_i \leq 1 \, \forall i \in V \right\} \tag{1.4}$$

In general, $STAB(G) \subseteq FRAC(G)$ holds: given a complete graph K with $|V(K)| \geq 3$, the vector whose components are all equal to $1/2$ is feasible for $FRAC(K)$, but not for $STAB(K)$. On the other hand, if G is a connected bipartite graph, we get $STAB(G) = FRAC(G)$ [GLS88]. This result has the immediate consequence that a maximum stable set of a bipartite graph can be computed in polynomial time.

In view of previous consideration, we are interested in finding other inequalities which strengthen $FRAC(G)$ when G is not a bipartite graph. Let $C = (V', E')$ be an odd cycle of G, i.e. a connected subgraph of G with an odd number of vertices, such that each vertex is adjacent to exactly two other vertices. Let us observe that each variable x_i with $i \in V'$ appears in exactly two edge inequalities of $STAB(C)$. By summing all these inequalities and dividing both sides by 2, we obtain $\sum_{i \in V'} x_i \leq |V'|/2$. As all variables are integer, the previous inequality is still valid if we round down its right-hand-side. Thus, the following inequalities, called *odd-cycle inequalities*, are valid for $STAB(G)$:

$$\sum_{i \in V'} x_i \leq \frac{|V'| - 1}{2} \qquad \forall \text{ odd cycle } C = (V', E') \subseteq G \tag{1.5}$$

The suitable polytope, called *odd-cycle stable set polytope*, is a relaxation of $STAB(G)$, stronger than $FRAC(G)$:

$$CSTAB(G) := \left\{ x \in FRAC(G) : \, \sum_{i \in V'} x_i \leq \frac{|V'| - 1}{2} \; \forall \text{ odd cycle } C \subseteq G \right\} \tag{1.6}$$

We can easily check that $CSTAB(G) \subseteq STAB(G)$ holds in general. Although the number of odd-cycles of a graph grows exponentially in the number of vertices, the separation problem for odd-cycle inequalities is solvable in polynomial time [GLS81]: equivalently, a linear function over $CSTAB(G)$ can be optimized in polynomial time. A graph G such that $CSTAB(G) = STAB(G)$ is called *t-perfect*, so MSSP in t-perfect graphs can be solved in polynomial time.

Given a graph G, a clique Q is a subgraph of G whose vertices are pairwise adjacent, i.e. Q is a complete subgraph of G. Since a stable set S intersects at most one vertex of Q, the following inequalities, called *clique inequalities*, are valid for $STAB(G)$:

$$\sum_{i \in Q} x_i \leq 1 \qquad \forall \text{ clique } Q \subseteq G \tag{1.7}$$

The following polytope, called *clique stable set polytope*, is a relaxation of $STAB(G)$, stronger than $FRAC(G)$:

$$QSTAB(G) := \left\{ x \in FRAC(G) : \sum_{i \in Q} x_i \leq 1 \, \forall \text{ clique } Q \subseteq G \right\} \tag{1.8}$$

In contrast to odd-cycle inequalities, separating clique inequalities of G is $\mathcal{NP}$-hard, since it is equivalent to finding a maximum clique in G, i.e. a maximum stable set in the complement of G. Hence, it is $\mathcal{NP}$-hard optimizing over $QSTAB(G)$. However, a remarkable fact follows. A graph G is called *perfect* if $QSTAB(G) = STAB(G)$ and the MSSP in perfect graphs can be solved in polynomial time [GLS88]: the result is based on the *Lovász ϑ-number*, a semidefinite relaxation of $STAB(G)$.

Finally, let us consider the following class of inequalities, called *rank inequalities*:

$$\sum_{i \in U} x_i \leq \alpha(G[U]) \qquad U \subset V \tag{1.9}$$

Rank inequalities are obviously valid for $STAB(G)$, in particular they are a generalization of all other inequalities presented above. In general, generating strong rank inequalities is hard.

In practice, computational experiments show that clique inequalities are significant for integer linear programming approaches to MSSP [RS01]. Often, it is successful considering more general classes of inequalities, e.g. rank inequalities, which also contain clique inequalities.

1.5 A few more notations

Throughout the thesis we denote $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ respectively the set of natural, integer, rational and real numbers. By $\mathbb{Z}_+$ and $\mathbb{R}_+$ we denote respectively the set of non-negative integer and non-negative real numbers. Given $n \in \mathbb{N}$, we denote by $[n]$ the finite set $\{1, 2, \ldots, n\}$. Let $f, g : \mathbb{N} \to \mathbb{R}$ be functions from the set of natural numbers to set of real numbers. We say that $f = O(g)$ if there exist constants C and n_0 such that $f(n) \leq Cg(n)$ for all integers $n \geq n_0$. Given a set S and a function $x : S \to \mathbb{R}$, for each $\bar{S} \subseteq S$ we define $x(\bar{S}) := \sum_{s \in \bar{S}} x(s)$.

Chapter 2

Symmetry in Graphs

In this chapter we deal with several tools that allow to grasp symmetries in graphs. We first revise the definitions of isomorphism and automorphism of graphs, leading to the crucial notions of orbit and orbit partitions of graphs. We then move to the definition of equitable partition of graphs, show that this is a relaxation of the orbit partition, and introduce the crucial definition of coarsest equitable partitions. We deal with algorithmic issues and discuss the complexity of finding the orbit partition of a graph, and the complexity of finding the coarsest equitable partition. Eventually we present our first original contribution, the EP-graph associated with some equitable partition.

2.1 Isomorphism and Automorphism of graphs

The problem of deciding whether two graphs have the same structure is one of the most studied problems in Graph Theory, since it has several applications in different scientific fields as well as it is fascinating from a theoretic point of view. Another problem, related to the previous one, consists of finding the vertices of a given graph that are *symmetric* or indistinguishable under the entire structure of the graph. These informal notions, which concern the idea of symmetry in graphs, are formalized with the concepts of *isomorphism* and *automorphism* of graphs.

Definition 2.1.1. *An* isomorphism *between graphs G and H is a bijection $\phi : V(G) \to V(H)$ such that $uv \in E(G)$ if and only if $\phi(u)\phi(v) \in E(H)$.*

Substantially, an isomorphism is a mapping between $V(G)$ and $V(H)$ which preserves the adjacencies of the corresponding graphs. If there exists an isomorphism ϕ between G and H, it is denoted by $G \cong H$ and we can say that that G and H are isomorphic. This problem is theoretically fascinating since it is unknown either to be solvable in polynomial time or to be $\mathcal{NP}$-complete. However, it is solvable

in polynomial time for certain classes of graphs, like trees, planar graphs, interval graphs, permutation graphs, partial k-trees, and, also for graphs with certain bounded parameters (degree, eigenvalue multiplicity and genus).

Now, let us focus on the properties of symmetry of a given graph G.

Definition 2.1.2. *An automorphism of a graph G is a permutation $\pi : V \to V$ of vertices such that $uv \in E$ if and only if $\pi(u)\pi(v) \in E$.*

An automorphism is essentially a mapping of a graph onto itself which preserves the adjacencies of the vertices. Clearly, every graph admits a trivial automorphism given by the identity permutation. Nevertheless, it is well-known that the problem of finding a non-trivial automorphism is at least as difficult as the problem of finding an isomorphism between graphs.

2.2 The orbit partition of a graph

In order to formalize the definition of orbit partition and how orbits are related to automorphisms, some definitions of Group Theory are recalled.

Definition 2.2.1. *A non-empty, finite, set of elements $\mathcal{G}$ together with a binary operation $\circ$ is called* group *if the following axioms hold:*

- *for all $a, b \in \mathcal{G}$, $a \circ b \in G$ (*closure*);*
- *for all $a, b, c \in \mathcal{G}$, $a \circ (b \circ c) = (a \circ b) \circ c$ (*associativity*);*
- *there exists $e \in \mathcal{G}$ such that $a \circ e = e \circ a = a$ for every $a \in \mathcal{G}$ (*identity element*);*
- *for every $a \in \mathcal{G}$, there exists $a^{-1} \in \mathcal{G}$ such that $a \circ a^{-1} = e$ (*inverse element*).*

Given a set of indices $I_n = \{1, \ldots, n\}$, the set of all permutations of elements of I_n together with the binary operation of *composition* of permutations form a group $\mathcal{S}_n$, called *symmetric group*. The term *permutation group* usually refers to a subgroup of $\mathcal{S}_n$, i.e. a subset of $\mathcal{S}_n$ which satisfies the group axioms.

Definition 2.2.2. *An* action *of a group $\mathcal{G}$ on a set X is a function $\mathcal{G} \times X \to X$, mapping (g, x) in $g(x)$, which satisfies the following conditions:*

- *for all $x \in X$ and $g, h \in \mathcal{G}$, $g \circ h(x) = g(h(x))$ (*associativity*);*
- *if e is the identity of $\mathcal{G}$, then $e(x) = x$ for all $x \in X$ (*identity*).*

Then, the definition of orbit follows:

Definition 2.2.3. *Let $\mathcal{G}$ a group acting on a set X. The* orbit *of $x \in X$ is the subset of elements of X onto which x can be mapped by the elements of $\mathcal{G}$:*

$$\mathrm{orb}(x, \mathcal{G}) := \left\{x' \in X : \exists g \in \mathcal{G} \text{ such that } x' = g(x)\right\} \tag{2.1}$$

From the previous definition, we can observe that two different elements $x, y \in X$ share a same orbit, denoted by $x \sim y$, if and only if there exists $g \in \mathcal{G}$ with $g(x) = y$. The properties of a group guarantee that the binary relation $\sim$ is an equivalence relation, thus the action of $\mathcal{G}$ on X partitions X in equivalence classes which correspond to the orbits of X. Finally, the orbit partition of X is:

$$\mathcal{O}(\mathcal{G}) := \bigcup_{x \in X} \mathrm{orb}(x, \mathcal{G}) \tag{2.2}$$

Now, let $\mathrm{Aut}(G)$ be the set of all automorphisms of a graph G. It easy to check that $\mathrm{Aut}(G)$ is permutation group under the operation of composition of two automorphisms, which corresponds to the composition of the corresponding permutations of the vertices. Thus, the action of $\mathrm{Aut}(G)$ on the vertex set $V(G)$ gives rise to the orbit partition $\mathcal{O} = \{O_1, \ldots, O_t\}$ of G.

2.3 Equitable partitions of a graph

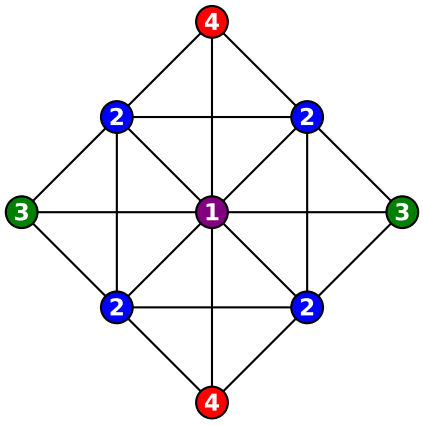
In this section we report definitions and results about equitable partitions of graph. Let U be a subset of V and let $d(v, U)$ denote the degree of v in U, that is $d(v, U) = |N(v) \cap U|$.

Definition 2.3.1. *A partition $\mathcal{P} = \{V_1, \ldots, V_p\}$ of the vertices of G is equitable provided that $d(x, V_j) = d(y, V_j)$ for all $x, y \in V_i$ and all $i, j \in \{1, \ldots, p\}$.*

Equitable partitions decompose a graph in more manageable pieces which are characterized by regular adjacencies. In particular, an equitable partition $\mathcal{P} = \{V_1, \ldots, V_p\}$ is such that:

- for all $i = 1, \ldots, p$, induced subgraphs $G[V_i]$ are regular;
- for all $i, j \in \{1, \ldots, p\}$ $(i \neq j)$, bipartite graphs $G[V_i, V_j] = G[V_i \cup V_j] \setminus (E(G[V_i]) \cup E(G[V_j]))$ are bi-regular, i.e. all vertices belonging to a same class of the bipartition have the same degree.

Let us observe that every graph has an equitable partition, called *trivial*, whose cells contain exactly one vertex. Moreover, it is easy to check that all regular graphs admit an equitable partition formed by a unique cell.



$$c = \begin{pmatrix} 1 \\ 4 \\ 2 \\ 2 \end{pmatrix}$$

$$D = \begin{pmatrix} 0 & 4 & 2 & 2 \\ 1 & 2 & 1 & 1 \\ 1 & 2 & 0 & 0 \\ 1 & 2 & 0 & 0 \end{pmatrix}$$

Figure 2.1: Example of equitable partition of a graph

Each equitable partition $\mathcal{P} = \{V_1, \ldots, V_p\}$ can be described by a pair of parameters (c, D), where c is a vector of size p such that $c_i = |V_i|$ for all $i \in \{1, \ldots, p\}$ and D is a matrix of size $p \times p$ such that $D_{ij} = d(u, V_j)$ with $u \in V_i$ for all $i, j \in \{1, \ldots, p\}$. The coloring of the graph showed in Figure 2.3 is an equitable partition, described by parameters (c, D).

As we will discuss in the following, there is one crucial equitable partition, since it can be computed in polynomial time: the coarsest equitable partition. We devote the rest of this section to its definition, which requires some other definitions and results from the literature.

Definition 2.3.2. *Given two partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ of a set V, we say that $\mathcal{P}_1$ is* finer *then $\mathcal{P}_2$ (or $\mathcal{P}_1$ is* coarser *than $\mathcal{P}_2$), denoted $\mathcal{P}_1 \preceq \mathcal{P}_2$, if every cell of $\mathcal{P}_1$ is a subset of some cell of $\mathcal{P}_2$.*

The relation $\preceq$ introduces the concept of *join* of partition.

Definition 2.3.3. *A join of partitions $\mathcal{P}_1$ and $\mathcal{P}_2$, denoted $\mathcal{P}_1 \vee \mathcal{P}_2$, is the finest partition coarser then $\mathcal{P}_1$ and $\mathcal{P}_2$.*

The set of all partitions of C with the relation $\preceq$ form a partially ordered set whose extrema are the partition with a unique cell, as maximal element, and the trivial partition, as minimal element. This property of partitions can be extended to equitable partitions.

Let $\xi(G)$ the set of all equitable partitions of a graph G. As we have already observed, $\xi(G)$ is nonempty since every graph admits a trivial equitable partition. One can easily observe that the joining operation preserves the property of equitable partitions:

Lemma 2.3.4 ([SU97])**.** *Let $\mathcal{P}$ and $\mathcal{Q}$ equitable partition of a graph G. Then $\mathcal{P} \vee \mathcal{Q}$ is equitable.*

Proof. Let $\mathcal{J} = \mathcal{P} \vee \mathcal{Q} = \{J_1, \ldots, J_p\}$. Let us note that each J_i can be partitioned into sets from $\mathcal{P}$ or into sets from $\mathcal{Q}$. Given $u, v \in J_i$, we must show that $d(u, J_j) = d(v, J_j)$ for all $j = 1, \ldots, p$. Since $\mathcal{J}$ is coarser than $\mathcal{P}$ and $\mathcal{Q}$, we have that u, v belong to a same cell of $\mathcal{P}$, or they belong to a same cell of $\mathcal{Q}$. W.l.o.g., let us suppose $u, v \in P_k$, where $P_k \in \mathcal{P}$. Hence, we get J_j partitioned into sets from $\mathcal{P}$, so let $J_j = P_1 \cup \ldots, \cup P_h$ with $P_1, \ldots P_h \in \mathcal{P}$. Thus, we have

$$d(u, J_j) = d(u, P_1) + \cdots + d(u, P_h)$$
$$d(v, J_j) = d(v, P_1) + \cdots + d(v, P_h)$$

Since $u, v \in \mathcal{P}_k$, $d(u, P_t) = d(v, P_t)$ holds for all $P_t \in \mathcal{P}$. It follows that $d(u, J_j) = d(v, J_j)$ for all $j = 1, \ldots, p$. □

Thus, the previous lemma implies that:

Theorem 2.3.5 ([SU97])**.** *Every graph has a unique coarsest equitable partition.*

Proof. We can show the theorem by contradiction. Since $\xi(G)$ is a non-empty set, let $\mathcal{P}$ and $\mathcal{Q}$ be two coarsest equitable partitions of G. Then, the partition $\mathcal{P} \vee \mathcal{Q}$ is coarser than both partitions $\mathcal{P}$ and $\mathcal{Q}$, a contradiction. □

We close this section by recalling a very interesting result from [RSU94]. There the authors investigate the interrelation between equitable partitions and the fractional isomorphism of graphs. Their main result is the following:

Theorem 2.3.6 ([RSU94])**.** *Let G and H be graphs. The following are equivalent:*

1. *$G \cong_f H$;*
2. *G and H have a common coarsest equitable partition;*
3. *G and H have some common equitable partition.*

In particular they show that each feasible matrix S of $(FRAC)$ is in bijection with an equitable partition common to G and H, i.e. S corresponds to an equitable partition of G and an equitable partition of H which are described by the same parameters (n, D).

2.4 Orbit partitions vs equitable partitions

In this section, discuss the analogies and the differences between orbit partitions and equitable partitions. Loosely speaking, one can say that the orbit partition gives a description of the symmetry of a graph showing more informations than equitable

partitions, since the vertices that share a same orbit are related by automorphisms. In fact, while the orbit partition of a graph is an equitable partition, the converse does not hold. We first show that:

Proposition 2.4.1. *The orbit partition of a graph is an equitable partition.*

Proof. Let $\mathcal{O} = \{O_1, \ldots, O_k\}$ the orbit partition of a graph G. Since there exists an automorphism $\pi \in \mathrm{Aut}(G)$ such that $\pi(O_j) = O_j$ for all $j = 1, \ldots, k$, $d(u, O_i) = d(v, O_i)$ holds for all $u, v \in O_j$ $i \in \{1, \ldots, k\}$. □

In Fig. 2.2, the coloring of the graph represents the orbit partition. We can easily check that the orbit partition is also equitable, since each class (corresponding to a color) has exactly the same number of neighbors in every class of the graph. However, in general, an equitable partition is not orbit partition: in Fig. 2.3, the coloring of the graph represents an equitable partition, namely the coarsest equitable partition. Let us observe that vertex a belongs to a cycle, but vertex d does not, hence a and b cannot share the same orbit, i.e. an automorphism which map a onto d does not exist.

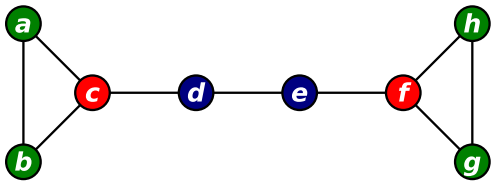


Figure 2.2: The orbit partition is equitable

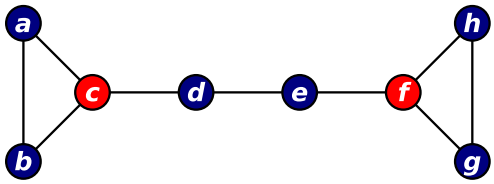


Figure 2.3: Equitable partition is not an orbit partition

From a computational point of view, the complexity of finding the orbit partition of graph is open, and it is one of the fundamental open questions in complexity theory. To the contrary, the computation of the *coarsest* equitable partition of a graph can be carried out in polynomial time (see the following section for the the definition of coarsest partition). In the following, we first shortly recap a combinatorial algorithm that was proposed by McKay to compute the orbit partition of

a graph, and then discuss in detail how to find the coarsest equitable partition in polynomial time.

In literature we can find several algorithms which are related to the computation of the orbit partition $\mathcal{O}(G)$ of a graph G. We can distinguish two approaches: one of them exploits algebraic tools and the other develops combinatorial algorithms. In the thesis, we focus on a combinatorial algorithm proposed by McKay, that computes the orbits of a graph through the computation of all automorphism generators of the graph. This algorithm is one of the powerful and best-known algorithms in literature: it has exponential time complexity, but performs exceptionally well in most circumstances. The McKay algorithm is implemented in the software `Nauty` [Mck90], which have been used in computational experiments described in Chapter 4. In the following, the main idea of the algorithm is reported; for a complete description we refer to [McK81] and [HR09].

McKay's algorithm computes the automorphisms of G, thus the orbits of G, by using degree informations of vertices and building a search tree with the property that the leaves return isomorphisms of G. In particular, the nodes of the search tree are equitable partitions: the root is the coarsest equitable partition $\mathcal{P}^*(G)$ and the leaves are trivial partitions. Given a node $\mathcal{P}(G) = \{V_1, \dots, V_p\}$ and a subscript $i \in \{1, ..., p\}$, let $\mathcal{A}_j$ be the partition finer than $\mathcal{P}(G)$ obtained by choosing vertex $u_j \in V_i$ and splitting V_i in two cells such that one of them contains only u_j, i.e. $\mathcal{A}_j := \{V_1, \dots, V_{i-1}, \{u_j\}, V_i \setminus \{u_j\}, V_{i+1}, \dots, V_p\}$. Every non-leaf node $\mathcal{P}(G)$ has $|V_i|$ children which are the coarsest equitable refinements $\{R(\mathcal{A}_j)\}_{u_j \in V_i}$ for a fixed cell $V_i \in \mathcal{P}(G)$. The algorithm starts computing the coarsest equitable partition $\mathcal{P}^*(G)$ and initializes the search tree. The main procedure generates recursively the nodes of the search tree until all relevant automorphisms are found. The exploration of the tree follows a depth-first strategy according to a back-tracking scheme and some criteria of pruning which allow to detect implied automorphisms.

2.4.1 Computing the coarsest equitable partition in polynomial time

Let us observe that the set of all equitable partitions $\xi(G)$ with the binary relation "finer than" $\preceq$ give rise to a partially ordered set: the maximal element of $\xi(G)$ is the coarsest equitable partition $\mathcal{P}^*(G)$; the minima element of $\xi(G)$ is the trivial partition; all other equitable partitions are finer than the coarsest and coarser than the trivial.

In the following, we will see that finding the coarsest equitable partition finer than a given partition can be done in polynomial time.

Definition 2.4.2. *Given a partition $\mathcal{A}$ of $V(G)$, the coarsest equitable refinement*

of $\mathcal{A}$, denoted $R(\mathcal{A})$, is the coarsest equitable partition finer the $\mathcal{A}$.

With similar arguments to Theorem 2.3.5, one may show that the coarsest equitable refinement $R(\mathcal{A})$ of a given partition $\mathcal{A}$ is unique. Now, we report an algorithm that exhibits how to compute such unique equitable partition $R(\mathcal{A})$ from $\mathcal{A}$.

Given a graph G and a partition $\mathcal{A} = \{V_1, \ldots, V_s\}$ of the vertex set $V(G)$, we say that V_i *shatters* V_j if there exist vertices $u, v \in V_j$ such that $d(u, V_i) \neq d(v, V_i)$. Then, the *shattering* of V_j by V_i is a partition of V_j, namely $\{V_{j_1}, \ldots, V_{j_t}\}$, such that $d(u, V_i) = d(v, V_i)$ for all $u, v \in V_{j_k}$ and all $k \in \{1, \ldots, t\}$. Given a partition $\mathcal{A}$, Algorithm 2.1, called *coarsest equitable refinement*, allows to compute $R(\mathcal{A})$ from $\mathcal{A}$.

Algorithm 2.1 Coarsest equitable refinement [McK81]

Input: a graph G, a partition $\mathcal{A} = \{V_1, \ldots, V_s\}$ of $V(G)$
Output: $R(\mathcal{A})$

```
P ← A
find a pair of subscripts (i, j) such that V_i shatters V_j
while (i, j) does exist do
    let {V_{j_1}, ..., V_{j_t}} be the shattering of V_j by V_i
    P ← {V_1, ..., V_{j-1}, V_{j_1}, ..., V_{j_t}, V_{j+1}, ..., V_s}
    find a pair of subscripts (i, j) such that V_i shatters V_j
end while
R(A) ← P
return R(A)
```

The coarsest equitable refinement algorithm iteratively splits the cells of $\mathcal{A}$ until the resulting partition is not equitable. Next two propositions states the correctness and the efficiency of the algorithm.

Proposition 2.4.3. *The coarsest equitable refinement algorithm returns the coarsest equitable partition finer than $\mathcal{A}$.*

Proof. If every cell of an incumbent partition cannot shatter any other cell, it means that the partition is equitable; therefore, the algorithm terminates after a bounded number of iterations since every graph has at least one equitable partition, i.e. the trivial partition. Moreover, the partition computed by the algorithm $R(\mathcal{A})$ is obviously finer than $\mathcal{A}$, as it is obtained by splitting cells of $\mathcal{A}$. Finally, we can conclude that $R(\mathcal{A})$ is the coarsest equitable partition finer than $\mathcal{A}$, since all partitions $\mathcal{P}$ computed by the algorithm are coarser than $R(\mathcal{A})$, but not equitable. □

Proposition 2.4.4. *The coarsest equitable refinement algorithm has a computational complexity of $O(nm)$, where $n = |V|$ and $m = |E|$.*

Proof. In the worst case, the algorithm visits on each iteration all the the edges of the graph to verify whether the current partition is equitable. Moreover, if the the graph has the trivial partition as unique equitable partition, the number of iterations is bounded by the number of vertices in V. Therefore, the computational complexity of the algorithm is $O(nm)$. □

Let us observe that Algorithm 2.1 allows to compute the coarsest equitable partition $\mathcal{P}^*(G)$ by setting as input $\mathcal{A} = \{V(G)\}$. Therefore, we can conclude that the computation of $\mathcal{P}^*(G)$ can be done in polynomial time.

2.5 EP-graph

In this section, we present our first original contribution. As we discussed in the introduction, exact methods for solving MSSP are based on branch&bbound paradigm. Therefore, instances of MSSP with property of symmetry are often challenging since branch&bound wastes computational time exploring equivalent subproblems.

One is then tempted of removing symmetries, e.g. with some symmetry-breaking methods. However, there are two drawbacks. The first is that avoiding symmetry requires a significant computational effort, even when orbits can be computed very fast in practice. The second, and more relevant, is that a complete removal of symmetry can have bad effects since information about the structure of instances gets lost. In fact symmetric instances have often a quite regular structure that one should try to exploit, while avoiding the combinatorial explosion due to the symmetry.

Equitable partitions, and, in particular, the orbit partition, decomposes a graph in more manageable classes that often have a "small" size and a simple symmetry structure. It is often the case, that in symmetric instances, these classes are connected not just in a regular way (this follows from the definition of equitable partition) but also in a *simple* way, e.g. tree like. In order to be more precise we need the following definition

Definition 2.5.1. *Given a graph G and an equitable partition $\mathcal{P} = \{V_1, \ldots, V_p\}$ of G, the EP-graph of G associated to $\mathcal{P}$, denoted by $G^{\mathcal{P}}$, is a simple graph that has a vertex $i \in V(G^{\mathcal{P}})$ for each cell $V_i \in \mathcal{P}$ and an edge $\{i, j\} \in E(\mathcal{P})$ if there exist vertices $u \in V_i, v \in V_j$ such that $\{u, v\} \in E(G)$.*

Fig. 2.4 exhibits an example of EP-graph: on the left-hand side, we show a graph and an equitable partition given by the coloring of vertices; on the right-hand side, the corresponding EP-graph is displayed.

As we will show in the following chapters, the EP-graph is often a powerful tool for capturing the regular structure of a graph due to properties of symmetry. Note also that, when we deal with *equitable* partitions, the EP graphs might show some symmetries that would be missed by just looking at orbit partitions.

We close by pointing out that, in some cases, the computation of the EP-graph associated to some equitable partition of a graph G allows us to preprocess G and therefore reduce its size. This is summarized in the following lemmas, whose simple proofs we skip. Given a subset $U \subseteq V$, let $\alpha(U)$ denote the stability number of $G[U]$.

Lemma 2.5.2. *Let $\mathcal{P} = \{V_1, \ldots, V_p\}$ be an equitable partition of a graph G. For each $V_i \in \mathcal{P}$, the following holds: if $\alpha(V_i) = \alpha(N[V_i])$, then $\alpha(G) = \alpha(G \setminus N[V_i]) + \alpha(V_i)$.*

An *homogeneous set* is a set of vertices $Q \subset V$ such that each vertex $v \notin Q$ is either complete or anti-complete to Q. We point out that checking whether a cell V_i is an homogeneous set is straightforward, given the parameters (c, D) (see Sec. 2.3) of the equitable partition. Namely, V_i is an homogeneous set if and only if, for each $j \neq i$, either $D_{ji} = 0$ or $D_{ji} = |V_i|$.

Lemma 2.5.3. *Let $\mathcal{P} = \{V_1, \ldots, V_p\}$ be an equitable partition of a graph G. For each $V_i \in \mathcal{P}$, the following holds: if V_i is an homogeneous set, then $\alpha(G) = \max\{\alpha(G \setminus N[V_i]) + \alpha(V_i), \alpha(G \setminus V_i)\}$.*

Figure 2.4: Example of EP-graph

Chapter 3

Symmetry in Maximum Stable Set Problem

The *maximum stable set problem* (MSSP) is known to be $\mathcal{NP}$-hard and, as well, extremely hard to approximate [Hås96]. Exact methods for MSSP are mainly based on the *branch&bound* paradigm. Although branch&bound is the most suitable method for solving MSSP, it could fail to solve instances with a few hundred vertices. Furthermore, instances with property of symmetry (or, more in general, integer programs with symmetric feasible regions) are challenging since branch&bound algorithms waste computational time exploring equivalent subproblems.

In this chapter we deal with exact methods for solving symmetric instances of MSSP. We recall in Section 1.2 some standard integer linear programming formulation for the maximum stable set problem. In Section 3.1, we give an overview of *Orbital Branching*, one of the most effective symmetry-breaking method for applications. Finally, in Section 3.2, we introduce some more (new) tools: equitable partition inequalities, equitable partition formulations and aggregate equitable partition formulations.

3.1 Orbital Branching

In this section, we discuss basic concepts of the *orbital branching* method [OLRS11a]: for an exhaustive exposition the reader may refers also to [Ost09]. A suitable application of orbital branching to MSSP is discussed in 4.2.

We start introducing some notations. Let $A \in \{0,1\}^{m\times n}$ be a binary matrix with m rows and n columns. Given sets of indices $I^m = \{1,\dots,m\}$ of the rows and $I^n = \{1,\dots,n\}$ of the columns of A, let Π^m and Π^n be respectively the symmetry group of I^m and I^n (for definition of group, symmetry group and permutation group, see Sec. 2.2). For each $\sigma \in \Pi^m, \pi \in \Pi^n$, let $A(\sigma,\pi) \in \{0,1\}^{m\times n}$ be the

matrix obtained by permuting rows of A by σ and columns of A by π.

The permutation group $\mathcal{G}(A)$ of the matrix A is the subset of permutations $\pi \in \Pi^n$ such that there exist permutation $\sigma \in \Pi^m$ which map $A(\sigma, \pi)$ onto A.

$$\mathcal{G}(A) = \{\pi \in \Pi^n : \exists \sigma \in \Pi^m \text{ such that } A(\sigma, \pi) = A\}$$

Given a subset $S \subseteq I^n$, the orbit of S under the action of $\mathcal{G}(A)$, denoted by $orb(S, \mathcal{G}(A))$, is the set of all subsets $S' \subseteq I^n$ such that there exists a permutation $\pi \in \mathcal{G}(A)$ which maps elements of S onto elements of S':

$$orb(S, \mathcal{G}(A)) = \{S' \subseteq I^n : \exists \pi \in \mathcal{G}(A) \text{ such that } S' = \pi(S)\}$$

Thus, the *orbit partition* of the columns of A, denoted by $\mathcal{O}(A)$, is the union of the orbits of each column with index $j \in I^n$:

$$\mathcal{O}(\mathcal{G}(A)) = \bigcup_{j=1}^{n} orb\left(\{j\}, \mathcal{G}(A)\right)$$

Now, we introduce the basic idea behind orbital branching. Let us suppose we want to solve the following binary integer program:

$$(ILP) = \max_{x \in \{0,1\}^n} \left\{c^T x : Ax \leq b\right\}$$

where b, c are w.l.o.g. integer vectors, by using a general branch&bound scheme. Let $\mathcal{F}$ be the feasible region of (ILP). Branch&bound partitions $\mathcal{F}$ building up an enumeration tree whose nodes are subproblems obtained by fixing the values of some variables. We define a node of the enumeration tree, denoted by a, with the pair of subsets (F_1^a, F_0^a) such that $F_1^a \subseteq I^n$, $F_0^a \subseteq I^n$ contain respectively indices of variables fixed to 1 and indices of variables fixed to 0. The set of free variables of node a is $N^a = I^n \setminus (F_1^a \cup F_0^a)$. The feasible region of a, say $\mathcal{F}^a$, is described by the submatrix $A(F_1^a, F_0^a)$ that is obtained by removing from A all columns with indices in $F_1^a \cup F_0^a$ and all rows intersecting columns with indices in F_1^a.

Given $S = \{j_1, \ldots, j_{|S|}\} \subseteq N^a$, a standard branching rule partitions $\mathcal{F}^a$ by fixing to 1 in turn each variable with index in S, implementing the following disjunction:

$$x_{j_1} = 1 \vee (x_{j_2} = 1 \wedge x_{j_1} = 0) \vee \cdots \vee (x_{j_k} = 1 \wedge \sum_{h=1}^{k-1} x_{j_h} = 0) \vee \cdots \vee \sum_{h=1}^{|S|} x_{j_h} = 0$$

Thus, a general branch&bound scheme generates a child subproblem $a(k)$ for each $k \in S$, leaving aside the effects of $\mathcal{G}(A(F_1^a, F_0^a))$ onto feasible regions of children of a. On the other hand, orbital branching exploits symmetry information by avoid-

ing the generation of equivalent subproblems. Given an orbit $O = \{j_1, \ldots, j_{|O|}\} \subseteq N^a$ under the action of $\mathcal{G}(A(F_1^a, F_0^a))$, the authors of [OLRS11a] show that for each pair $\{j_u, j_v\} \subseteq O$, subproblems $a(u)$ and $a(v)$ are equivalent since they are characterized by symmetric solutions. In particular, orbital branching implements the following branching dichotomy based on the orbit partition $\mathcal{O}(\mathcal{G}(A(F_1^a, F_0^a)))$:

$$x_{j_u} = 1 \vee \sum_{h=1}^{|O|} x_{j_h} = 0 \qquad j_u \in O$$

The basic orbital branching method is formalized in Algorithm 3.1.

Algorithm 3.1 Orbital branching

Input: subproblem $a = (F_1^a, F_0^a)$
Output: two child subproblems l, r
compute orbit partition $\mathcal{O}(\mathcal{G}(A(F_1^a, F_0^a))) = \{O_1, \ldots, O_p\}$
Select orbit O_{j^*} , $j^* \in \{1, \ldots, p\}$
Choose arbitrary $k \in O_{j^*}$
return $l = (F^1 \cup \{k\}, F^0)$ and $r = (F^1, F^0 \cup O_{j^*})$

The choice of the orbit may influence the performance of the method: the authors compare several rules for deciding the orbit on which to base the branching.

In practice, orbital branching is one of the most effective symmetry-breaking methods. Computational results on symmetric instances show that orbital branching performs much better than CPLEX with symmetry-reduction setting and it is also comparable to isomorphism pruning. However, orbital branching has two disadvantages. In terms of efficiency, computing orbits at each node of the enumeration could require such a computational effort as the purpose of removing symmetry is defeated, even though the orbit partition can be efficiently computed in practice. The authors propose a way around this first disadvantage by deriving local permutation groups of subproblems from the global permutation group of the entire problem. This approach decreases the orbits computation overhead, but it is not completely conclusive since it weakens the orbital branching dichotomy.

A more significant disadvantage is implied by the fact that the performance of orbital branching is closely related to the structure of permutation group of the problem. After few fixing of variables, it may happen that the symmetry is completely removed from the problem, thus, orbital branching could perform much worse than standard branch&bound methods. Some feasible regions with a wide permutation group are such that if we remove exactly one column from the corresponding orbit, i.e. we artificially distinguish a column from all its symmetric columns, the size of the corresponding permutation subgroup considerably decreases with respect to the size of the original permutation group. On the contrary, it may occur that the size

of permutation group of the same feasible region will not be subject to substantial variations if we partition a certain orbit in more subsets of columns.

3.2 Equitable partition inequalities

The performances of orbital branching suggests that there is a trade-off between removing and maintaining symmetry in integer problems: on one hand it is useful avoiding symmetry in order to reduce the search space of solutions, on the other hand it is advantageous preserving some symmetry information.

The main contribution of this thesis, we believe, is that of showing that the EP graphs are a powerful tool to find equilibria in removing and maintaining symmetry. In many cases, we realized that hard instances have, for some suitable equitable partition, a very nice and simple EP graph (often a tree!). In such cases, it is often possible to efficiently compute the stability number of subgraphs that are suggested by the EP graph, that have a "small" size and a simple symmetry structure. In fact, these "local" lower bounds provide quite valuable additional informations that might reduce the search space of solutions.

While we believe that there are several methods for exploiting these informations (e.g. via combinatorial branch&bound algorithms), in the thesis we focus on how to derive from EP graphs valid inequalities that strengthen the linear programming relaxations of the maximum stable set problem. That is, indeed quite simple. Suppose that we are given a graph G and an equitable partition $\mathcal{P}$ of G. We derive from the topology of EP-graph $G^{\mathcal{P}}$ the following rank inequalities, called *equitable partition inequalities*:

$$x(V_i) \leq \alpha(V_i) \qquad \forall i \in V(G^{\mathcal{P}}) \qquad (3.1)$$

$$x(V_i) + x(V_j) \leq \alpha(V_i \cup V_j) \qquad \forall \{i,j\} \in E(G^{\mathcal{P}}) \qquad (3.2)$$

$$\sum_{j \in N_{G^{\mathcal{P}}}[i]} x(V_j) \leq \alpha\left(\bigcup_{j \in N_{G^{\mathcal{P}}}[i]} V_j\right) \qquad \forall i \in V(G^{\mathcal{P}}) \qquad (3.3)$$

$$x(V_i) + x(V_j) + x(V_k) \leq \alpha(V_i \cup V_j \cup V_k) \qquad \forall \{i,j,k\} \text{ triangle of } G^{\mathcal{P}} \qquad (3.4)$$

A few remarks are necessary and important:

- We remark that other inequalities than the one above could be derived from the topology of EP-graph (e.g. one might be tempted to look at inequalities associated to odd cycles in the EP graph). However, our computational observations show that the subgraphs induced by the classes involved in the above

EP inequalities have often a small size and a simple symmetry structure. And this takes us to our second remark:

- Computing right-hand sides of the above inequalities requires to find the stability numbers of some subgraphs of G, hence an additional computational effort is necessary. However, if these subgraphs have a small size and a simple symmetry structure, then the computation of right-hand sides can be often effectively carried out by eventually using symmetry-breaking method like orbital branching.

- Trivially, we might restrict to orbit partitions, instead than equitable partition. But, on the other hand, why? We are not planning to use these partitions to *fix* the values of some variables (in *that* case we would need not just equitable partitions, but orbit partitions). So we better use equitable partitions, as they are more (and so it is more likely that we find one with a nice EP graph), and are easy to find (recall that it is easy to find the coarsest one, and it is easy to find the coarsest equitable refinement of any other partition, see Section 2.4.1).

Given an equitable partition $\mathcal{P}$ of a graph G, we add the equitable partition inequalities associated to $\mathcal{P}$ to $QSTAB(G)$ and get a new formulation of the maximum stable set problem, that we call *equitable-partition formulation* (associated with P):

$$
\begin{aligned}
(EQP) := \quad & \max \sum_{i \in V(G)} x_i \\
& \text{subject to} \\
& x \text{ satisfies (3.1)-(3.3) } \forall \mathcal{P} \in \mathcal{P}^\star(G) \\
& x \in QSTAB(G) \cap \mathbb{Z}_+^{|V|}
\end{aligned}
\tag{3.5}
$$

Trivially, if we are given a *family* of equitable partition of G we may simultaneously add all the equitable partition inequalities associated to each equitable partition in the family

3.2.1 Aggregate equitable partition formulation

Program 4.1 could be unsolvable in practice, even when a considerable computation capacity is employed. In this regard, we present a reformulation of MSSP in a lower dimension than of the original space, which provides upper bound on the optimal solution. Given a graph G and equitable partition $\mathcal{P}$ of G, let y_i the number of

vertices that a stable set S intersects $V_i \in \mathcal{P}$. Since $|S \cap U| \leq \alpha(U)$ trivially holds for each $U \subseteq V(G)$, the optimal solution of the following linear program, called *aggregate equitable partition formulation*, is an upper bound on $\alpha(G)$:

$$(AEP) := \quad \max \quad \sum_{i \in V(G^{\mathcal{P}})} y_i \tag{3.6}$$

subject to

$$y_i \leq \alpha(V_i) \quad \forall i \in V\left(G^{\mathcal{P}}\right) \tag{3.7}$$

$$y_i + y_j \leq \alpha(V_i \cup V_j) \quad \forall \{i,j\} \in E\left(G^{\mathcal{P}}\right) \tag{3.8}$$

$$y_i + y_j + y_k \leq \alpha(V_i \cup V_j \cup V_k) \quad \forall \{i,j,k\} \text{ triangle of } G^{\mathcal{P}} \tag{3.9}$$

$$\sum_{j \in N_{G^{\mathcal{P}}}[i]} y_j \leq \alpha\left(\bigcup_{j \in N_{G^{\mathcal{P}}}[i]} V_j\right) \quad \forall i \in V\left(G^{\mathcal{P}}\right) \tag{3.10}$$

$$y \in \mathbb{Z}_+ \quad \forall i \in V\left(G^{\mathcal{P}}\right) \tag{3.11}$$

Constraints (3.7)-(3.10) are equitable partition inequalities in the aggregate space. Let us observe that (AEP) becomes a linear integer formulation of MSSP in the original space when $\mathcal{P}$ is a trivial equitable partition, i.e. each class of $\mathcal{P}$ contains exactly one vertex of G. However, if we consider equitable partitions with a small number of classes, the optimal solution of (AEP) can be found very fast in practice.

Let us consider the example shown in Fig. 2.4. The corresponding aggregate

equitable partition formulation is given by the following integer program:

$$
\begin{aligned}
\max \quad & y_1 + y_2 + y_3 + y_4 \\
\text{subject to} \quad & \\
& y_1 \leq 1 \\
& y_2 \leq 1 \\
& y_3 \leq 1 \\
& y_4 \leq 1 \\
& y_1 + y_2 \leq 1 \\
& y_2 + y_3 \leq 2 \\
& y_3 + y_4 \leq 1 \\
& y_1 + y_2 + y_3 \leq 2 \\
& y_2 + y_3 + y_4 \leq 2 \\
& y_i \in \mathbb{Z}_+ \quad \forall i = 1, \ldots, 4
\end{aligned}
$$

In the following chapter, computational results show that equitable partition inequalities can be efficiently generated, even without using symmetry breaking methods for the computation of right-hand sides. Moreover, we will see that equitable partition formulations often provide strong relaxation of MSSP and allow to efficiently solve some hard symmetric instances.

Chapter 4

Computational Experiments

In this chapter we summarize our computational experiments on some hard instances of the maximum stable set problem. In Section 4.1 we present our main formulation and discuss strategies for generating equitable partition inequalities and designing orbital branching for maximum stable set problem. In Section 4.2 we give a few technical details about our implementations. In Section 4.3, we illustrate a few symmetric instances of the stable set instances that we focused on, and report our computational experiments.

4.1 Our main integer linear program

Our standard integer linear program formulation for the maximum stable set problem is the equitable partition formulation we discussed in the previous chapter. Namely, given a family of equitable partition $\mathcal{P}^*$ of G, we consider:

$$\begin{aligned} (EQP) := \quad & \max \sum_{i \in V(G)} x_i \\ & \text{subject to} \\ & x \text{ satisfies (3.1)-(3.3) } \forall \mathcal{P} \in \mathcal{P}^* \\ & x \in QSTAB(G) \cap \mathbb{Z}_+^{|V|} \end{aligned} \tag{4.1}$$

The generation of this formulation, requires the following tasks:

(1) computing a $\mathcal{P}^*$ of equitable partitions;

(2) computing for each $\mathcal{P} \in \mathcal{P}^*$, and for each equitable partition inequality derived from that partition, the corresponding right-hand sides.

As we have discussed in Section 2.4.1, the coarsest equitable partition $\mathcal{C}$ can be computed in polynomial time by the coarsest equitable refinement procedure, starting from the unit partition $\{V(G)\}$ (see Algorithm 2.1). However, our experiments have shown that the potentiality of equitable partition formulation turns out to be restricted if (EQP) contains only inequalities uniquely derived from $\mathcal{C}$. In many cases, using inequalities derived from equitable partitions finer than $\mathcal{C}$ has been successful. Thus, a first crucial question is *how* to derive other equitable partitions than the coarsest.

That is easy, however, as the coarsest equitable refinement procedure is able to produce the coarsest equitable partition that is a refinement of any given partition. In particular, we suggest the possible simple procedure to produce a different equitable partition $\mathcal{P}'$ from an equitable partition $\mathcal{P}$. Given an equitable partition $\mathcal{P} = \{V_1, \ldots, V_p\}$ (possibly $\mathcal{P} = \mathcal{C}$) and one vertex $u, \in V_i$, let $\mathcal{P}'$ be the coarsest refinements of $\{V_1, \ldots, V_{i-1}, \{u\}, V_i \setminus \{u\}, V_{i+1}, \ldots, V_p\}$. We point out that we can choose $v \in V_i$ into $|V_i|$ possible different ways. However, in many symmetric instances, the equitable partitions we get isolating u and v are often characterized by the same parameters, and therefore produce inequalities whose right-hand sides require almost the same computational time. That means that such a breadth-first strategy does not pay off when we are simply looking for some partition for which computing the right-hand sides of the equitable partition inequalities is doable, see the following.

So the question is: *when* other equitable partition should we generate? This is, we believe a fascinating question, we attempt here a preliminary answer, based on computational analysis and empirical observations.

First of all, we have empirically observed that there is a trade-off between using equitable partitions with a small number of classes and equitable partitions with many classes. Usually, equitable partitions with few classes give rise to strong inequalities, however the computation of the corresponding right-hand sides could require a considerable computational effort. On the other hand, equitable partitions with many classes often lead to weak inequalities whereas their generation is very fast.

Our final procedure is again simple. We start with the coarsest equitable partition. We then generate a new equitable partition $\mathcal{P}'$ from the current equitable partition $\mathcal{P}$ (as discussed above) when: either we are not able to compute the right-hand sides for some inequality associated to $\mathcal{P}$, or the the upper bound associated to the current equitable partition formulation (which takes into account $\mathcal{P}$ and possibly other equitable partitions generated in the past) is too weak. Algorithm 4.1 formalizes such considerations.

Conditional expression at row 9 of Algorithm 4.1 refers to cases when the com-

Algorithm 4.1 Equitable partition inequalities generation scheme

Input: a graph $G = (V, E)$, clique formulation $QSTAB(G)$
Output: equitable partition inequalities $\mathcal{I}$ for G

1: $\mathcal{I} \leftarrow \emptyset$
2: compute coarsest equitable partition $\mathcal{C} = \{V_1, \ldots, V_{|\mathcal{C}|}\}$ of G
3: $\mathcal{Q} \leftarrow \mathcal{C}$
4: $flag \leftarrow$**false**
5: **while** $flag =$**false do**
6: $flag \leftarrow$**true**
7: compute equitable partition inequalities $\mathcal{I}^{\mathcal{Q}}$ associated to $\mathcal{Q}$
8: $\mathcal{I} \leftarrow \mathcal{I} \cup \mathcal{I}^{\mathcal{Q}}$
9: **if** rhs of some inequality in $\mathcal{I}^{\mathcal{Q}}$ is hard to compute **then**
10: $flag \leftarrow$**false**
11: **end if**
12: **if** $QSTAB(G) \cap \mathcal{I}$ is weak **then**
13: $flag \leftarrow$**false**
14: **end if**
15: **if** $flag =$**falseand** $|\mathcal{Q}| < |V|$ **then**
16: $i \leftarrow \arg\min_{j=1,\ldots,|\mathcal{Q}|}\{|V_j| : |V_j| \geq 2\}$
17: select $u \in V_i$
18: $\mathcal{Q} \leftarrow R(\{V_1, \ldots, V_{i-1}, \{u\}, V_i \setminus \{u\}, V_{i+1}, \ldots, V_p\})$ {see Section 2.4.1}
19: **end if**
20: **end while**
21: **return** $\mathcal{I}$

putation of inequalities right-hand sides requires an effort in terms of computational time or memory needed for storing branch&bound search tree: we assume that the performance of the method is acceptable within fixed limits of time and size of the tree. In the same way, conditional expression at row 12 typifies situations in which equitable partition formulations do not allow to efficiently solve instances in terms of computational time or growing of the branch&bound tree.

We point out two positive sides of Algorithm 4.1. Given an equitable partition $\mathcal{Q}$, let $\mathcal{I}^{\mathcal{Q}}$ be the set of equitable partition inequalities (3.1)-(3.4) associated to $\mathcal{Q}$. The generation of each inequality in $\mathcal{I}^{\mathcal{Q}}$ requires to solve a certain instance of maximum stable set problem that corresponds to a subgraph of G. Thus, the generation of each inequality in $\mathcal{I}^{\mathcal{Q}}$ can be carried out independently from other inequalities in $\mathcal{I}^{\mathcal{Q}}$. It follows that parallel computing is possible.

Moreover, let us observe that subgraphs associated to equitable partition inequalities contain local symmetry information of G. This feature could allow orbital branching to effectively compute right-hand sides of inequalities. We have discussed basic ideas and an essential functioning of the method in Section 3.1. A natural implementation of orbital branching is based on the following simple observation. Given a graph G, let $M \in \{0,1\}^{|V|\times|E|}$ be the incidence matrix of G and, given a clique K_i in G, let $\chi(K_i) \in \{0,1\}^{|V|}$ the incidence vector of K_i. Now, let us consider the following integer formulation of the maximum stable set problem for

G:

$$\max_{x \in \{0,1\}^{|V|}} \left\{ \sum_{i=1}^{|V|} : Ax \leq \mathbf{1}_{|V|} \right\}$$

where $\mathbf{1}_{|V|}$ is a vector of all ones with size $|V|$ and $A = \begin{pmatrix} M^T \\ \chi(K_1)^T \\ \vdots \\ \chi(K_q)^T \end{pmatrix}$. Since each column of A is in bijection with a vertex of G, the symmetry group $\mathcal{G}(A)$ corresponds to $\mathrm{Aut}(G)$ (see Sec. 2.2 for definitions), hence the orbit partition of columns of A is given by the orbit partition of vertices of G. For this reason, the orbit partition at each subproblem in the search tree corresponds to orbit partition of the suitable subgraph obtained by removing vertices referring to fixed variables. Since `Nauty` is very effective for computing orbits of graphs, we have implemented orbital branching to perform the computation of orbits for each explored subproblem of the enumeration tree. Furthermore, we have adopted the rule of branching on the largest orbit since it is one of the most effective rule to perform orbital branching (as described in [OLRS11a]) and, it allows to exploits the previous implementation choice in the best way possible.

4.2 Implementation

The entire implementation has been developed in C language and consists of over 5000 lines of code (see Appendix B). The generation scheme of equitable partition inequalities (discussed in the previous section) has been implemented by several functions of our source code. Table 4.1 summarizes main procedure of Algorithm 4.1, whose code is reported in Appendix B. The computation of coarsest equitable partition is implemented in function `cep`, while the process of finding finer equitable partitions is due to the interaction of functions `refineEQP` and `genEQP`. The generation of equitable partition inequalities refers to functions `printEQP_v_rhs`, `printEQP_e_rhs`, `printEQP_t_rhs` and `printEQP_n_rhs`. These procedures exploit subroutine `verifyISO` that allows to avoid the computation of equivalent right-hand sides by invoking `Nauty`. Finally, the computation of right-hand side have been carried out by IBM ILOG CPLEX 12.4 [IBM11] using `callable library`, the C programming interface of CPLEX. In our code, function `cpxRHS` implements a usage of CPLEX with standard features, while function `orbRHS` exploits CPLEX `callbacks` to implement orbital branching. Finally, procedures `printEQP`, `printEQPpar` and `EPgraphDOTform` are important tools to analyze the structure of instances: for each

equitable partition considered, these routines provide equitable partition parameters (see Sec. 2.3) and a file for visualizing EP-graphs by `Graphviz` [Res88], an open source graph visualization software.

4.3 Computational results

This section is devoted to the discussion of computational results. Experiments have been carried on a workstation with 4 processors at 2.0 GHz and 8GB of RAM under Linux operating system. The source code has been compiled by `gcc` with optimization flag. Computational experiments concern symmetric stable set instances, thus we have focused our attention on classes of symmetric graphs from *Dimacs clique benchmark set* [Dim92], that are *Mann* and *Keller*. Mann and Keller graphs have been artificially generated in order to test algorithms for maximum stable set problem and, more in general, integer linear programming methods. They are widely considered hard instances of benchmark set Dimacs. Our first purpose is testing the performance of our method comparing it with orbital branching, which is one of the most effective symmetry-breaking method in practice.

We have also considered instances arising from applications: *1zc*-graphs. The maximum stable set problem in 1zc-graphs corresponds to a reformulation of a well-studied problem in coding theory. This class of graphs turns out to be challenging since no optimal solutions are known for largest instances of the class. In Section 4.4, we will see that equitable partition formulations allows to certify the optimality for 1zc1024, a 1zc-graph whose stability number is not known so far, also defined *most wanted* by N. Sloane [Slo00].

4.4 1zc-instances

The 1zc-instances are a class of instances from the Sloane *Independent Set Challenge* [Slo00] known to be among the hardest instances for the maximum stable set problem. In fact, for most instances of this class the optimal solutions is still unknown.

As we will show in this section, the graphs arising in these instances admit a quite natural equitable partition whose EP-graph is a path. The corresponding equitable partition formulations, in the original and the aggregate space, are quite effective. We have in fact been able to solve to optimality the 1zc1024-instance, defined as “the most wanted instance” by Sloane. We also substantially improved the best upper bound on the value of the optimal solution to the 1zc2048-instance.

More details about the 1zc-instances are available on the Sloane’s independent set challenge web page [Slo00].

	Function	Description
(B.1)	`cep`	The function computes the coarsest equitable partition.
(B.3)	`refineEQP`	Given equitable partition $\mathcal{P}$, procedure computes an equitable partition $\mathcal{Q}$ finer than $\mathcal{P}$ guaranteeing minimal fragmentation of the classes of $\mathcal{P}$.
(B.4)	`genEQP`	The function explores the space of equitable partitions according to desired parameters.
(B.5)	`printEQP_rhs`	Given an equitable partition $\mathcal{P}$, the function initializes the generation of equitable partition inequalities by extracting corresponding subgraphs of G.
(B.6)	`verifyISO`	The function invokes `Nauty` to test isomorphism among subgraphs extracted by `printEQP_rhs`, in order to avoid the computation of equivalent right-hand sides.
(B.7)	`cpxRHS`	The function computes the right-hand side of equitable partition inequality by using callable library of CPLEX.
(B.8)	`orbRHS` `usersetbranch`	`orbRHS` computes the right-hand side of equitable partition inequality by implementing orbital branching under CPLEX. `usersetbranch` modifies the generation of subproblems in CPLEX enumeration tree: at each node node of tree, `Nauty` is invoked to compute orbits, then two children are generated by branching largest orbit, according to the orbital branching dichotomy.

(B.2)	`printEQP`
	`printEQPpar`
	`EPgraphDOTForm`
		Functions extract information from equitable partition: they generate equitable partition parameters and a file to visualize the corresponding EP-graph by `Graphviz`

Table 4.1: Main procedures of equitable partition inequalities generation scheme

4.4.1 Origin of the 1zc-instances

In this subsection we describe the context from which the 1zc-instances arise. This requires that a few basic definitions from information and coding theory. In the following, it is convenient to think of the following (practical) problem: a sender needs to communicate some message to a receiver, and we may assume that the message is from some finite set of messages M.

In order to communicate his/her message, the sender will make use of some (noisy) channel. A *channel* is a theoretical model with certain *error* characteristics that describes the process of conveying information signals from one sender to one receiver. In a *binary communication channel*, the sender wishes to send bits, and the receiver receives bits. In a binary communication channel, an error, in general, refers to a crossover $1 \to 0$ or $0 \to 1$ from the input to the output of the channel. We are here interested in a particular binary communication channel:

Definition 4.4.1. *The Z-channel (or* binary completely asymmetric channel*) is the channel with* $\{0,1\}$ *as input and output, where the crossover* $1 \to 0$ *occurs with positive probability* p*, whereas the crossover* $0 \to 1$ *never occurs.*

For example, in the Z-channel the error given by the crossover $(1110010) \to (1110011)$ cannot occur since the last entry of the message changes value from 0 to 1 during transmission.

As the sender uses a binary communication channel, he needs to associate a binary code to each message in M. A *binary code* is a way of representing text or computer processor instructions by the use of bits. Formally, a *binary code of length* n, denoted by C, is a subset of binary vectors in $\{0,1\}^n$. We may therefore think that our problem is that of choosing $C \subseteq \{0,1\}^n$ and defining a bijection between C and the set M of messages.

Trivially, this requires that $n \geq \lceil \log_2 |M| \rceil$; however it might be convenient to choose a number n of bits larger than $\lceil n \log_2 |M| \rceil$, so that we may use an *error-correcting code*. The central idea behind error-correcting codes is that sender encodes their message in a redundant way: the redundancy allows the receiver to detect a limited number of errors that may occur anywhere in the message, and often to correct these errors without retransmission. The following error-correcting code is designed for Z-channels:

Definition 4.4.2. *Given* $x \in C \subseteq \{0,1\}^n$*, let* $y \in \{0,1\}^n$ *be a binary vector of length* n *that is obtained from* x *by changing at most* t *"1"s into "0"s (and keeping unchanged the other bits of* x*). Then,* C *is a* t asymmetric error correcting code *(or simply* t*-code) if there exists a function* $\psi : \{0,1\}^n \to C$*, called* decoder*, which recovers* x *from* y*, i.e.* ψ *univocally maps* y *onto* x*.*

Let us consider the following binary vectors, and suppose that $t = 1$:

$$x_1 = (01011) \quad x_2 = (10101) \quad x_3 = (11000) \quad x_4 = (10011) \tag{4.2}$$

Let us check that $C_1 = \{x_1, x_3, x_4\}$ is not a 1-code since a rule that decodes $y = (00011)$ onto any $x_i \in C_1$ does not exist: there is no way to tell if y is obtained from x_1 or from x_4. At contrary, $C_2 = \{x_1, x_2, x_3\}$ is a 1-code since we can map each y, that is obtained from some $x \in C_1$ by changing at most one "1" into "0", into a unique vector $x_i \in C_2$.

Given the set of all binary vectors $\{0,1\}^n$, the problem of finding a t-code $C \subseteq \{0,1\}^n$ of maximum cardinality is a well-known problem of coding theory. This problem can be easily reformulated as a maximum stable set problem, as we show in the following.

Definition 4.4.3. *Given $x, y \in \{0,1\}^n$, the asymmetric distance between $x$and y, denoted $\Delta(x, y)$, is:*

$$\Delta(x, y) := \max\{|\{i : x_i > y_i, i = 1, \ldots, n\}|, |\{i : y_i > x_i, i = 1, \ldots, n\}|\} \tag{4.3}$$

The proof of following theorem is essentially taken from [Klø81].

Theorem 4.4.4. *A code $C \in \{0,1\}^n$ is a t-code if and only if $\Delta(x, y) > t$ holds for all $x, y \in C, x \neq y$.*

Proof. For each $x, y \in \{0,1\}^n$, let $N(v, x) := |\{i : x_i > v_i, i = 1, \ldots, n\}|$. It follows that $\Delta(x, y) = \max\{N(v, x), N(x, v)\}$. Also, for each $x \in \{0,1\}^n$, let $S_t(x) = \{v \in \{0,1\}^n : v \leq x \text{ and } N(v, x) \leq t\}$.

It follows from Definition 4.4.1 that C is a t-code if and only if $S_t(x) \cap S_t(y) = \emptyset$ for each $x, y \in C, x \neq y$. We now show that, for each $x, y \in \{0,1\}^n$, $S_t(x) \cap S_t(y) \neq \emptyset$ if and only if $\Delta(x, y) \leq t$.

First, consider x and $y \in \{0,1\}^n$ such that $\Delta(x, y) \leq t$. Define a third vector $v \in \{0,1\}^n$ such that $v_i = 1$ if and only if $x_i = y_i = 1$. By construction, $v \leq x$ and $v \leq y$. Moreover, $v_i = 0$ and $x_i = 1$ (resp. $y_i = 1$) if and only if $x_i = 1$ and $y_i = 0$ (resp. $x_i = 0$ and $y_i = 1$). Since $\Delta(x, y) \leq t$, it follows that $v \in S_t(x) \cap S_t(y)$.

Now consider x and $y \in \{0,1\}^n$ such that $S_t(x) \cap S_t(y) \neq \emptyset$, and let $v \in S_t(x) \cap S_t(y)$. By definition, $N(v, x) \leq t$ and $N(v, y) \leq t$. Moreover, it is easy to check that $N(x, y) \leq N(v, y)$ and $N(y, x) \leq N(v, x)$. But then $\Delta(x, y) = \max\{N(v, x), N(x, v)\} \leq t$. □

Let G^n be a graph such that its vertices are in bijection with the binary vectors in $\{0,1\}^n$ and $\{x, y\} \in E(G^n)$ if and only if $\Delta(x, y) \leq t$ (for simplification, we use

the same notation for a vertex and its corresponding vector). It follows from the previous theorem that finding a largest t-code of length n is equivalent to finding a maximum stable set problem in G^n.

4.4.2 The 1zc-graphs

When $t = 1$, graphs G^n (see previous section) are known as *1zc-graphs*. In the thesis, we focused in particular on G^9, G^{10} and G^{11}.

Our first remark is that the EP-graphs associated to the coarsest equitable partitions of G^9, G^{10} and G^{11} have an interesting path structure, that is shown in Fig. 4.1.

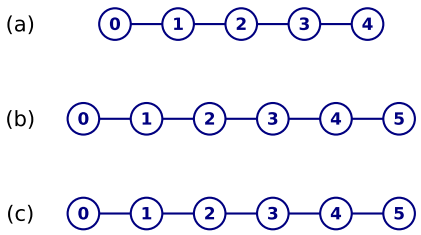


Figure 4.1: EP-graphs associated to coarsest equitable partitions of 1zc512 (a), 1zc1024 (b) and 1zc2048 (c)

However, for each of those graphs, the subgraphs induced by each of the coarsest partition (but for the rightmost one, see Fig. 4.1) is made of two connected components. Actually, if we split each non-connected cell into its two "connected" classes, we get another partition of the vertex set that is still equitable and whose EP-graph still holds a path structure, see Fig. 4.2.

Inspired by these observations, we could prove the following theorem showing that this structure is indeed common to each 1zc instance. We point out that, while the proof of the theorem is simple, its statement was suggested by the observation of the EP-graph.

Theorem 4.4.5. *Each graph G^n has an equitable partition with classes $V_0, V_1, \ldots, V_n$ such that the EP-graph is a path. Moreover, the parameters are the following*

- *for each $0 \leq i \leq n$, $d(i,i) = i(n-i)$;*
- *for each $1 \leq i \leq n$, $d(i,i-1) = i$*
- *for each $0 \leq i \leq n-1$, $d(i,i+1) = n-i$*
- *for each $0 \leq i < j \leq n$, with $j - i \geq 2$, $d(i,j) = 0$.*

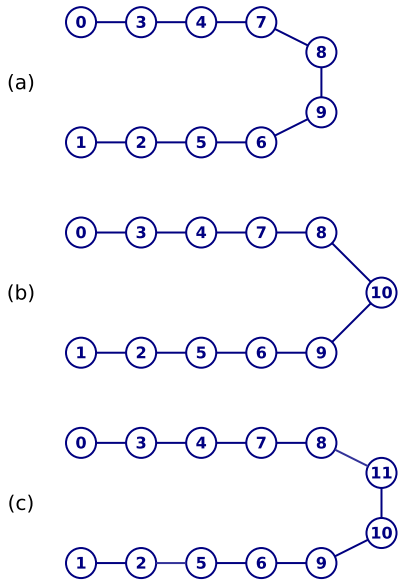


Figure 4.2: EP-graphs associated to non-coarsest equitable partitions of 1zc512 (a), 1zc1024 (b), 1zc2048 (c)

Proof. Each vertex $v \in G_n$ corresponds to a binary vectors in $\{0,1\}^n$ and in the proof we use the same notation for a vertex and its corresponding vector.

For each $i = 0..n$ let V_i be the set of vertices with exactly i bits of value 1. (We point out, and apologize for that, that this numeration is not consistent with the one that is given in Fig. 4.2). Now observe that the asymmetric distance between two vertices $u \in V_i$ and $v \in V_j$ is at least $|j - i|$. Moreover, each $u \in V_i$, $1 \leq i \leq n$, is adjacent to some vertex $v \in V_{i-1}$ (build v from u by just changing one "1" bit of u into "0"). Therefore the EP-graph associated to the partition $V_0, V_1, \ldots, V_n$ is a path.

However, we are a bit abusing terminology, because we still have to prove that $V_0, V_1, \ldots, V_n$ is an *equitable* partition! But again this is easy. Each vertex $u \in V_i$ is adjacent to $i(n-i)$ vertices of its class, i vertices of V_{i-1} and $n-i$ vertices of V_{i+1}. □

4.4.3 The maximum stable set problem on 1zc-graphs

The 1zc-graphs G^n provide hard instances for the maximum stable set problem, the so-called 1zc-instances (for each n, the 1zc2^n-instance corresponds to the graph G^n). In fact, no optimal solution is known for $n \geq 10$ and even estimating upper bounds on stability numbers turns out to be quite challenging. A first bound on the stability number of these graphs follows from [Var65], where the author gives a formula to obtain upper bound on the size of a maximum cardinality t-code of

length n. For the case $t = 1$, this corresponds to the following bound on $\alpha(G^n)$:

Theorem 4.4.6 ([Var65])**.**

$$\alpha(G^n) \leq \frac{2^{n+1}}{n+2}$$

A different line of attack to those instances was carried out in [Kløø81]. The author provides a linear integer formulation that allows to estimate upper bounds on the maximum cardinality t-code of length n (and therefore on the value of $\alpha(G^n)$). Unfortunately, the generation of some inequalities that are required for this integer program is in practice prohibitive. This was however improved in [EO98], where the authors are able to compute, in some cases, these upper bounds. They also are able to provide an heuristic algorithm to produce lower bounds, and eventually improve the best upper bounds known for many instance and solve to optimality the 1zc512-instance.

We here discuss (a part of) our computational experience with the maximum stable set problem on 1zc-graphs. Again, we focus in particular on 1zc512, 1zc1024 and 1zc2048 (i.e. $n = 9, 10, 11$). We built upon the equitable partitions returned by Theorem 4.4.5. We considered both the equitable partition formulations in the original space and in the aggregated space. We point out that in some cases CPLEX could not compute the right-hand sides of some of our equitable partition inequalities, even with symmetry reduction settings. On the other hand, orbital branching turned out to be effective for the computation of those right-hand sides, and in Table A.3, we report the computational performance for generating our equitable partition inequalities.

We were always able to solve our the aggregate equitable partition formulations (interestingly there was never integrality gap for these formulations). As for the equitable partition formulation in the original space, we were able to solve it only for 1zc512. However, for 1zc1024 and 1zc2048, on the one hand, the aggregate formulations provide upper bounds stronger than the best upper bounds known so far; on the other, the best integer solutions found by CPLEX are often the best lower bounds known so far. This is summarized in Table 4.2, where we compare: our upper bounds, denoted by "EQP" with other bounds known in literature and the bound provided by a semidefinite relaxation, denoted by"ϑ"; our upper bounds, denoted by"EQP_AS", with other upper from another upper bound from the literature.

4.5 Solving 1zc1024 to optimality

Table 4.2 shows that aggregate equitable partition formulations provide upper bounds stronger than the best upper bounds known so far: in particular, for 1zc1024, the

1zc-graphs	Upper Bounds				Lower Bounds	
	ϑ	[Klø81]	[EO98]	EQP	[EO98]	EQP_AS
1zc512	68.7500	64	62	62	62	62
1zc1024	128.6667	118	117	113	112	112
1zc2048	237.4000	210	210	202	198	196

Table 4.2: Improving upper bound for some 1zc-graphs

gap between the upper bound provided by the aggregate formulation and the lower bound provided by the best known solution is equal to 1.

However, a closer look to the EP-graph associated to the equitable partition returned by Theorem 4.4.5 (see Fig. 4.2(b)) and some more computation allowed us to prove the following:

Theorem 4.5.1. *The stability number of 1zc1024 is 112.*

Proof. Let us consider the equitable partition $\mathcal{P} = \{V_0, \dots, V_{10}\}$ of 1zc1024 whose EP-graph is shown in Fig. 4.2(b). The corresponding aggregate equitable partition formulation has the following inequalities:

$$y_4 \leq 5 \tag{4.4}$$
$$y_5 \leq 5 \tag{4.5}$$
$$y_0 + y_3 \leq 1 \tag{4.6}$$
$$y_1 + y_2 \leq 1 \tag{4.7}$$
$$y_7 + y_8 \leq 35 \tag{4.8}$$
$$y_6 + y_9 \leq 35 \tag{4.9}$$
$$y_7 + y_8 + y_{10} \leq 66 \tag{4.10}$$
$$y_6 + y_9 + y_{10} \leq 66 \tag{4.11}$$

By summing inequalities (4.4)-(4.7), we obtain:

$$y_0 + y_1 + y_2 + y_3 + y_4 + y_5 \leq 12$$

Analogously, we get:

$$y_6 + y_7 + y_8 + y_9 + y_{10} \leq 101$$

by summing (4.8) and (4.11) or (4.9) and (4.10). Thus, we can easily see that if there exists a stable set of cardinality 113, then inequalities (4.4)-(4.11) must hold tight.

Moreover, we observed two other facts. First, the cell V_0 (resp. V_1) contains

exactly one vertex that is complete respectively to the vertices in cells V_3 (resp. V_2). Since the subgraphs induced by V_2 (resp. V_3) is a clique, that means that the single vertex in V_0 (resp. V_1) is a simplicial vertex and therefore we may assume without loss of generality that a maximum stable set of 1zc1024 picks the single vertex in V_0 (resp. V_1). (This fact can alternatively be proven by Lemma 2.5.2). It follows that $y_0 = y_1 = 0$ and $y_2 = y_3 = 0$. Therefore, the inequality

$$y_0 + y_1 + y_2 + y_3 + y_4 + y_5 \leq 12$$

that has to hold tight now reads

$$y_4 + y_5 \leq 10$$

and, given inequalities (4.4)-(4.5), the following inequalities have to hold tight:

$$y_4 \leq 5; y_5 \leq 5$$

Our second remarks exploits the structure of V_4. The vertices of V_4 correspond to binary strings of length 10 with exactly 2 "1". It is easy to see that each maximum stable set of V_4 has size 5, and that, for each pair of maximum stable sets of V_4, there is an automorphism mapping one into the other. Therefore, we may arbitrarily choose a stable set of V_4, call it S_4, to satisfy $y_4 \leq 5$ tight.

Now let $N(S_4)$ be the neighborhood of S_4 in V_7, and let $\tilde{G}$ be the subgraph of 1zc1024 induced by:

$$V_7 \cup V_8 \cup V_{10} \setminus N(S_4).$$

Following the previous discussion, if there exists a stable set S of cardinality 113, then:

$$|S \cap (V_7 \cup V_8) \setminus N(S_4)| = 35 \tag{4.12}$$

$$|S \cap (V_7 \cup V_8 \cup A_{10}) \setminus N(S_4)| = 66 \tag{4.13}$$

Finally, let $w : V(\tilde{G}) \to \{1, 2\}$ be a weight function such that:

$$w(u) := \begin{cases} 1 & \forall u \in V_7 \cup V_8 \setminus N(S_4) \\ 2 & \forall u \in V_{10}. \end{cases}$$

Following (4.12) and (4.13), $w(S) = 101$. However, solving by CPLEX the maximum weighted stable set problem for $(\tilde{G}, w)$ (using the equitable partition formulation, in the original space, corresponding to the partition $\{V_7 \setminus N(S_4), V_8, V_{10}\}$ of $V(\tilde{G})$), we obtain an optimal solution of value 100. Then, we can conclude that

1zc1024 does not contain any stable set of cardinality 113. □

4.6 Mann-graphs

Mann-graphs belong to the Dimacs [Dim92] benchmark set and correspond to stable set formulations of the Steiner Triple Problem, translated from the set covering formulations by Mannino [MS95]. In this section we compare our method to orbital branching on this class of graphs.

4.6.1 Origin of mann-instances

The Steiner triple instances are set covering formulations of Steiner Triple Systems, that are well-known as *sts*-instances. `sts9`, `sts15`, `sts27` and `sts45` have respectively 9, 15, 27 and 45 variables, and they were introduced in [FNT74]. The same authors were able to solve `sts9`, `sts15` and `sts27`. Five years later, the optimality of `sts45` was proved by Ratliff [FR89]. New instances `sts81`, `sts135` and `sts243`, of respectively 81, 135 and 243 variables, were introduced in [FR89, MS95]. The optimality of `sts81` was proved in [MS95], while biggest instances remained unsolved. Recently, optimal solutions for `sts135` and `sts243` have been published in [OLRS11b].

Before introducing mann-graphs, we need more details about sts-instances. Given a set X of cardinality n, a Steiner Triple System on X is a collection B of m sets T_i with 3 elements (*triples*) such that for any pair of distinct elements $x, y \in X$, x and y belong to exactly one triple $T_k \in B$. A Steiner Triple System for a set X can be represented by a binary matrix $A^n \in \{0,1\}^{n \times m}$ that has a column for every element in X and a row for every triple in B. Each entry a_{ij^n} of A^n is equal to 1 if and only if element $j \in X$ belongs to triple $T_i \in B$. The following matrix A^9 is the

set covering matrix of the Steiner triple system on 9 elements, well-known as `sts9`:

$$A^9 = \left[\begin{array}{ccc|ccc|ccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ \hline 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}\right]$$

All other sts-matrices are generated by a tripling procedure on Steiner triple system on n elements, for instance `sts27` is obtained by tripling `sts9`. Given a matrix $A_n \in \{0,1\}^{m\times n}$ which corresponds to a Steiner triple system, this procedure generates the following matrix:

$$A^{3n} = \begin{pmatrix} A^n & & \\ & A^n & \\ & & A^n \\ I & I & I \\ D_1 & D_2 & D_3 \end{pmatrix}$$

where I is the identity matrix and $D_k \in \{0,1\}^{3m\times n}$ has exactly an element equal to 1 for each row.

Now, we report the transformation from set covering problem to stable set problem, introduced in [MS95], that allows to obtain stable set instances, i.e. mann-graphs, from corresponding sts-instances. Given a sts-matrix $A^n \in \{0,1\}^{m\times n}$, let $G^n = (V, E)$ be the graph such that:

- G^n has a vertex $\{u_j\} \in V$ for each $j \in \{1,\dots,n\}$;
- G^n has a vertex $u_{ij} \in V$ for each $i \in \{1,\dots,m\}, j \in \{1,\dots,n\}$ such that $a^n_{ij} = 1$;
- G^n has an edge $\{v_{ij}, v_{ik}\} \in E$ for each $i \in \{1,\dots,m\}$ and $j,k \in \{1,\dots,n\}$ such that $a^n_{ij} = a^n_{ik} = 1$;
- G has an edge $\{u_j, v_{hj}\} \in E$ for each $h \in \{1,\dots,m\}, j \in \{1,\dots,n\}$ such that $a^n_{hj} = 1$.

G	$\|V\|$	$\|E\|$	$\alpha(G)$
mann_a9	45	72	16
mann_a27	378	302	126
mann_a45	1035	1980	345
mann_a81	3321	6480	1100

Table 4.3: Optimal solutions for mann-graphs

From the structure of G^n, we get $|V| = n + 3 * m$. Each triple of vertices $T_i = \{v_{ix}, v_{iy}, v_{iz}\} \subset V$, that respectively correspond to entries $a^n_{ix} = a^n_{iy} = a^n_{iz} = 1$ of each row i of A^n, forms clique in G^n, i.e. a triangle. Moreover, each vertex $a_{ij} \in T_i$ is adjacent to u_j. Then, if follows that $|E| = 6 * m$. For instance, graph G^9, well-known as mann_a9, has 45 vertices such that 72 edges. The authors of [MS95] show that the transformation from Steiner triples to stable set instances guarantees the following relation:

$$\alpha(G^n) + z^*(A^n) = n + m$$

where $z^*_{A^n}$ denotes the cardinality of a minimum cover of matrix A^n. Then, the stability numbers of all mann-graphs are determined by optimal solutions found for corresponding sts-instances. Table 4.3 shows optimal solution for each mann-graph considered in this work.

Mann-graphs are hard instances of the maximum stable set problem, even though their optimal solutions are known. In literature, they are frequently used for testing quality of heuristic algorithms, since it is even difficult to find good feasible solutions in practice. Moreover, state-of-art exact methods are able to solve only the graphs of the class which are strictly smaller than mann_a81. In the following, we will see that the equitable partition formulation is one of the main ingredient to solve mann_a81 via maximum stable set problem.

4.6.2 Equitable partitions of on mann-graphs

In the following experiments, we have generated inequalities from equitable partitions finer than the coarsest equitable partitions. The structures of coarsest equitable partitions are very simple and associated EP-graphs, shown in Fig. 4.3, are the same for all mann-graphs. However, coarsest equitable partitions do not help to obtain strengthening formulations. For this reason, we have looked for finer equitable partitions. For each coarsest equitable partition, we have split the cell with minimum cardinality in two subcells such that one of them contains exactly one vertex. Then, we have obtained finer equitable partitions by executing the coarsest equitable refinement procedure. Again, these finer equitable partitions admit a common EP-graph, shown in Fig. 4.4.

Figure 4.3: EP-graphs associated to coarsest equitable partition of all mann-graphs

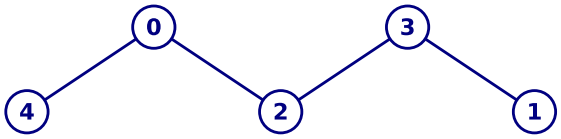


Figure 4.4: EP-graphs associated to a non-coarsest equitable partition of all mann-graphs

For mann_a9, mann_a27 and mann_a45, equitable partition formulations derived from EP-graph shown in Fig. 4.4 can be computed very efficiently by using CPLEX with standard settings. However, this fact does not happen for mann_a81, whose equitable partition inequalities require a remarkable computational effort. Table 4.4 reports the number of nodes ("#Nodes") and the computational time for computing the equitable partition formulations discussed, within a time-limit of 3600 seconds.

Moreover, we observed that orbital branching does not allow to improve the performance of CPLEX shown in Table 4.4. For this reason, we investigated further non-coarsest equitable partitions of mann_a81. Our first attempt was that of finding equitable partitions finer than the equitable partition associated to EP-graph shown in Fig. 4.4. This strategy turned out to be unsuccessful since new equitable partitions did not allow to obtain good formulation for mann_a81. Subsequently, we tried to derive equitable partitions finer than the coarsest, by splitting its cell of minimum cardinality into more than two cells. After several attempts, we found an equitable partition with a regular structure, whose EP-graph is depicted in Fig. 4.5.

Using CPLEX with standard setting, the equitable partition formulation associated to EP-graph in Fig. 4.5 is computed in 8.12 seconds and requires 5956 nodes. Moreover, we will see that it allows to obtain a strong upper bound and certify the optimality for mann_a81.

	#Nodes	tot time
mann_a9	0	0.03
mann_a27	1062	1.55
mann_a45	113003	111.76
mann_a81	647112	t-limit

Table 4.4: Generating equitable partition formulations associated to Fig. 4.4

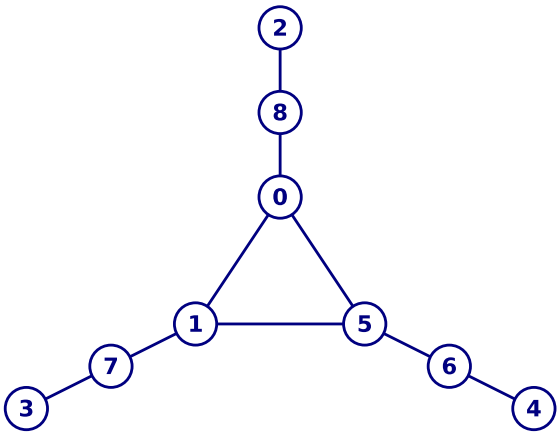


Figure 4.5: A non-coarsest equitable partition of mann_a81

	$\alpha(G)$	$QSTAB$	ϑ	EQP_AS
mann_a9	16	18	17.47	17
mann_a27	126	135	132.76	127
mann_a45	345	360	356.04	346
mann_a81	1100	1134	1129.43	1107

Table 4.5: Comparing upper bounds for mann-graphs

Computational experiments on mann-graphs

In experiments, we have computed right-hand sides of inequalities by using CPLEX with standard settings. Parameters of equitable partition used for each graph are reported in Subsection 4.6.3.

Moreover, we have imposed an overall time-limit of 3600 seconds for generating inequalities and solving equitable partition formulations. Table A.1 shows computational performance for generating inequalities.

Table 4.5 compares upper bounds given by the aggregate equitable partition formulation (column "EQP_AS") with upper bounds obtained by the clique relaxation (column "$QSTAB$") and the semidefinite relaxation (column "ϑ"). Aggregate equitable partition formulations provide excellent upper bounds: we obtain an absolute gap of 1 unit for mann_a9, mann_a27 and mann_a45, while the gap for mann_a81 is drastically reduced.

Table 4.6 compares the computational performance of CPLEX (with standard settings) solving the equitable partition formulation in the original space (column "CPLEX+EQP_OS") with the performance of orbital branching. Fields "gap%", "#Nodes", "tot time" and "`Nauty` time" corresponds respectively to relative gap between upper bound and lower bound at the end of optimization, the number of

	Orbital Branching				CPLEX + EQP_OS		
	gap%	#Nodes	tot time	`Nauty` time	gap%	#Nodes	tot time
mann_a9	0	0	0.05		0	0	0.05
mann_a27	2.91	961965	t-limit	2597.24	0	1062	2.67
mann_a45	3.29	115157	t-limit	2753.42	0	126901	130.1
mann_a81	2.84	15686	t-limit	2475.53	0.64	478989	t-limit

Table 4.6: Comparing equitable partition formulation with orbital branching on mann-graphs

node generated during the branch&bound process, the total time in seconds of optimization, the overall time in seconds that `Nauty` have used for computing orbits. Let us observe that equitable partition formulations for mann_a27 and mann_a45 allow CPLEX to outperform orbital branching: the computation of optimal solutions is carried out very efficiently. On the contrary, orbital branching does not obtain optimal solutions within the time-limit for mann_a27, mann_a45. The bad performance of orbital branching is mostly due to the computation of the orbits, that takes about 70% of total time. We also see that both methods are not able to solve mann_a81 within the time limit. However, equitable partition formulation achieves the best relative gap at the end of the optimization for this graph, founding an incumbent solution of value 1100 after less than 6 seconds from the start.

4.6.3 Equitable partition parameters of mann-graphs

mann_a9
$$\begin{pmatrix} 0 & 0 & 2 & 0 & 1 \\ 0 & 2 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 3 & 1 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 4 \\ 24 \\ 8 \\ 8 \\ 1 \end{pmatrix}$$

mann_a27
$$\begin{pmatrix} 0 & 0 & 2 & 0 & 1 \\ 0 & 2 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 12 & 1 & 0 & 0 \\ 13 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 13 \\ 312 \\ 26 \\ 26 \\ 1 \end{pmatrix}$$

mann_a45
$$\begin{pmatrix} 0 & 0 & 2 & 0 & 1 \\ 0 & 2 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 21 & 1 & 0 & 0 \\ 22 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 22 \\ 924 \\ 44 \\ 44 \\ 1 \end{pmatrix}$$

$$\text{mann\_a81} \quad \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 13 & 27 & 0 & 0 & 0 \\ 0 & 27 & 0 & 13 & 0 & 0 & 0 & 0 & 0 \\ 27 & 0 & 13 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 729 \\ 729 \\ 351 \\ 351 \\ 351 \\ 729 \\ 27 \\ 27 \\ 27 \end{pmatrix}$$

4.7 Solving mann_a81 to optimality

Table 4.5 shows that aggregate equitable partition formulations provide upper bounds stronger than the best upper bounds known so far: in particular, for mann_a81, the gap between the upper bound provided by the aggregate formulation and the optimal solution is equal to 7.

However, a closer look to the EP-graph associated to the equitable partition $\mathcal{P}$ shown in Fig. 4.5 and some more computation allowed us to enhance the equitable partition formulation associated to $\mathcal{P}$ and close the gap so that CPLEX could solve the problem to optimality. We sketch our approach in the following.

Let us consider the equitable partition $\mathcal{P} = \{V_0, \ldots, V_8\}$ of mann_a81 whose EP-graph is shown in Fig. 4.5.

The corresponding aggregate equitable partition formulation has the following

inequalities:

$$\max \quad \sum_{i=0}^{8} y_i \tag{4.14}$$

$$\text{subject to} \tag{4.15}$$

$$y_0 \leq 729 \tag{4.16}$$

$$y_1 \leq 729 \tag{4.17}$$

$$y_2 \leq 117 \tag{4.18}$$

$$y_3 \leq 117 \tag{4.19}$$

$$y_4 \leq 117 \tag{4.20}$$

$$y_5 \leq 729 \tag{4.21}$$

$$y_6 \leq 27 \tag{4.22}$$

$$y_7 \leq 27 \tag{4.23}$$

$$y_8 \leq 27 \tag{4.24}$$

$$y_0 + y_1 \leq 729 \tag{4.25}$$

$$y_0 + y_5 \leq 729 \tag{4.26}$$

$$y_0 + y_8 \leq 729 \tag{4.27}$$

$$y_1 + y_5 \leq 729 \tag{4.28}$$

$$y_1 + y_7 \leq 729 \tag{4.29}$$

$$y_2 + y_8 \leq 126 \tag{4.30}$$

$$y_3 + y_7 \leq 126 \tag{4.31}$$

$$y_4 + y_6 \leq 126 \tag{4.32}$$

$$y_5 + y_6 \leq 729 \tag{4.33}$$

$$y_0 + y_1 + y_5 + y_8 \leq 756 \tag{4.34}$$

$$y_0 + y_1 + y_5 + y_7 \leq 756 \tag{4.35}$$

$$y_0 + y_1 + y_5 + y_6 \leq 756 \tag{4.36}$$

$$y_4 + y_5 + y_6 \leq 846 \tag{4.37}$$

$$y_1 + y_3 + y_7 \leq 846 \tag{4.38}$$

$$y_0 + y_2 + y_8 \leq 846 \tag{4.39}$$

$$y_0 + y_1 + y_5 \leq 729 \tag{4.40}$$

Note that $\alpha(G[V_0 \cup V_1 \cup V_5 \cup V_4 \cup V_6]) \leq \alpha(G[V_0 \cup V_1 \cup V_5]) + \alpha(G[V_4 \cup V_6]) = 729 + 126 = 855$.

Now let S be a maximum stable set. Now first suppose that $|S \cap (V_0 \cup V_1 \cup V_5 \cup$

$V_4 \cup V_6)| \leq 846$. In this case, $|S| \leq |S \cap (V_0 \cup V_1 \cup V_5 \cup V_4 \cup V_6)| + |S \cap (V_2 \cup V_8)| + |S \cap (V_3 \cup V_7)| \leq 846 + 126 + 126 = 1098$. By the same argument, one may show that, if either $|S \cap (V_0 \cup V_1 \cup V_5 \cup V_2 \cup V_8)| \leq 846$ or $|S \cap (V_0 \cup V_1 \cup V_5 \cup V_3 \cup V_7)| \leq 846$, then $|S| \leq 1100$.

We may therefore assume add to the equitable formulation corresponding to $\mathcal{P}$ (in the original space) the following inequalities: $\sum_{v \in V_0 \cup V_1 \cup V_4 \cup V_5 \cup V_6} x_v \geq 847$; $\sum_{v \in V_0 \cup V_1 \cup V_3 \cup V_5 \cup V_7} x_v \geq 847$; $\sum_{v \in V_0 \cup V_1 \cup V_2 \cup V_5 \cup V_8} x_v \geq 847$. In this case, CPLEX is able to close the computation finding, again, an optimal solution of value 1100. Then, we can conclude that the for mann_a81 the maximum stable set has cardinality 1100.

4.8 Keller-instances

In this section we compare our method to orbital branching on Keller-graphs.

4.8.1 Origin of Keller-graphs

The origin of Keller-graphs is related to Keller's cube-tiling conjectures. A tiling of $\mathbb{R}^n$ by unit cubes is a set of unit cubes such that every point in R^n is covered by one of the cubes, and such that the interiors of no two cubes overlap. The Keller's conjecture was introduced in 1930 as a generalization of Minkowski's conjecture.

Conjecture 4.8.1 (Keller)**.** *Every tiling of R^n by unit cubes contains two cubes that meet in an $n-1$ dimensional face.*

Ten years later, Perron proved that the Keller's conjecture is true in six and fewer dimensions [Per40]. In the subsequent years, interest in Keller's conjecture decreased since Minkoski's conjecture was proved by Hajós. However, in 1990 new motivations arose for the reformulation of Keller's conjecture as a combinatorial problem. The author of [CS90] stated that there is a counterexample for this conjecture if and only if the n-dimensional Keller-graph has a clique of size 2^n. Given the set of all strings $\{0,1,2,3\}^n$, the n-dimensional Keller-graph $G^n = (V, E)$ is such that:

- each vertex $u \in V$ is in bijection with a string $u \in \{0,1,2,3\}^n$;
- $u, v \in E$ if there exist two distinct indices $i, j \in \{1, \ldots, n\}$ such that $|u_i - v_i| = 1$ and $u_j \neq v_j$.

In 1992, the author of [GLS92] state that the keller's conjecture is false in dimensions greater or equal than ten. In 2002, this result was extended to dimension greater or equal than eight [Mac02]. Finally, in 2011, the result published in

G	$\|V\|$	$\|E\|$	$\alpha(G)$
keller4	171	5100	11
keller5	776	74710	27

Table 4.7: Optimal solutions for keller-graphs

[DEL+11] shows that the Keller's conjecture is true in seven dimension. All these recent results are based on counterexamples that have been found by exploiting theoretical property of Keller-graphs in order to build up suitable enumeration schemes. Table 4.7 shows optimal solution for keller-graphs considered in this work.

4.8.2 Experiments on Keller-graphs

In this section, we discuss computational experiments on Keller-graphs. In the following experiments, we have generated equitable partition inequalities from coarsest equitable partitions of keller-graphs. We have observed that structures of coarsest equitable partitions are quite complicated. Moreover, for each of these graphs, generating finer equitable partitions than the coarsest does not help to obtain improving upper bounds or higher computational efficiency. In Fig. 4.6 we show EP-graphs on which we have based the computation of equitable partition inequalities. In Section 4.8.3 we report parameters of equitable partitions used.

For this class of graphs, orbital branching allows to compute right-hand sides more efficiently than CPLEX with symmetry reduction settings. Moreover, we have imposed an overall time-limit of 10800 seconds for generating inequalities and solving equitable partition formulations. Table A.2 shows computational performance for generating inequalities.

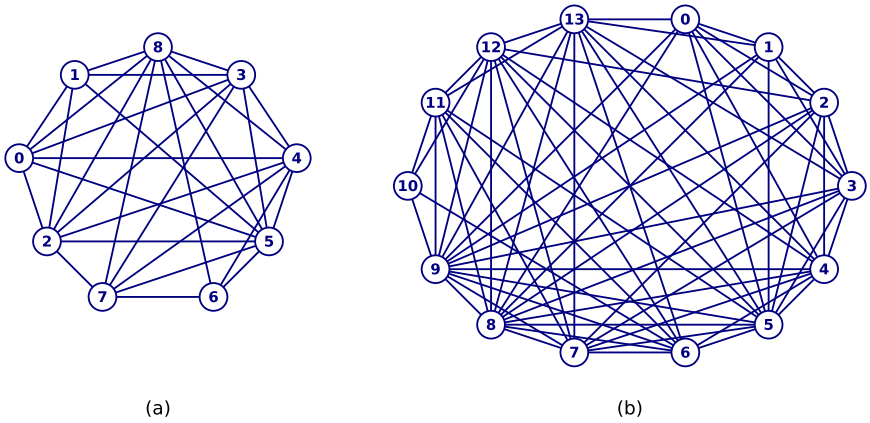


Figure 4.6: EP-graphs associated to coarsest equitable partitions of keller4 (a) and keller5 (b)

	$\alpha(G)$	$QSTAB$	ϑ	EQP_AS
keller4	11	14.82	14.01	11
keller5	27	31	31	28

Table 4.8: Comparing upper bounds for keller-graphs

	OB			CPLEX		
	gap%	#Nodes	tot time	gap%	#Nodes	tot time
keller4	0	339	23.05	0	710	27.52
keller5	0	191180	4233.52	0	1465598	8145.91

Table 4.9: Comparing CPLEX with orbital branching for generating equitable partition formulation on keller-graphs

Table 4.8 compares upper bounds given by the aggregate equitable partition formulation (column "EQP_AS") with upper bounds obtained by the clique relaxation (column "$QSTAB$") and the semidefinite relaxation (column "ϑ"). Let us note that aggregate equitable partition formulations provide excellent upper bounds for keller graphs. In particular, the upper bound is tight for keller4 and it forms a 1 unit gap for keller5.

We report a remarkable fact. Table 4.9 shows computational results that confirm the potentiality of our inequalities generator to exploit symmetry-braking methods for the computation of right-hand sides. Performance obtained for computing and solving equitable partition formulation by orbital branching (column "OB") is not comparable to performance achieved by CPLEX with symmetry reduction settings (column "CPLEX"): the number of nodes generated by CPLEX and orbital branching differs by one order of magnitude and the total time required by CPLEX is almost the double of time needed by orbital branching.

Table 4.10 compares the computational performance of generating and solving the equitable partition formulation in the original space by orbital branching (column "OB+EQP_OS") with the performance of orbital branching. Fields "gap%", "#Nodes", "tot time" and "`Nauty` time" correspond respectively to relative gap between upper bound and lower bound at the end of optimization, the number of node generated during the branch&bound process, the total time in seconds of optimization, the overall time in seconds that `Nauty` have used for computing orbits. We observe that both equitable partition formulation and orbital branching allow to obtain optimal solutions within the time limit. However, orbital branching turn out to be very efficient for keller-graphs.

	Orbital Branching				OB + EQP_OS		
	gap%	#Nodes	tot time	`Nauty` time	gap%	#Nodes	tot time
keller4	0	35	2.7	0	0	339	23.05
keller5	0	12670	919.3	52.99	0	191180	4233.52

Table 4.10: Comparing equitable partition formulation with orbital branching on keller-graphs

4.8.3 Equitable partition parameters of Keller-graphs

keller4

$$\begin{pmatrix} 0 & 4 & 16 & 4 & 2 & 16 & 0 & 0 & 4 \\ 1 & 7 & 16 & 6 & 0 & 12 & 0 & 0 & 4 \\ 2 & 8 & 18 & 8 & 1 & 10 & 0 & 2 & 9 \\ 1 & 6 & 16 & 7 & 1 & 16 & 0 & 3 & 10 \\ 3 & 0 & 12 & 6 & 0 & 24 & 1 & 2 & 12 \\ 3 & 9 & 15 & 12 & 3 & 9 & 1 & 5 & 9 \\ 0 & 0 & 0 & 0 & 4 & 32 & 0 & 8 & 24 \\ 0 & 0 & 12 & 9 & 1 & 20 & 1 & 7 & 18 \\ 1 & 4 & 18 & 10 & 2 & 12 & 1 & 6 & 14 \end{pmatrix} \begin{pmatrix} 6 \\ 24 \\ 48 \\ 24 \\ 4 \\ 32 \\ 1 \\ 8 \\ 24 \end{pmatrix}$$

keller5

$$\begin{pmatrix} 0 & 4 & 24 & 6 & 3 & 48 & 0 & 0 & 12 & 32 & 0 & 0 & 0 & 8 \\ 1 & 9 & 30 & 9 & 0 & 44 & 0 & 0 & 12 & 24 & 0 & 0 & 0 & 8 \\ 2 & 10 & 34 & 12 & 1 & 44 & 0 & 4 & 26 & 20 & 0 & 0 & 4 & 16 \\ 1 & 6 & 24 & 9 & 1 & 48 & 0 & 6 & 24 & 32 & 0 & 0 & 4 & 20 \\ 3 & 0 & 12 & 6 & 0 & 48 & 2 & 4 & 24 & 48 & 0 & 0 & 4 & 24 \\ 3 & 11 & 33 & 18 & 3 & 39 & 1 & 11 & 33 & 14 & 0 & 2 & 10 & 19 \\ 0 & 0 & 0 & 0 & 4 & 32 & 0 & 8 & 24 & 64 & 1 & 2 & 16 & 48 \\ 0 & 0 & 12 & 9 & 1 & 44 & 1 & 9 & 30 & 40 & 0 & 3 & 14 & 36 \\ 1 & 4 & 26 & 12 & 2 & 44 & 1 & 10 & 34 & 24 & 0 & 2 & 13 & 26 \\ 4 & 12 & 30 & 24 & 6 & 28 & 4 & 20 & 36 & 12 & 1 & 6 & 14 & 16 \\ 0 & 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 & 80 & 0 & 10 & 40 & 80 \\ 0 & 0 & 0 & 0 & 0 & 32 & 1 & 12 & 24 & 48 & 1 & 9 & 32 & 56 \\ 0 & 0 & 12 & 6 & 1 & 40 & 2 & 14 & 39 & 28 & 1 & 8 & 26 & 38 \\ 1 & 4 & 24 & 15 & 3 & 38 & 3 & 18 & 39 & 16 & 1 & 7 & 19 & 27 \end{pmatrix} \begin{pmatrix} 10 \\ 40 \\ 120 \\ 60 \\ 10 \\ 160 \\ 5 \\ 40 \\ 120 \\ 80 \\ 1 \\ 10 \\ 40 \\ 80 \end{pmatrix}$$

Chapter 5

Conclusions and future work

Hard problems represent a significant challenge since they cannot be simply solved by using a huge quantity of computational resources. In the case of maximum stable set problem, even small instances with a few hundred of vertices could be intractable (to optimality) for state-of-art solvers. Sometimes, hard instances contain symmetry, which is not just artificially generated in order to test methods, but also it naturally arises from important problems and applications. Most part of literature uses powerful tools, but sometimes computationally prohibitive, that exploit information of symmetry with the purpose of avoiding its bad effects on optimization methods. However, properties of symmetry imply regular structures that, sometimes, could be very simple. EP-graphs might be crucial to deal with hard instances. In some cases, it allows to detect "local symmetries" and therefore interesting subproblems to solve.

We have exploited potentiality of the EP-graph to strengthen formulations of the maximum stable set problem for hard and highly symmetric instances. Our method derives from its structure, that is often simple in this case, equitable partition inequalities that are derived from subgraphs, sometimes very tractable. In particular, we have also seen that their stability numbers can be computed in a very effective way by orbital branching. Moreover, EP-graph allows to perform a parallel computing of inequalities.

Equitable partition formulations constitute an effective tool for solving some hard instances. In same cases, they are generated very fast and allow an impressive performance, as we have observed for mann-graphs. Aggregate equitable partition formulations often provide excellent bounds. They have also been the key to improve best known upper bounds for some 1zc-graphs, and certify the optimality for 1zc1024, whose stability number has been unknown so far. Furthermore, equitable partition inequalities have allowed us to solve to optimality a stable set formulation of mann_a81.

This work highlights three important open questions. The first is based on the potentiality of EP-graph. Throughout the thesis, we have used EP-graph to strengthen formulation of the maximum stable set problem. Clearly, this is not the unique way to exploit it: EP-graph contains information that we have not considered in our work, for instance the regularity of degrees among classes of equitable partition. In future research we will investigate approaches that could exploit EP-graph in branching or fixing strategies.

The second open question concerns combinatorial approaches based on the structure of EP-graph. We have seen that some hard instances are characterized by a tree-structure. This feature suggests to design combinatorial algorithms, e.g. dynamic programming, which can exploit the simple structure of EP-graph and the properties arising from equitable partition, in order to build up an enumeration scheme, possibly effective in practice.

The third open question refers to the generation of equitable partitions. Inequalities derived from non-coarsest equitable partitions are often stronger than inequalities associated to coarsest equitable partition. We have proposed some rules, based on computational experience, that allows to generate finer equitable partitions with few classes, in some cases. Then, it would be helpful to investigate how to obtain improving equitable partitions that, for instance, provide simple EP-graphs.

Appendix A

Computing equitable partition inequalities

The following tables show a performance profile of the generated equitable partitions inequalities for each class of graphs discussed in Section 4.3. Inequalities are labelled with respect of syntax `<graph>_EQP_<type>_<id>`, where:

`graph` refers to the name of the considered graph;

`type` denotes the category of the equitable partition inequality. Possible values are `v`,`e`, `t`, `n` that respectively identify *vertex*, *edge*, *triangle*, *closed neighborhood* equitable partition inequalities;

`id` is the identifier of the inequality, according to the representation of EP-graphs showed in Sections 4.4, 4.6, 4.8. Possible values are a single indices, pair and triple of indices which respectively refer to vertices, edges and triangles of the corresponding EP-graph.

For each inequality, fields "B&B Nodes" and "Total time" respectively show the number of nodes generated during branch&bound and the total time in seconds for computing equitable partitions inequalities. Field "`Nauty` time" appears when have used orbital branching for generating inequalities, then it shows the total time in second required by `Nauty` to compute orbits.

Table A.1: Computing right-hand sides for mann-graphs

Inequality	**B&B Nodes**	**Total time**	**`Nauty` time**
mann_a9_EQP_v_0	0	0	
mann_a9_EQP_v_1	0	0	

Table A.1: Continued on next page

Table A.1: Continued from previous page

Inequality	**B&B Nodes**	**Total time**	`Nauty` **time**
mann_a9_EQP_v_2	0	0	
mann_a9_EQP_v_3	0	0	
mann_a9_EQP_v_4	0	0	
mann_a9_EQP_e_(0,2)	0	0	
mann_a9_EQP_e_(0,4)	0	0	
mann_a9_EQP_e_(1,3)	0	0.01	
mann_a9_EQP_e_(2,3)	0	0	
mann_a9_EQP_n_0	0	0	
mann_a9_EQP_n_2	0	0	
mann_a9_EQP_n_3	0	0.02	
mann_a27_EQP_v_0	0	0	
mann_a27_EQP_v_1	0	0	
mann_a27_EQP_v_2	0	0	
mann_a27_EQP_v_3	0	0	
mann_a27_EQP_v_4	0	0	
mann_a27_EQP_e_(0,4)	0	0	
mann_a27_EQP_e_(1,3)	753	1.25	
mann_a27_EQP_e_(2,3)	0	0	
mann_a27_EQP_n_0	0	0	
mann_a27_EQP_n_2	0	0	
mann_a27_EQP_n_3	309	0.3	
mann_a45_EQP_v_0	0	0	
mann_a45_EQP_v_1	0	0	
mann_a45_EQP_v_2	0	0	
mann_a45_EQP_v_3	0	0	
mann_a45_EQP_v_4	0	0	
mann_a45_EQP_e_(0,2)	0	0	
mann_a45_EQP_e_(0,4)	0	0	
mann_a45_EQP_e_(1,3)	91725	79.33	
mann_a45_EQP_e_(2,3)	0	0	
mann_a45_EQP_n_0	0	0	
mann_a45_EQP_n_2	0	0	
mann_a45_EQP_n_3	21278	32.43	
MANN_a81_EQP_v_0	0	0	

Table A.1: Continued on next page

Table A.1: Continued from previous page

Inequality	**B&B Nodes**	**Total time**	`Nauty` **time**
MANN_a81_EQP_v_1	0	0	
MANN_a81_EQP_v_2	0	0	
MANN_a81_EQP_v_3	0	0	
MANN_a81_EQP_v_4	0	0	
MANN_a81_EQP_v_5	0	0	
MANN_a81_EQP_v_6	0	0	
MANN_a81_EQP_v_7	0	0	
MANN_a81_EQP_v_8	0	0	
MANN_a81_EQP_e_(0,1)	0	0	
MANN_a81_EQP_e_(0,5)	0	0	
MANN_a81_EQP_e_(0,8)	0	0	
MANN_a81_EQP_e_(1,5)	0	0	
MANN_a81_EQP_e_(1,7)	0	0	
MANN_a81_EQP_e_(2,8)	376	3.26	
MANN_a81_EQP_e_(3,7)	2790	2.59	
MANN_a81_EQP_e_(4,6)	2790	2.27	
MANN_a81_EQP_e_(5,6)	0	0	
MANN_a81_EQP_n_0	0	0	
MANN_a81_EQP_n_1	0	0	
MANN_a81_EQP_n_5	0	0	
MANN_a81_EQP_n_6	0	0	
MANN_a81_EQP_n_7	0	0	
MANN_a81_EQP_n_8	0	0	
MANN_a81_EQP_t_(0,1,5)	0	0	

Table A.2: Computing right-hand sides for Keller-graphs

Inequality	**B&B Nodes**	**Total time**
keller4_EQP_v_0	0	0
keller4_EQP_v_1	0	0
keller4_EQP_v_2	0	0.23
keller4_EQP_v_3	0	0.01
keller4_EQP_v_4	0	0
keller4_EQP_v_5	0	0

Table A.2: Continued on next page

Table A.2: Continued from previous page

Inequality	**B&B Nodes**	**Total time**
keller4_EQP_v_6	0	0
keller4_EQP_v_7	0	0
keller4_EQP_v_8	0	0.01
keller4_EQP_e_(0,1)	0	0
keller4_EQP_e_(0,2)	0	0.76
keller4_EQP_e_(0,3)	0	0
keller4_EQP_e_(0,4)	0	0
keller4_EQP_e_(0,5)	0	0
keller4_EQP_e_(0,8)	0	0.01
keller4_EQP_e_(1,2)	15	0.45
keller4_EQP_e_(1,3)	0	0
keller4_EQP_e_(1,5)	0	0.23
keller4_EQP_e_(1,8)	0	0.02
keller4_EQP_e_(2,3)	3	0.53
keller4_EQP_e_(2,4)	3	0.05
keller4_EQP_e_(2,5)	33	1.04
keller4_EQP_e_(2,7)	0	0.01
keller4_EQP_e_(2,8)	7	0.43
keller4_EQP_e_(3,4)	0	0
keller4_EQP_e_(3,5)	0	0.1
keller4_EQP_e_(3,7)	0	0
keller4_EQP_e_(3,8)	0	0.28
keller4_EQP_e_(4,5)	0	0
keller4_EQP_e_(4,6)	0	0
keller4_EQP_e_(4,7)	0	0
keller4_EQP_e_(4,8)	0	0
keller4_EQP_e_(5,6)	0	0
keller4_EQP_e_(5,7)	0	0
keller4_EQP_e_(5,8)	0	0
keller4_EQP_e_(6,7)	0	0
keller4_EQP_e_(6,8)	0	0
keller4_EQP_e_(7,8)	0	0.01
keller4_EQP_n_0	59	2.61
keller4_EQP_n_1	39	3.08
keller4_EQP_n_2	49	3.8

Table A.2: Continued on next page

Table A.2: Continued from previous page

Inequality	**B&B Nodes**	**Total time**
keller4_EQP_n_3	49	3.95
keller4_EQP_n_4	99	3.25
keller4_EQP_n_6	0	0.02
keller4_EQP_n_7	43	2.17
keller4_EQP_t_(0,1,2)	5	0.34
keller4_EQP_t_(0,1,3)	0	0
keller4_EQP_t_(0,1,5)	0	0.58
keller4_EQP_t_(0,1,8)	0	0.01
keller4_EQP_t_(0,2,3)	3	0.87
keller4_EQP_t_(0,2,4)	0	0.01
keller4_EQP_t_(0,2,5)	0	1.47
keller4_EQP_t_(0,2,8)	27	0.48
keller4_EQP_t_(0,3,4)	0	0
keller4_EQP_t_(0,3,5)	0	0.25
keller4_EQP_t_(0,3,8)	5	0.05
keller4_EQP_t_(0,4,5)	0	0
keller4_EQP_t_(0,4,8)	0	0
keller4_EQP_t_(0,5,8)	0	0.03
keller4_EQP_t_(1,2,3)	17	0.95
keller4_EQP_t_(1,2,5)	5	0.5
keller4_EQP_t_(1,2,8)	59	0.82
keller4_EQP_t_(1,3,5)	3	1.15
keller4_EQP_t_(1,3,8)	0	0.03
keller4_EQP_t_(1,5,8)	15	0.42
keller4_EQP_t_(2,3,4)	0	0.46
keller4_EQP_t_(2,3,5)	15	1.13
keller4_EQP_t_(2,3,7)	13	0.11
keller4_EQP_t_(2,3,8)	41	0.68
keller4_EQP_t_(2,4,5)	0	1
keller4_EQP_t_(2,4,7)	5	0.13
keller4_EQP_t_(2,4,8)	11	0.83
keller4_EQP_t_(2,5,7)	7	0.39
keller4_EQP_t_(2,5,8)	9	1.09
keller4_EQP_t_(2,7,8)	17	0.27
keller4_EQP_t_(3,4,5)	0	0.19

Table A.2: Continued on next page

Table A.2: Continued from previous page

Inequality	**B&B Nodes**	**Total time**
keller4_EQP_t_(3,4,7)	0	0
keller4_EQP_t_(3,4,8)	7	0.1
keller4_EQP_t_(3,5,7)	0	0.09
keller4_EQP_t_(3,5,8)	0	0.62
keller4_EQP_t_(3,7,8)	0	0.07
keller4_EQP_t_(4,5,6)	0	0
keller4_EQP_t_(4,5,7)	0	0.01
keller4_EQP_t_(4,5,8)	0	0.01
keller4_EQP_t_(4,6,7)	0	0
keller4_EQP_t_(4,6,8)	0	0
keller4_EQP_t_(4,7,8)	0	0
keller4_EQP_t_(5,6,7)	0	0
keller4_EQP_t_(5,6,8)	0	0
keller4_EQP_t_(5,7,8)	0	0
keller4_EQP_t_(6,7,8)	0	0
keller5_EQP_v_0	0	0
keller5_EQP_v_1	0	0
keller5_EQP_v_10	0	0
keller5_EQP_v_11	0	0
keller5_EQP_v_12	0	0.02
keller5_EQP_v_13	0	0
keller5_EQP_v_2	27	6.12
keller5_EQP_v_3	0	0
keller5_EQP_v_4	0	0
keller5_EQP_v_5	59	5.98
keller5_EQP_v_6	0	0
keller5_EQP_v_7	0	0
keller5_EQP_v_8	49	5.35
keller5_EQP_v_9	0	0
keller5_EQP_e_(0,1)	0	0
keller5_EQP_e_(0,13)	0	0.53
keller5_EQP_e_(0,2)	79	3.91
keller5_EQP_e_(0,3)	0	0
keller5_EQP_e_(0,4)	0	0
keller5_EQP_e_(0,5)	92	4.25

Table A.2: Continued on next page

Table A.2: Continued from previous page

Inequality	**B&B Nodes**	**Total time**
keller5_EQP_e_(0,8)	22	0.87
keller5_EQP_e_(0,9)	0	0.28
keller5_EQP_e_(1,13)	5	0.1
keller5_EQP_e_(1,2)	123	3.02
keller5_EQP_e_(1,3)	0	0
keller5_EQP_e_(1,5)	471	5.96
keller5_EQP_e_(1,8)	37	0.43
keller5_EQP_e_(1,9)	39	2.76
keller5_EQP_e_(10,11)	0	0
keller5_EQP_e_(10,12)	0	0.01
keller5_EQP_e_(10,13)	0	0.01
keller5_EQP_e_(11,12)	0	0
keller5_EQP_e_(11,13)	0	0.01
keller5_EQP_e_(12,13)	0	0.01
keller5_EQP_e_(2,12)	247	0.8
keller5_EQP_e_(2,13)	3327	5.37
keller5_EQP_e_(2,3)	105	5.18
keller5_EQP_e_(2,4)	18	0.74
keller5_EQP_e_(2,5)	1177	10.71
keller5_EQP_e_(2,7)	11	0.4
keller5_EQP_e_(2,8)	16363	24.82
keller5_EQP_e_(2,9)	1017	8
keller5_EQP_e_(3,12)	0	0.02
keller5_EQP_e_(3,13)	17	4.18
keller5_EQP_e_(3,4)	0	0
keller5_EQP_e_(3,5)	338	6.99
keller5_EQP_e_(3,7)	0	0
keller5_EQP_e_(3,8)	325	3.81
keller5_EQP_e_(3,9)	5	3.64
keller5_EQP_e_(4,12)	0	0.01
keller5_EQP_e_(4,13)	0	0.01
keller5_EQP_e_(4,5)	92	4.95
keller5_EQP_e_(4,6)	0	0
keller5_EQP_e_(4,7)	0	0
keller5_EQP_e_(4,8)	103	4.17

Table A.2: Continued on next page

Table A.2: Continued from previous page

Inequality	**B&B Nodes**	**Total time**
keller5_EQP_e_(4,9)	0	1.25
keller5_EQP_e_(5,11)	113	2.09
keller5_EQP_e_(5,12)	1095	6.38
keller5_EQP_e_(5,13)	645	9.99
keller5_EQP_e_(5,6)	187	3.76
keller5_EQP_e_(5,7)	671	8.38
keller5_EQP_e_(5,8)	1215	10.95
keller5_EQP_e_(5,9)	522	7.53
keller5_EQP_e_(6,10)	0	0
keller5_EQP_e_(6,11)	0	0
keller5_EQP_e_(6,12)	0	0
keller5_EQP_e_(6,13)	0	0.22
keller5_EQP_e_(6,7)	0	0
keller5_EQP_e_(6,8)	33	0.6
keller5_EQP_e_(6,9)	0	0.07
keller5_EQP_e_(7,11)	0	0
keller5_EQP_e_(7,12)	45	0.34
keller5_EQP_e_(7,13)	5	1.66
keller5_EQP_e_(7,8)	213	2.5
keller5_EQP_e_(7,9)	3	0.81
keller5_EQP_e_(8,11)	0	0.01
keller5_EQP_e_(8,12)	121	2.85
keller5_EQP_e_(8,13)	339	4.47
keller5_EQP_e_(8,9)	7	3.26
keller5_EQP_e_(9,10)	0	0
keller5_EQP_e_(9,11)	0	0.02
keller5_EQP_e_(9,12)	0	0.03
keller5_EQP_e_(9,13)	0	0.01
keller5_EQP_n_0	3619	188.71
keller5_EQP_n_1	3856	176.72
keller5_EQP_n_10	0	0.36
keller5_EQP_n_11	5205	73.35
keller5_EQP_n_12	22506	588.01
keller5_EQP_n_2	16830	397.32
keller5_EQP_n_3	16830	403.39

Table A.2: Continued on next page

Table A.2: Continued from previous page

Inequality	**B&B Nodes**	**Total time**
keller5_EQP_n_4	23196	615.36
keller5_EQP_n_5	20723	458.89
keller5_EQP_n_6	5391	64.25
keller5_EQP_n_7	22939	614.3
keller5_EQP_n_8	20723	462.26
keller5_EQP_t_(0,1,13)	2517	2.26
keller5_EQP_t_(0,1,2)	39	2.03
keller5_EQP_t_(0,1,3)	0	0
keller5_EQP_t_(0,1,5)	350	6.43
keller5_EQP_t_(0,1,8)	22	0.75
keller5_EQP_t_(0,1,9)	111	4.08
keller5_EQP_t_(0,2,13)	5581	10.84
keller5_EQP_t_(0,2,3)	99	5.4
keller5_EQP_t_(0,2,4)	13	0.85
keller5_EQP_t_(0,2,5)	529	7.87
keller5_EQP_t_(0,2,8)	3962	13.2
keller5_EQP_t_(0,2,9)	356	7.22
keller5_EQP_t_(0,3,13)	235	0.69
keller5_EQP_t_(0,3,4)	0	0
keller5_EQP_t_(0,3,5)	252	6.83
keller5_EQP_t_(0,3,8)	1389	1.68
keller5_EQP_t_(0,3,9)	22	2
keller5_EQP_t_(0,4,13)	0	1.01
keller5_EQP_t_(0,4,5)	33	5.58
keller5_EQP_t_(0,4,8)	13	0.72
keller5_EQP_t_(0,4,9)	0	0.99
keller5_EQP_t_(0,5,13)	355	9.81
keller5_EQP_t_(0,5,8)	2730	10.67
keller5_EQP_t_(0,5,9)	195	8.39
keller5_EQP_t_(0,8,13)	45	1.35
keller5_EQP_t_(0,8,9)	127	5.27
keller5_EQP_t_(0,9,13)	5	1.05
keller5_EQP_t_(1,2,13)	18951	23.06
keller5_EQP_t_(1,2,3)	6519	12.26
keller5_EQP_t_(1,2,5)	708	12.61

Table A.2: Continued on next page

Table A.2: Continued from previous page

Inequality	**B&B Nodes**	**Total time**
keller5_EQP_t_(1,2,8)	119713	181.22
keller5_EQP_t_(1,2,9)	1475	9.13
keller5_EQP_t_(1,3,13)	55	1.44
keller5_EQP_t_(1,3,5)	1514	10.91
keller5_EQP_t_(1,3,8)	8513	8.57
keller5_EQP_t_(1,3,9)	213	6.9
keller5_EQP_t_(1,5,13)	40829	88.43
keller5_EQP_t_(1,5,8)	17441	48.87
keller5_EQP_t_(1,5,9)	437	9.77
keller5_EQP_t_(1,8,13)	28	1.27
keller5_EQP_t_(1,8,9)	6370	14.49
keller5_EQP_t_(1,9,13)	79	2.49
keller5_EQP_t_(10,11,12)	0	0.01
keller5_EQP_t_(10,11,13)	0	0.01
keller5_EQP_t_(10,12,13)	0	0.01
keller5_EQP_t_(11,12,13)	0	0.01
keller5_EQP_t_(2,12,13)	6129	9.5
keller5_EQP_t_(2,3,12)	773	3.22
keller5_EQP_t_(2,3,13)	10229	21.25
keller5_EQP_t_(2,3,4)	773	3.17
keller5_EQP_t_(2,3,5)	4008	19.55
keller5_EQP_t_(2,3,7)	2191	6.88
keller5_EQP_t_(2,3,8)	37227	69.69
keller5_EQP_t_(2,3,9)	2176	14
keller5_EQP_t_(2,4,12)	59	2.46
keller5_EQP_t_(2,4,13)	2091	10.68
keller5_EQP_t_(2,4,5)	2137	16.35
keller5_EQP_t_(2,4,7)	16	1.27
keller5_EQP_t_(2,4,8)	11666	31.21
keller5_EQP_t_(2,4,9)	400	8.44
keller5_EQP_t_(2,5,12)	13272	39.96
keller5_EQP_t_(2,5,13)	304102	623.23
keller5_EQP_t_(2,5,7)	12297	35.74
keller5_EQP_t_(2,5,8)	24881	76.43
keller5_EQP_t_(2,5,9)	250	14.39

Table A.2: Continued on next page

Table A.2: Continued from previous page

Inequality	**B&B Nodes**	**Total time**
keller5_EQP_t_(2,7,12)	0	0.44
keller5_EQP_t_(2,7,13)	5203	14.93
keller5_EQP_t_(2,7,8)	167566	270.88
keller5_EQP_t_(2,7,9)	7236	18.81
keller5_EQP_t_(2,8,12)	154291	295.77
keller5_EQP_t_(2,8,13)	78073	122.39
keller5_EQP_t_(2,8,9)	62708	118.66
keller5_EQP_t_(2,9,12)	2814	13.13
keller5_EQP_t_(2,9,13)	12509	32.9
keller5_EQP_t_(3,12,13)	31	1.1
keller5_EQP_t_(3,4,12)	0	0.01
keller5_EQP_t_(3,4,13)	39	4.75
keller5_EQP_t_(3,4,5)	268	7.11
keller5_EQP_t_(3,4,7)	0	0
keller5_EQP_t_(3,4,8)	143	5.53
keller5_EQP_t_(3,4,9)	9	4.84
keller5_EQP_t_(3,5,12)	3773	12.5
keller5_EQP_t_(3,5,13)	11426	31.32
keller5_EQP_t_(3,5,7)	1754	12.18
keller5_EQP_t_(3,5,8)	4629	21.72
keller5_EQP_t_(3,5,9)	839	14.04
keller5_EQP_t_(3,7,12)	5	0.25
keller5_EQP_t_(3,7,13)	88	3.11
keller5_EQP_t_(3,7,8)	6603	8.75
keller5_EQP_t_(3,7,9)	11	4.46
keller5_EQP_t_(3,8,12)	6007	9.07
keller5_EQP_t_(3,8,13)	279	4.33
keller5_EQP_t_(3,8,9)	272	7.95
keller5_EQP_t_(3,9,12)	18	3.64
keller5_EQP_t_(3,9,13)	13	3.22
keller5_EQP_t_(4,12,13)	0	0.17
keller5_EQP_t_(4,5,12)	575	7.73
keller5_EQP_t_(4,5,13)	87	8.63
keller5_EQP_t_(4,5,6)	106	6.35
keller5_EQP_t_(4,5,7)	255	6.03

Table A.2: Continued on next page

Table A.2: Continued from previous page

Inequality	**B&B Nodes**	**Total time**
keller5_EQP_t_(4,5,8)	457	7.7
keller5_EQP_t_(4,5,9)	604	8.07
keller5_EQP_t_(4,6,12)	0	0.01
keller5_EQP_t_(4,6,13)	0	0.02
keller5_EQP_t_(4,6,7)	0	0
keller5_EQP_t_(4,6,8)	27	0.64
keller5_EQP_t_(4,6,9)	0	0.67
keller5_EQP_t_(4,7,12)	9	0.19
keller5_EQP_t_(4,7,13)	4	0.56
keller5_EQP_t_(4,7,8)	53	1.58
keller5_EQP_t_(4,7,9)	0	2.59
keller5_EQP_t_(4,8,12)	515	2.8
keller5_EQP_t_(4,8,13)	116	5.23
keller5_EQP_t_(4,8,9)	19	5.54
keller5_EQP_t_(4,9,12)	0	0.35
keller5_EQP_t_(4,9,13)	0	2.8
keller5_EQP_t_(5,11,12)	1155	5.91
keller5_EQP_t_(5,11,13)	1227	8.58
keller5_EQP_t_(5,12,13)	7681	20.3
keller5_EQP_t_(5,6,11)	103	2.82
keller5_EQP_t_(5,6,12)	517	6.62
keller5_EQP_t_(5,6,13)	79	5.39
keller5_EQP_t_(5,6,7)	877	6.21
keller5_EQP_t_(5,6,8)	213	3.93
keller5_EQP_t_(5,6,9)	484	8.99
keller5_EQP_t_(5,7,11)	801	5.04
keller5_EQP_t_(5,7,12)	2923	10.35
keller5_EQP_t_(5,7,13)	5549	21.69
keller5_EQP_t_(5,7,8)	227	8.71
keller5_EQP_t_(5,7,9)	655	11.31
keller5_EQP_t_(5,8,11)	0	0.8
keller5_EQP_t_(5,8,12)	18947	42.49
keller5_EQP_t_(5,8,13)	6621	37.83
keller5_EQP_t_(5,8,9)	217	13.39
keller5_EQP_t_(5,9,11)	229	7.1

Table A.2: Continued on next page

Table A.2: Continued from previous page

Inequality	B&B Nodes	Total time
keller5_EQP_t_(5,9,12)	359	7.3
keller5_EQP_t_(5,9,13)	1675	18.4
keller5_EQP_t_(6,10,11)	0	0
keller5_EQP_t_(6,10,12)	0	0
keller5_EQP_t_(6,10,13)	0	0.06
keller5_EQP_t_(6,11,12)	0	0.01
keller5_EQP_t_(6,11,13)	0	0.18
keller5_EQP_t_(6,12,13)	0	0.28
keller5_EQP_t_(6,7,11)	0	0
keller5_EQP_t_(6,7,12)	9	1.23
keller5_EQP_t_(6,7,13)	0	1.04
keller5_EQP_t_(6,7,8)	209	0.97
keller5_EQP_t_(6,7,9)	0	0.97
keller5_EQP_t_(6,8,11)	13	0.51
keller5_EQP_t_(6,8,12)	127	2.72
keller5_EQP_t_(6,8,13)	132	3.2
keller5_EQP_t_(6,8,9)	4	3.79
keller5_EQP_t_(6,9,10)	0	0.03
keller5_EQP_t_(6,9,11)	0	0.2
keller5_EQP_t_(6,9,12)	0	0.37
keller5_EQP_t_(6,9,13)	0	0.41
keller5_EQP_t_(7,11,12)	91	0.47
keller5_EQP_t_(7,11,13)	0	0.92
keller5_EQP_t_(7,12,13)	11	1.43
keller5_EQP_t_(7,8,11)	171	0.84
keller5_EQP_t_(7,8,12)	975	5.25
keller5_EQP_t_(7,8,13)	581	5.54
keller5_EQP_t_(7,8,9)	7	4.74
keller5_EQP_t_(7,9,11)	0	0.39
keller5_EQP_t_(7,9,12)	0	0.44
keller5_EQP_t_(7,9,13)	3	0.7
keller5_EQP_t_(8,11,12)	365	1.18
keller5_EQP_t_(8,11,13)	398	2.43
keller5_EQP_t_(8,12,13)	771	4.97
keller5_EQP_t_(8,9,11)	11	3.32

Table A.2: Continued on next page

Table A.2: Continued from previous page

Inequality	**B&B Nodes**	**Total time**
keller5_EQP_t_(8,9,12)	9	4.53
keller5_EQP_t_(8,9,13)	41	4.23
keller5_EQP_t_(9,10,11)	0	0.03
keller5_EQP_t_(9,10,12)	0	0.03
keller5_EQP_t_(9,10,13)	0	0.04
keller5_EQP_t_(9,11,12)	0	0.02
keller5_EQP_t_(9,11,13)	0	0.04
keller5_EQP_t_(9,12,13)	0	0.05

Table A.3: Computing right-hand sides for 1zc-graphs

Inequality	**B&B Nodes**	**Total time**	`Nauty` **time**
1zc512_EQP_v_0	0	0	0
1zc512_EQP_v_2	0	0	0
1zc512_EQP_v_4	0	0	0
1zc512_EQP_v_6	0	0	0
1zc512_EQP_v_8	0	0.17	0
1zc512_EQP_e_(0,3)	0	0	0
1zc512_EQP_e_(2,5)	0	0	0
1zc512_EQP_e_(4,7)	0	0.06	0
1zc512_EQP_e_(6,9)	0	0.12	0
1zc512_EQP_e_(8,9)	0	0.5	0
1zc512_EQP_n_2	0	0	0
1zc512_EQP_n_4	0	0.02	0
1zc512_EQP_n_6	15	0.87	0.02
1zc512_EQP_n_8	0	0.48	0
1zc1024_EQP_v_0	0	0	0
1zc1024_EQP_v_10	0	0.99	0
1zc1024_EQP_v_2	0	0	0
1zc1024_EQP_v_4	0	0	0
1zc1024_EQP_v_6	0	0.08	0
1zc1024_EQP_v_8	0	0.03	0
1zc1024_EQP_e_(0,3)	0	0	0
1zc1024_EQP_e_(2,5)	0	0	0

Table A.3: Continued on next page

Table A.3: Continued from previous page

Inequality	**B&B Nodes**	**Total time**	`Nauty` **time**
1zc1024_EQP_e_(4,7)	23	0.4	0.02
1zc1024_EQP_e_(6,9)	19609	30.81	18.92
1zc1024_EQP_e_(8,10)	0	0.22	0
1zc1024_EQP_n_10	0	2.11	0
1zc1024_EQP_n_2	0	0	0
1zc1024_EQP_n_4	0	0.05	0
1zc1024_EQP_n_6	1313	5.76	1.6
1zc1024_EQP_n_8	4632099	35633.77	0
1zc2048_EQP_v_0	0	0	0
1zc2048_EQP_v_10	0	0.22	0
1zc2048_EQP_v_2	0	0	0
1zc2048_EQP_v_4	0	0	0
1zc2048_EQP_v_6	23	0.4	0.02
1zc2048_EQP_v_8	19609	30.81	18.92
1zc2048_EQP_e_(0,3)	0	0	0
1zc2048_EQP_e_(2,5)	0	0	0
1zc2048_EQP_n_(4,7)	-	-	-

Appendix B

Source Code

In this Appendix, we show the source code of main functions described in Sec. 4.2.

Listing B.1: Function `cep`

```
NodePtr * cep(int **a, int n, int *dim_ep)
{
        void partition(int **adj, NodePtr *part, int *p, int i, int j, int *eq);
        NodePtr * eqPart;
        NodePtr newNode, current;
        int i, j, k;
        int p, eqp;
        int * const pPtr = &p;
        int * const equi = &eqp;

        eqPart = (NodePtr*) malloc(n * sizeof(NodePtr));
        eqPart[0] = (NodePtr) malloc(sizeof(Node));
        eqPart[0]->id = 0;
        eqPart[0]->next = NULL;
        p = 1;
        for(i=1; i<n; i++){
                eqPart[i] = NULL;
        }
        current = eqPart[0];
        for(i=0; i<n; i++){
                newNode = (NodePtr) malloc(sizeof(Node));
                newNode->id = i;
                newNode->next = NULL;
                current->next = newNode;
                current = current->next;
        }
        /*main procedure*/
        i = 0;
```

```
29          while(i<p){
                    j = 0;
31                  k = p;
                    if((eqPart[i]->next)->next!=NULL){
33                          while(j<p){
                                    eqp = 1;
35                                  partition(a, eqPart, pPtr, i, j, equi);
                                    if(eqp==1){
37                                          j++;
                                    }
39                                  else{
                                            j = p;
41                                  }
                            }
43                  }
                    if(k==p){
45                          i++;
                    }
47                  else{
                            i = 0;
49                  }
            }
51          *dim_ep = p;
            return eqPart;
53  }
```

Listing B.2: Functions `printEQP`, `printEQPpar`, `EPgraphDOTForm`

```
1   void printEQP(char *filename, NodePtr *eqPart, int p, int classLP, int**a)
    {
3           FILE *file;
            char fileN[500];
5           int i, j, k;
            int *v;
7           NodePtr current;

9           strcpy(fileN, filename);
            strcat(fileN, ".eqp");
11          file = fopen(fileN, "w");
            printf("\nEquitable partition:\n\n");
13          for(i=0; i<p; i++){
                    printf("V[%3d]= ", i);
15                  fprintf(file, "V[%3d]= ", i);
                    current = eqPart[i]->next;
17                  k = 0;
```

```
                while(current!=NULL){
19                      k++;
                        current = current->next;
21              }
                if(classLP==1){
23                      v = (int*) malloc(k*sizeof(int));
                        j = 0;
25              }
                current = eqPart[i]->next;
27              while(current!=NULL){
                        printf("%d ", current->id+1);
29                      fprintf(file, "%d ", current->id+1);
                        if(classLP==1){
31                              v[j] = current->id;
                                j++;
33                      }
                        current = current->next;
35              }
                if(classLP==1){
37                      extractSUBgrf(i, v, k, a);
                        free(v);
39              }
                printf("\n");
41              fprintf(file, "\n");
        }
43      fclose(file);
}
45
void printEQPpar(char *filename, NodePtr *eqPart, int ** a, int p)
47 {
        FILE *eqpP;
49      char fileN[500];
        int i, j, k, flag;
51      NodePtr current;

53      strcpy(fileN, filename);
        strcat(fileN, ".par");
55    eqpP = fopen(fileN, "w");
    fprintf(eqpP, "Matrix:");
57  for(i=0; i<p; i++){
                flag = eqPart[i]->next->id;
59              fprintf(eqpP, "\n");
                for(j=0; j<p; j++){
61                      k = 0;
```

```
                        current = eqPart[j]->next;
63                      while(current!=NULL){
                                if(a[flag][current->id]==1){
65                                      k++;
                                }
67                              current = current->next;
                        }
69                      fprintf(eqpP, "%2d ", k);
                }
71
        }
73      fprintf(eqpP, "\n\nVector:");
        for(j=0; j<p; j++){
75                      k = 0;
                        current = eqPart[j]->next;
77                      while(current!=NULL){
                                k++;
79                              current = current->next;
                        }
81                      fprintf(eqpP, "\n%3d ", k);
    }
83  fclose(eqpP);
}
85
void EPgraphDOTform(char *filename, NodePtr *eqPart, int ** a, int p)
87 {
        FILE *eqpP;
89      char fileN[100];
        int i, j, k, flag;
91      NodePtr current;

93      strcpy(fileN, filename);
        strcat(fileN, "_EPgraph.dot");
95  eqpP = fopen(fileN, "w");
    fprintf(eqpP, "graph G {\n\toverlap=scale;\n\tnode[shape=circle];\n\tedge[len=8];");
97      for(i=0; i<p-1; i++){
                flag = eqPart[i]->next->id;
99              for(j=i+1; j<p; j++){
                        k = -1;
101                     current = eqPart[j]->next;
                        while(current!=NULL && k==-1){
103                             if(a[flag][current->id]==1){
                                        k = current->id;
105                             }
```

```
                        current = current->next;
107                 }
                    if(k!=-1){
109                     fprintf(eqpP, "\n\t%d -- %d;", i, j);
                    }
111             }
        }
113     fprintf(eqpP, "\n}");
        fclose(eqpP);
115 }
```

Listing B.3: Function `refineEQP`

```
1  void refineEQP(NodePtr *eqPart, int dim_p, int **a, int n, int *dim_ep, int *param)
   {
3       void partition(int **adj, NodePtr *part, int *p, int i, int j, int *eq);

5       NodePtr newNode, current, prev, curr;
        int i, j, k, l, x, y;
7       int p, eqp;
        int * const pPtr = &p;
9       int * const equi = &eqp;

11      p = dim_p;
        k = n+1;
13      l = -1;
        if(param==NULL){
15          for(i=0; i<p; i++){
                j = 0;
17              current = eqPart[i]->next;
                while(current!=NULL){
19                  j++;
                    current = current->next;
21              }
                if(k>j && j>1){
23                  k = j;
                    l = i;
25              }
            }
27      }
        else{
29          for(i=1; i<=param[0]; i++){
                j = 0;
31              current = eqPart[param[i]]->next;
                while(current!=NULL){
```

```
33                              j++;
                                current = current->next;
35                      }
                        if(k>j && j>1){
37                              k = j;
                                l = param[i];
39                      }
                }
41      }
        p += 1;
43      eqPart[p-1] = malloc(sizeof(Node));
        eqPart[p-1]->id = p-1;
45      eqPart[p-1]->next = NULL;
        for(i=p-1; i>=l+2; i--){
47              eqPart[i]->next = eqPart[i-1]->next;
        }
49      current = eqPart[l]->next;
        eqPart[l+1]->next = current;
51      eqPart[l]->next = current->next;
        current->next = NULL;
53      i = 0;
        while(i<p){
55              j = 0;
                k = p;
57              if((eqPart[i]->next)->next!=NULL){
                        while(j<p){
59                              eqp = 1;
                                partition(a, eqPart, pPtr, i, j, equi);
61                              if(eqp==1){
                                        j++;
63                              }
                                else{
65                                      j = p;
                                }
67                      }
                }
69              if(k==p){
                        i++;
71              }
                else{
73                      i = 0;
                }
75      }
        *dim_ep = p;
```

```
77 }
```

Listing B.4: Function `genEQP`

```
1  void genEQP(NodePtr *eqPart, int dim_p, int **a, int n, int *dim_ep, int **param)
   {
3          int i;
           refineEQP(eqPart, dim_p, a, n, dim_ep, NULL);
5          if(param!=NULL){
                   for(i=1; i<=param[0][0]; i++){
7                          refineEQP(eqPart, *dim_ep, a, n, dim_ep, param[i]);
                   }
9          }
   }
```

Listing B.5: Function `printEQP_v_rhs`

```
   void printEQP_v_rhs(char *filename, char *dir, char *clq, NodePtr *eqPart, int p, int n, int **a)
2  {
           int i, j, k, l, q, x, *v, **sg, rhs, *cns;
4          int status;
           NodePtr current;
6          char fileLP[500];
           CPXENVptr env = NULL;
8      CPXLPptr lp = NULL;
       FILE *file;
10
           if(n>p){
12                 env = CPXopenCPLEX(&status);
                   for(x=0; x<p; x++){
14                         v = malloc(n*sizeof(int));
                           l = 0;
16                         current = eqPart[x]->next;
                           while(current!=NULL){
18                                 v[l] = current->id;
                                   l++;
20                                 current = current->next;
                           }
22                         sortV(v, l);
                           sg = malloc(l*sizeof(int*));
24                         for(i=0; i<l; i++){
                                   sg[i] = malloc(l*sizeof(int));
26                         }
                           for(i=0; i<l; i++){
28                                 for(j=0; j<l; j++){
                                           sg[i][j] = a[v[i]][v[j]];
```

```
30                      }
                }
32              if(verifyISO(sg, l, dir, filename, &rhs)==0){
                        sprintf(fileLP, "data/%s/results/%s_EQP_v_%d.", dir, filename, x);
34                      printDimacs(fileLP, sg, l);
                        lp = CPXcreateprob(env, &status, "rhs");
36                      status = CPXreadcopyprob(env, lp, clq, NULL);
                        j = l-1;
38                      q = 0;
                        for(i=n-1; i>=0; i--){
40                              if(i!=v[j]){
                                        if(q==0){
42                                              k = i;
                                                q = 1;
44                                      }
                                }
46                              else{
                                        if(q==1){
48                                              status = CPXdelcols(env, lp, i+1, k);
                                                q = 0;
50                                      }
                                        if(j>=1){
52                                              j--;
                                        }
54                              }
                        }
56                      if(q==1){
                                status = CPXdelcols(env, lp, 0, k);
58                              q = 0;
                        }
60                      strcat(fileLP, "lp");
                        status = CPXwriteprob(env, lp, fileLP, NULL);
62                      status = CPXfreeprob(env, &lp);
                }
64              else{
                        sprintf(fileLP, "data/%s/results/%s_EQP_vertices.txt", dir, filename);
66                      file = fopen(fileLP, "a");
                        fprintf(file, "v %d %d\n", x, rhs);
68                      fclose(file);
                        sprintf(fileLP, "data/%s/results/%s_EQP.cns", dir, filename);
70                      file = fopen(fileLP, "ab");
                        cns = malloc((l+2)*sizeof(int));
72                      cns[0] = l;
                        cns[l+1] = rhs;
```

```
74                              for(i=0; i<l; i++){
                                        cns[i+1] = v[i]+1;
76                              }
                                fwrite(cns, (l+2)*sizeof(int), 1, file);
78                              fclose(file);
                                free(cns);
80                      }
                        freeMatrix(sg, l);
82                      free(v);
                }
84              status = CPXcloseCPLEX(&env);
        }
86      else if(n==p){
                FILE *file;
88              v = malloc(n*sizeof(int));
                int **b;
90              b = malloc(n*sizeof(int*));
                for(i=0; i<n; i++){
92                      v[i] = eqPart[i]->next->id;
                        b[i] = malloc(n*sizeof(int));
94              }
                for(i=0; i<n; i++){
96                      for(j=0; j<n; j++){
                                b[i][j] = a[v[i]][v[j]];
98                      }
                }
100             free(v);
                sprintf(fileLP, "data/%s/results/%s.epg", dir, filename);
102             file = fopen(fileLP, "wb");
                for(i=0; i<n; i++){
104                     b[i][i] = 1;
                        fwrite(b[i], n*sizeof(int), 1, file);
106                     free(b[i]);
                }
108             free(b);
                fclose(file);
110     }
}
```

Listing B.6: Function `verifyISO`

```
1 int verifyISO(int **a, int n, char *dir, char *filename, int *rhs)
  {
3       int i, j, m, e, l;
        char path[500];
```

```
5       FILE *file;
        SubpPtr sp;
7
        sprintf(path, "data/%s/EQPform/%s.rhs", dir, filename);
9       sp = malloc(sizeof(Subp));
        e = 0;
11      for(i=0; i<n-1; i++){
                for(j=i+1; j<n; j++){
13                      e += a[i][j];
                }
15      }
        file = fopen(path, "rb");
17      if(file!=NULL){
                j = 0;
19              while(feof(file)==0 && j==0){
                        fread(sp, sizeof(Subp), 1, file);
21                      if(sp->n==n && sp->m==e && sp->opt==1){
                                j = 1;
23                      }
                }
25              fclose(file);
                if(j==1){
27                      int **b;
                        b = dimacs2adjM(sp->id, &n);
29                      DYNALLSTAT(int, lab1, lab1_sz);
                        DYNALLSTAT(int, lab2, lab2_sz);
31                      DYNALLSTAT(int, ptn, ptn_sz);
                        DYNALLSTAT(int, orbits, orbits_sz);
33                      DYNALLSTAT(setword, workspace, workspace_sz);
                        statsblk stats;
35                      static DEFAULTOPTIONS_SPARSEGRAPH(options);
                        options.getcanon = TRUE;
37                      sparsegraph sg1, sg2, cg1, cg2;
                        m = (n+WORDSIZE-1)/WORDSIZE;
39                      SG_INIT(sg1);
                        SG_INIT(sg2);
41                      SG_INIT(cg1);
                        SG_INIT(cg2);
43                      DYNALLOC1(setword, workspace, workspace_sz, 2*m, "malloc");
                        DYNALLOC1(int, lab1, lab1_sz, n, "malloc");
45                      DYNALLOC1(int, lab2, lab2_sz, n, "malloc");
                        DYNALLOC1(int, ptn, ptn_sz, n, "malloc");
47                      DYNALLOC1(int, orbits, orbits_sz, n, "malloc");
                        SG_ALLOC(sg1, n, 2*e, "malloc");
```

```
49                  SG_ALLOC(sg2, n, 2*e, "malloc");
                    sg1.nv = n;
51                  sg1.nde = 2*e;
                    sg2.nv = n;
53                  sg2.nde = 2*e;
                    l = 0;
55                  for(i=0; i<n; i++){
                            sg1.v[i] = l;
57                          sg1.d[i] = 0;
                            for(j=0; j<n; j++){
59                                  if(a[i][j]==1){
                                            sg1.d[i]++;
61                                          sg1.e[l] = j;
                                            l++;
63                                  }
                            }
65                  }
                    l = 0;
67                  for(i=0; i<n; i++){
                            sg2.v[i] = l;
69                          sg2.d[i] = 0;
                            for(j=0; j<n; j++){
71                                  if(b[i][j]==1){
                                            sg2.d[i]++;
73                                          sg2.e[l] = j;
                                            l++;
75                                  }
                            }
77                  }
                    nauty((graph*)&sg1, lab1, ptn, NULL, orbits, &options, &stats,
79                  workspace, 2*m, m, n, (graph*)&cg1);
                    nauty((graph*)&sg2, lab2, ptn, NULL, orbits, &options, &stats,
81                  workspace, 2*m, m, n, (graph*)&cg2);
                    if(aresame_sg(&cg1, &cg2)==TRUE){
83                          *rhs = (int) sp->sol;
                            DYNFREE(workspace, workspace_sz);
85                          DYNFREE(lab1,lab1_sz);
                            DYNFREE(lab2,lab2_sz);
87                          DYNFREE(ptn, ptn_sz);
                            DYNFREE(orbits, orbits_sz);
89                          SG_FREE(sg1);
                            SG_FREE(sg2);
91                          free(sp);
                            freeMatrix(b, n);
```

```
93                              return 1;
                        }
95                      else{
                                DYNFREE(workspace, workspace_sz);
97                              DYNFREE(lab1,lab1_sz);
                                DYNFREE(lab2,lab2_sz);
99                              DYNFREE(ptn, ptn_sz);
                                DYNFREE(orbits, orbits_sz);
101                             SG_FREE(sg1);
                                SG_FREE(sg2);
103                             free(sp);
                                freeMatrix(b, n);
105                             return 0;
                        }
107             }
                else{
109                     free(sp);
                        return 0;
111             }
        }
113     else{
                free(sp);
115             return 0;
        }
117 }
```

Listing B.7: Function `cpxRHS`

```
1 double cpxRHS(int **a, int n, char *filename, char *dir, int f,
                 int *nodecount, double *setup_t)
3 {
        void handler(int sig);
5       int i, j, k, status, mipstat;
        double objval;
7       char string[100], path[100], *s;
        clock_t start, end;
9       CPXENVptr env = NULL;
        CPXLPptr lp = NULL;
11      CPXFILEptr file;
        FILE *fl;
13      SubpPtr sp;
        int colnamespace, surplus, numcols, *cns;
15      char *colnamestore, **colname;

17      signal(SIGINT, handler);
```

```
        start = clock();
19      env = CPXopenCPLEX(&status);
        lp = CPXcreateprob(env, &status, "rhs");
21      i = 0;
        while(filename[i]!='_' || filename[i+1]!='E' || filename[i+2]!='Q' || filename[i+3]!='P'){
23              i++;
        }
25      strncpy(path, filename, i);
        path[i] = '\0';
27      sprintf(string, "data/%s/results/%s.log", dir, filename);
        file = CPXfopen(string, "w");
29      status = CPXsetlogfile(env, file);
        status = CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_ON);
31      status = CPXsetterminate(env, &terminator);
        status = CPXsetintparam(env, CPX_PARAM_THREADS, 0);
33      sprintf(string, "data/%s/results/%s.lp", dir, filename);
        status = CPXreadcopyprob(env, lp, string, NULL);
35      end = clock();
        status = CPXmipopt(env, lp);
37      status = CPXgetbestobjval(env, lp, &objval);
        mipstat = CPXgetstat(env, lp);
39      strcpy(string, filename);
        s = strtok(string, "_");
41      while(s!=NULL){
                if(strncmp(s, "v", 1)==0){
43                      s = strtok(NULL, "_");
                        sscanf(s, "%d", &j);
45                      sprintf(string, "data/%s/results/%s_EQP_vertices.txt", dir, path);
                        fl = fopen(string, "a");
47                      fprintf(fl, "v_%d_%.0lf\n", j, floor(objval+1e-9));
                        fclose(fl);
49                      s = NULL;
                }
51              else if(strncmp(s, "e", 1)==0){
                        s = strtok(NULL, "_(,");
53                      sscanf(s, "%d", &i);
                        s = strtok(NULL, ")");
55                      sscanf(s, "%d", &j);
                        sprintf(string, "data/%s/results/%s_EQP_edges.txt", dir, path);
57                      fl = fopen(string, "a");
                        fprintf(fl, "e_%d_%d_%.0lf\n", i, j, floor(objval+1e-9));
59                      fclose(fl);
                        s = NULL;
61              }
```

```
                else{
63                      s = strtok(NULL, "_");
                }
65      }
        k = 0;
67      for(i=0; i<n-1; i++){
                for(j=i+1; j<n; j++){
69                      k += a[i][j];
                }
71      }
        if(k>0){
73              sp = malloc(sizeof(Subp));
                sprintf(sp->id, "data/%s/EQPform/%s.stb", dir, filename);
75              sp->sol = objval;
                sp->n = n;
77              sp->m = k;
                if(mipstat==CPXMIP_OPTIMAL){
79                      sp->opt = 1;
                }
81              else{
                        sp->opt = 0;
83              }
                sprintf(string, "data/%s/EQPform/%s.rhs", dir, path);
85              fl = fopen(string, "ab");
                fwrite(sp, sizeof(Subp), 1, fl);
87              fclose(fl);
                free(sp);
89      }
        numcols = CPXgetnumcols(env, lp);
91      status = CPXgetcolname(env, lp, NULL, NULL, 0, &surplus, 0, numcols-1);
        colnamespace = -surplus;
93      colname = malloc(numcols*sizeof(char*));
        colnamestore = malloc(colnamespace*sizeof(char));
95      status = CPXgetcolname(env, lp, colname, colnamestore, colnamespace,
                                &surplus, 0, numcols-1);
97      sprintf(string, "data/%s/results/%s_EQP.cns", dir, path);
        fl = fopen(string, "ab");
99      cns = malloc((n+2)*sizeof(int));
        cns[n+1] = (int) (objval + 1e-9);
101     cns[0] = n;
        for(i=0; i<numcols; i++){
103             s = strtok(colname[i], "x");
                sscanf(s, "%d", &j);
105             cns[i+1] = j;
```

```
        }
107     fwrite(cns, (n+2)*sizeof(int), 1, fl);
        fclose(fl);
109     free(cns);
        free(colnamestore);
111     free(colname);
        *nodecount += CPXgetnodecnt(env, lp);
113     *setup_t = ((double) (end - start))/CLOCKS_PER_SEC;
        CPXfclose(file);
115     status = CPXfreeprob(env, &lp);
        status = CPXcloseCPLEX(&env);
117     return objval;
}
```

Listing B.8: Functions `orbRHS` and `usersetbranch`

```
double orbRHS(int **a, int n, char *filename, char *dir, int f, int *nodecount,
2                 double *nty_t, double *setup_t)
{
4       void handler(int sig);
        DataPtr usrData;
6       int i, j, k, *av, status, mipstat;
        double objval;
8       char string[100], *s, path[100];
        clock_t start, end;
10      CPXENVptr env = NULL;
        CPXLPptr lp = NULL;
12      CPXFILEptr file;
        FILE *fl;
14      SubpPtr sp;
        int colnamespace, surplus, numcols, *cns;
16      char *colnamestore, **colname;

18      signal(SIGINT, handler);
        start = clock();
20      env = CPXopenCPLEX(&status);
        lp = CPXcreateprob(env, &status, "rhs");
22      i = 0;
        while(filename[i]!='_' || filename[i+1]!='E' || filename[i+2]!='Q' || filename[i+3]!='P'){
24              i++;
        }
26      strncpy(path, filename, i);
        path[i] = '\0';
28      sprintf(string, "data/%s/results/%s.log", dir, filename);
        file = CPXfopen(string, "w");
```

```
30      status = CPXsetlogfile(env, file);

32      status = CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_ON);
        status = CPXsetterminate(env, &terminator);
34      status = CPXsetintparam(env, CPX_PARAM_THREADS, 0);
        status = CPXsetintparam(env, CPX_PARAM_MIPCBREDLP, CPX_OFF);
36      status = CPXsetintparam(env, CPX_PARAM_MIPEMPHASIS, 4);
        sprintf(string, "data/%s/results/%s.lp", dir, filename);
38      status = CPXreadcopyprob(env, lp, string, NULL);
        usrData = malloc(sizeof(Data));
40      usrData->av = malloc((n*n+2)*sizeof(int));
        usrData->av[0] = n;
42      usrData->av[1] = f;
        for(i=0; i<n; i++){
44              for(j=0; j<n; j++){
                        usrData->av[2+i*n+j] = a[i][j];
46              }
        }
48      usrData->ntyTime = 0.0;
        end = clock();
50      status = CPXsetbranchcallbackfunc(env, usersetbranch, usrData);
        status = CPXmipopt(env, lp);
52      status = CPXgetbestobjval(env, lp, &objval);
        mipstat = CPXgetstat(env, lp);
54      strcpy(string, filename);
        s = strtok(string, "_");
56      while(s!=NULL){
                if(strncmp(s, "v", 1)==0){
58                      s = strtok(NULL, "_");
                        sscanf(s, "%d", &j);
60                      sprintf(string, "data/%s/results/%s_EQP_vertices.txt", dir, path);
                        fl = fopen(string, "a");
62                      fprintf(fl, "v_%d_%.0lf\n", j, floor(objval+1e-9));
                        fclose(fl);
64                      s = NULL;
                }
66              else if(strncmp(s, "e", 1)==0){
                        s = strtok(NULL, "_(,");
68                      sscanf(s, "%d", &i);
                        s = strtok(NULL, ")");
70                      sscanf(s, "%d", &j);
                        sprintf(string, "data/%s/results/%s_EQP_edges.txt", dir, path);
72                      fl = fopen(string, "a");
                        fprintf(fl, "e_%d_%d_%.0lf\n", i, j, floor(objval+1e-9));
```

```
74                      fclose(fl);
                        s = NULL;
76              }
                else{
78                      s = strtok(NULL, "_");
                }
80      }
        k = 0;
82      for(i=0; i<n-1; i++){
                for(j=i+1; j<n; j++){
84                      k += a[i][j];
                }
86      }
        if(k>0){
88              sp = malloc(sizeof(Subp));
                sprintf(sp->id, "data/%s/EQPform/%s.stb", dir, filename);
90              sp->sol = objval;
                sp->n = n;
92              sp->m = k;
                if(mipstat==CPXMIP_OPTIMAL){
94                      sp->opt = 1;
                }
96              else{
                        sp->opt = 0;
98              }
                sprintf(string, "data/%s/EQPform/%s.rhs", dir, path);
100             fl = fopen(string, "ab");
                fwrite(sp, sizeof(Subp), 1, fl);
102             fclose(fl);
                free(sp);
104     }
        numcols = CPXgetnumcols(env, lp);
106     status = CPXgetcolname(env, lp, NULL, NULL, 0, &surplus, 0, numcols-1);
        colnamespace = -surplus;
108     colname = malloc(numcols*sizeof(char*));
        colnamestore = malloc(colnamespace*sizeof(char));
110     status = CPXgetcolname(env, lp, colname, colnamestore, colnamespace,
                                &surplus, 0, numcols-1);
112     sprintf(string, "data/%s/results/%s_EQP.cns", dir, path);
        fl = fopen(string, "ab");
114     cns = malloc((n+2)*sizeof(int));
        cns[n+1] = (int) (objval + 1e-9);
116     cns[0] = n;
        for(i=0; i<numcols; i++){
```

```
118                 s = strtok(colname[i], "x");
                    sscanf(s, "%d", &j);
120                 cns[i+1] = j;
            }
122         fwrite(cns, (n+2)*sizeof(int), 1, fl);
            fclose(fl);
124         free(cns);
            free(colnamestore);
126         free(colname);
            *nodecount += CPXgetnodecnt(env, lp);
128         *setup_t = ((double) (end - start))/CLOCKS_PER_SEC;
            CPXfclose(file);
130         status = CPXfreeprob(env, &lp);
            status = CPXcloseCPLEX(&env);
132         return objval;
    }
134
    static int CPXPUBLIC
136 usersetbranch(CPXCENVptr env, void *cbdata, int wherefrom, void *cbhandle,
                  int brtype, int sos, int nodecnt, int bdcnt, const double *nodeest,
138               const int *nodebeg, const int *indices,
                  const char *lu, const int *bd, int *useraction_p)
140 {
            int status = 0;
142         int cols, n;
            double *lb, *ub;
144         double objval;
            int  *varindices;
146         char *varlu;
            int *varbd;
148         int seqnum0, seqnum1;
            CPXLPptr lp;
150
            DYNALLSTAT(graph,g,g_sz);
152         DYNALLSTAT(int,lab,lab_sz);
            DYNALLSTAT(int,ptn,ptn_sz);
154         DYNALLSTAT(int,orbits,orbits_sz);
            DYNALLSTAT(setword, workspace, workspace_sz);
156         static DEFAULTOPTIONS_GRAPH(options);
            statsblk stats;
158         set *gv;
            int m;
160
            int i, j, k, h, t, q, p, l;
```

```
162     int *v, *map, *o;
        int **Padj;
164     DataPtr usrData;
        clock_t start, end;
166
        *useraction_p = CPX_CALLBACK_DEFAULT;
168     status = CPXgetcallbacknodelp(env, cbdata, wherefrom, &lp);

170     /*ORBITAL BRANCHING*/
        usrData = (DataPtr) cbhandle;
172     n = usrData->av[0];
        lb = malloc(n*sizeof(double));
174     ub = malloc(n*sizeof(double));
        status = CPXgetcallbacknodelb(env, cbdata, wherefrom, lb, 0, n-1);
176     status = CPXgetcallbacknodeub(env, cbdata, wherefrom, ub, 0, n-1);
        status = CPXgetcallbacknodeobjval(env, cbdata, wherefrom, &objval);
178     v = malloc(n*sizeof(int));
        k = 0;
180     for(i=0; i<n; i++){
            if(lb[i]==1){
182             v[k] = i;
                k++;
184         }
        }
186     t = k;
        for(i=0; i<n; i++){
188         if(ub[i]==0){
                h = 0;
190             for(j=0; j<k; j++){
                    h += usrData->av[2+i*n+v[j]];
192             }
                if(h==0){
194                 v[t] = i;
                    t++;
196             }
            }
198     }
        start = clock();
200     if(usrData->av[1]==1){
            //fixing
202         sortV(v, t);
            map = malloc(n*sizeof(int));
204         j = 0;
            l = 0;
```

```
206             for(i=0; i<n; i++){
                    if(i<v[l]){
208                     map[j] = i;
                        j++;
210                 }
                    else if(l<t-1){
212                     l++;
                    }
214             }
                free(v);
216             t = j;
                Padj = malloc(t*sizeof(int*));
218             for(i=0; i<t; i++){
                    Padj[i] = malloc(t*sizeof(int));
220                 for(j=0; j<t; j++){
                        Padj[i][j] = usrData->av[2+map[i]*n+map[j]];
222                 }
                }
224             m = (t+WORDSIZE-1)/WORDSIZE;
                DYNALLOC2(graph, g, g_sz, m, t, "malloc");
226             DYNALLOC1(setword, workspace, workspace_sz, 5*m, "malloc");
                DYNALLOC1(int, lab, lab_sz, t, "malloc");
228             DYNALLOC1(int, ptn, ptn_sz, t, "malloc");
                DYNALLOC1(int, orbits, orbits_sz, t, "malloc");
230             for(i=0; i<t; i++){
                    gv = GRAPHROW(g, i, m);
232                 EMPTYSET(gv, m);
                    for(j=0; j<t; j++){
234                     if(Padj[i][j]==1){
                            ADDELEMENT(gv, j);
236                     }
                    }
238             }
                nauty(g, lab, ptn, NULL, orbits, &options, &stats, workspace,
240                   5*m, m, t, NULL);
                v = malloc(n*sizeof(int));
242             q = 0;
                for(i=0; i<t; i++){
244                 if(ub[map[i]]==0){
                        for(j=0; j<t; j++){
246                         if(j!=i && orbits[j]==orbits[i]
                               && lb[map[j]]==0 && ub[map[j]]==1){
248                             h = 0;
                                while(h<q && v[h]!=map[j]){
```

```
250                                                     h++;
                                                }
252                                             if(h==q){
                                                        v[q] = map[j];
254                                                     q++;
                                                }
256                                         }
                                    }
258                         }
                    }
260                 sortV(v, q);
                    freeMatrix(Padj, t);
262         }
            else{
264                 map = malloc(n*sizeof(int));
                    for(i=0; i<n; i++){
266                         map[i] = i;
                    }
268                 t = n;
                    q = 0;
270         }
            //branching
272         j = 0;
            for(i=0; i<n; i++){
274                 if(lb[i]==0 && ub[i]==1){
                            map[j] = i;
276                         j++;
                    }
278         }
            free(lb);
280         free(ub);
            t = j;
282         Padj = malloc(t*sizeof(int*));
            for(i=0; i<t; i++){
284                 Padj[i] = malloc(t*sizeof(int));
                    for(j=0; j<t; j++){
286                         Padj[i][j] = usrData->av[2+map[i]*n+map[j]];
                    }
288         }
            m = (t+WORDSIZE-1)/WORDSIZE;
290         DYNALLOC2(graph, g, g_sz, m, t, "malloc");
            DYNALLOC1(setword, workspace, workspace_sz, 5*m, "malloc");
292         DYNALLOC1(int, lab, lab_sz, t, "malloc");
            DYNALLOC1(int, ptn, ptn_sz, t, "malloc");
```

```
294         DYNALLOC1(int, orbits, orbits_sz, t, "malloc");
            for(i=0; i<t; i++){
296             gv = GRAPHROW(g, i, m);
                EMPTYSET(gv, m);
298             for(j=0; j<t; j++){
                    if(Padj[i][j]==1){
300                     ADDELEMENT(gv, j);
                    }
302             }
            }
304         nauty(g, lab, ptn, NULL, orbits, &options, &stats, workspace,
                5*m, m, t, NULL);
306         freeMatrix(Padj, t);
            k = 0;
308         o = malloc(t*sizeof(int));
            for(i=0; i<t; i++){
310             j = 0;
                while(j<k && orbits[i]!=o[j]){
312                 j++;
                }
314             if(j==k){
                    o[k] = orbits[i];
316                 k++;
                }
318         }
            h = 0;
320         p = -1;
            for(i=0; i<k; i++){
322             l = 0;
                for(j=0; j<t; j++){
324                 if(o[i]==orbits[j]){
                        l++;
326                 }
                }
328             if(l>h){
                    h = l;
330                 p = o[i];
                }
332
            }
334         free(o);
            end = clock();
336         usrData->ntyTime += ((double) (end - start))/CLOCKS_PER_SEC;
            /* SIDE 1 */
```

```
338         l = 0;
            for(i=0; i<n; i++){
340                 l += usrData->av[2+map[p]*n+i];
            }
342         j = q + 1 + l;
            varindices = malloc(j*sizeof(int));
344         varbd = malloc(j*sizeof(int));
            varlu = malloc(j*sizeof(char));
346         for(i=0; i<q; i++){
                    varindices[i] = v[i];
348                 varlu[i] = 'U';
                    varbd[i] = 0;
350         }
            varindices[i] = map[p];
352         varlu[i] = 'L';
            varbd[i] = 1;
354         i++;
            for(l=0; l<n; l++){
356                 if( usrData->av[2+map[p]*n+l]==1){
                            varindices[i] = l;
358                         varlu[i] = 'U';
                            varbd[i] = 0;
360                         i++;
                    }
362         }

364         status = CPXbranchcallbackbranchbds(env, cbdata, wherefrom, objval,
                                                j, varindices, varlu, varbd,
366                                             usrData, &seqnum1);
            free(varindices);
368         free(varlu);
            free(varbd);
370

            /* SIDE 0 */
372         j = q + h;
            varindices = malloc(j*sizeof(int));
374         varbd = malloc(j*sizeof(int));
            varlu = malloc(j*sizeof(char));
376         for(i=0; i<q; i++){
                    varindices[i] = v[i];
378                 varlu[i] = 'U';
                    varbd[i] = 0;
380         }
            for(l=0; l<t; l++){
```

```
382                 if(orbits[l]==p){
                            varindices[i] = map[l];
384                         varlu[i] = 'U';
                            varbd[i] = 0;
386                         i++;
                    }
388         }
            status = CPXbranchcallbackbranchbds(env, cbdata, wherefrom, objval,
390                                                 j, varindices, varlu, varbd,
                                                    usrData, &seqnum0);
392         free(varindices);
            free(varlu);
394         free(varbd);

396         free(map);
            free(v);
398
            DYNFREE(g,g_sz);
400         DYNFREE(workspace, workspace_sz);
            DYNFREE(lab,lab_sz);
402         DYNFREE(ptn,ptn_sz);
            DYNFREE(orbits,orbits_sz);
404
            *useraction_p = CPX_CALLBACK_SET;
406         return (status);
    }
```

Bibliography

[CS90] K. Corrádi and S. Szabó. A combinatorial approach for keller's conjecture. *Periodica Mathematica Hungarica*, 21(2):95–100, 1990.

[DEL+11] Jennifer Debroni, John D. Eblen, Michael A. Langston, Wendy Myrvold, Peter Shor, and Dinesh Weerapurage. A complete resolution of the keller maximum clique problem. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, pages 129–135. SIAM, 2011.

[Dim92] Dimacs. Clique benchmark instances (web site), 1992. `http://cs.hbg.psu.edu/txn131/clique.html`.

[EO98] T. Etzion and P. R.J. Ostergard. Greedy and heuristic algorithms for codes and colorings. *IEEE Trans. Inf. Theor.*, 44(1):382–388, September 1998.

[FNT74] D. R. Fulkerson, G. L. Nemhauser, and L. E. Trotter. Two computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of steiner triple systems. *Mathematical Programming Study*, 2:72–81, 1974.

[FR89] Thomas A Feo and Mauricio G. C Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Oper. Res. Lett.*, 8(2):67–71, April 1989.

[GLS81] Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[GLS88] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric algorithms and combinatorial optimization*. Springer4060 XII, 362 S, 1988.

[GLS92] DIMACS (GROUP), J.C. Lagarias, and P.W. Shor. *Keller's Cube-tiling Conjecture is False in High Dimensions*. 1992.

[Hås96] Johan Håstad. Clique is hard to approximate within $n^{1\text{-epsilon}}$. In *FOCS*, pages 627–636, 1996.

[HR09] Stephen G. Hartke and A.J. Radcliffe. *McKay's canonical graph labeling algorithm.*, pages 99–111. Providence, RI: American Mathematical Society (AMS), 2009.

[IBM11] IBM. *IBM ILOG CPLEX Optimization Studio 12.4 - CPLEX User's Manual*, 2011.

[Kha79] L. G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096, 1979.

[Kløv81] Torleiv Kløve. Upper bounds on codes correcting asymmetric errors. *IEEE Transactions on Information Theory*, 27(1):128–131, 1981.

[KP08] Volker Kaibel and Marc Pfetsch. Packing and partitioning orbitopes. *Math. Program.*, 114(1):1–36, March 2008.

[Mac02] J. Mackey. A cube tiling of dimension eight with no facesharing, 2002.

[Mar03] François Margot. Exploiting orbits in symmetric ilp. *Mathematical Programming*, pages 3–21, 2003.

[McK81] Brendan D. McKay. Practical graph isomorphism, 1981.

[Mck90] B. D. Mckay. nauty User's Guide (version 1.5), Tech. Rep. Technical report, Department Computer Science, Australian National University, 1990.

[MDZ08] Isabel Méndez-Díaz and Paula Zabala. A cutting plane algorithm for graph coloring. *Discrete Appl. Math.*, 156(2):159–179, January 2008.

[MS95] Carlo Mannino and Antonio Sassano. Solving hard set covering problems. *Oper. Res. Lett.*, 18(1):1–5, August 1995.

[OLRS11a] James Ostrowski, Jeff Linderoth, Fabrizio Rossi, and Stefano Smriglio. Orbital branching. *Mathematical Programming*, 126:147–178, 2011.

[OLRS11b] James Ostrowski, Jeff Linderoth, Fabrizio Rossi, and Stefano Smriglio. Solving large steiner triple covering problems. *Operations Research Letters*, 39(2):127 – 131, 2011.

[Ost09] J. Ostrowski. *Symmetry in Mixed Integer Programming.* PhD thesis, Lehigh University, 2009.

[Per40] O. Perron. Über lückenlose ausfüllung des n-dimensionalen raumes durch kongruente würfel, 1940.

[Res88] AT&T Research. Graphviz - graph visualization software (web site), 1988. `http://www.graphviz.org`.

[RS01] Fabrizio Rossi and Stefano Smriglio. A branch-and-cut algorithm for the maximum cardinality stable set problem. *Oper. Res. Lett.*, 28(2):63–74, 2001.

[RSU94] Motakuri V. Ramana, Edward R. Scheinerman, and Daniel Ullman. Fractional isomorphism of graphs. *Discrete Mathematics*, 132(1-3):247–265, 1994.

[Sch86] Alexander Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[Slo00] Neil J.A. Sloane. Challenge problems (web site), 2000. `http://neilsloane.com/doc/graphs.html`.

[SU97] E.R. Scheinerman and D.H. Ullman. *Fractional graph theory: a rational approach to the theory of graphs.* Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1997.

[Var65] R. R Varshamov. Some features of linear codes that correct asymmetric errors. *Trans. Soviet Physics-Doklady*, 9:538–540, 1965.