Contents

This chapter is to appear as

W.-J. van Hoeve. Semidefinite Programming and Constraint Programming. In M.F. Anjos and J.B. Lasserre (eds.), *Handbook on Semidefinite, Cone and Polynomial Optimization: Theory, Algorithms, Software and Applications*, Springer.

Semidefinite Programming and Constraint Programming

Willem-Jan van Hoeve

Tepper School of Business, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA
vanhoeve@andrew.cmu.edu

Recently, semidefinite programming relaxations have been applied in constraint programming to take advantage of the high-quality bounds and precise heuristic guidance during the search for a solution. The purpose of this chapter is to present an overview of these developments, and to provide future research prospects.

1 Introduction

Constraint programming is a modeling and solving paradigm for combinatorial optimization problems. In constraint programming, a combinatorial problem is modeled using variables that have an associated domain of possible values, and constraints over these variables. What makes constraint programming different from similar methods such as integer programming, is that the variable domains can be any finite set of elements, and the constraints can be of any type. For example, a constraint can be defined explicitly by a list of allowed tuples, or implicitly by an expression such as $\texttt{alldifferent}(x_1, x_2, \ldots, x_n)$, which specifies that the variables $x_1, x_2, \ldots, x_n$ take pairwise different values. In contrast, integer linear programming restricts the variables to take either a continuous or integer valued domain (in fact, an interval), while constraints can be only linear expressions. Hence, constraint programming offers a very rich, and often convenient, modeling language.

In order to solve a given model, a constraint programming solver implicitly enumerates all possible variable-value combinations through a systematic search process until a solution satisfying all constraints has been found (or until it is proved that no solution exists). Most constraint programming systems allow the user to specify a particular search strategy. For example, criteria can be expressed to determine the variable order to branch on, as well as the value order to assign to those variables.

To reduce the underlying exponentially large search space, so-called *domain filtering* algorithms are applied during the systematic search process. Each constraint has an associated filtering algorithm. Its task is to remove provably inconsistent values from the domains of the variables that are in the scope of the constraint, based on the individual constraint only. When a filtering algorithm for one constraint has

processed the variables in its scope, the updated domains are propagated to the other constraints that share these variables, whose filtering algorithms are in turn activated. This cascading effect is called constraint propagation. Whenever a filtering algorithm makes a domain empty, we know that at that point of the search no solution is possible, and we backtrack to a previous search state.

By design, the inference of domain filtering is based on feasibility of each domain value. For optimization problems, similar inference can be performed by means of *cost-based* domain filtering. That is, a domain value is considered infeasible if assigning it to its variable will always yield a suboptimal solution (if any). Focacci, Lodi, and Milano [1999a] introduced a systematic way of designing cost-based domain filtering algorithms for so-called 'optimization constraints', through embedding an associated relaxation. Although they apply linear relaxations, conceptually any suitable relaxation can be embedded, including semidefinite relaxations, as proposed by Van Hoeve [2003, 2006].

From a constraint programming perspective, there are two major reasons why semidefinite relaxations could be successfully integrated. First, the bounds obtained can be much tighter than those obtained with linear relaxations, and can help to improve the optimization reasoning. Second, the fractional solution of the semidefinite relaxation can be applied as an accurate search heuristic. It should be noted that the intention here is not to solve a semidefinite relaxation at every node of the search tree necessarily. Instead, a semidefinite relaxation may be solved only once at the root node, or at selected times during the search process.

Also from a semidefinite programming perspective, the integration with constraint programming can be beneficial. Namely, it is not uncommon that solving a semidefinite relaxation takes so much time that embedding it inside a pure SDP-based branch-and-bound scheme does not pay off. Instead, the approaches studied in this chapter focus on a limited number of relaxations to be solved, while different information is extracted and utilized from the solution, namely for domain filtering and search heuristics. These ideas may be applied also inside a pure SDP-based solver.

The remainder of the chapter is structured as follows. In Section 2 we present necessary background information on constraint programming. In Section 3 we describe how semidefinite relaxations have been applied within constraint programming. Section 4 presents a first application of this, for stable set problems. In Section 5 we discuss the accuracy of semidefinite relaxations when applied as a search heuristic. In Section 6, an integrated approach to a telecommunications application is presented, corresponding to the biclique completion problem. Finally, Section 7 presents conclusions and future research perspectives.

2 Constraint Programming

We first introduce basic constraint programming concepts that are necessary for this chapter. For more information on constraint programming we refer to the textbooks

by Apt [2003] and Dechter [2003], and to the Handbook on Constraint Programming by Rossi et al. [2006]. For more information on the integration of constraint programming and operations research we refer to Hooker [2000], Milano [2003] and Hooker [2007].

2.1 Modeling

Even though constraint programming has many similarities with integer linear programming (ILP), there are some basic differences, also in terminology, that may cause confusion.

First, ILP assumes that variables take a value from a continuous interval represented by a lower and upper bound. In ILP systems, a variable is considered to be integer if its value is within a small tolerance factor of an integer number. In contrast, a CP variable takes its value from a specific finite set called its *domain*. For a variable x, we denote its domain by $D(x)$. Variable domains are represented explicitly in CP systems. This is a fundamental difference between ILP systems and CP systems. Namely, the explicit domain representation of CP not only allows more expressive models, but also allows to represent high-level inferences (e.g., we can remove individual values from a variable domain). On the other hand, the continuity assumption and linearity of ILP models allow the use of efficient linear programming solvers, but inferences are limited to those that can be represented by linear constraints and changes in the bounds of variables.

In CP, constraints can take several forms. Formally, a constraint C on a set of variables $X = \{x_1, x_2, \ldots, x_k\}$ is defined as a subset of the Cartesian product of the domains of the variables in X, i.e., $C \subseteq D(x_1) \times D(x_2) \times \cdots \times D(x_k)$. A tuple $(d_1, \ldots, d_k) \in C$ is called a *solution* to C. An immediate form to represent constraints is thus to list out the individual tuples that form the definition, and we say that such constraints are given 'in extension'. For several applications, such as database applications, such representation is very natural and useful. However, often we prefer to model combinatorial problems using implicit formulations, for example using linear expressions. Most constraint programming systems allow constraints to be defined by any algebraic expression, using common operators such as addition and multiplication, but also logical connectives in which sub-expressions serve as Boolean functions, whose truth value can be interpreted as a binary value. For example, the constraint

$$z = \sum_{i=1}^{n} (x_i \geq b)$$

restricts variable z to be equal to the number of variables $x_1, \ldots, x_n$ that take a value of at least b. Furthermore, variables can appear as subscripts. For example, for an index set I, consider an array $c \in \mathbb{Q}^{|I|}$, a variable x with domain $D(x) = I$, and a variable z. The expression $z = c_x$ states that z will be assigned the x-th value in the array c. Such constraints, in which a variable functions as an index, are called 'element' constraints.

Constraint programming further allows the use of so-called *global constraints* [Régin, 2003; Van Hoeve and Katriel, 2006]. Global constraints are defined on an

arbitrary number of variables, and usually encapsulate a combinatorial structure that can be exploited during the solving process. A well-known example is the constraint $\texttt{alldifferent}(x_1, x_2, \ldots, x_n)$, that requires the variables $x_1, x_2, \ldots, x_n$ to take pairwise different values. Global constraints and element constraints can lead to very concise formulations. For example, consider a Traveling Salesman Problem on n cities, in which the distance between two cities i and j is denoted by $d_{i,j}$. The problem asks to find a closed tour visiting each city exactly once, with minimum total distance. We introduce a variable x_i representing the i-th city to be visited, for $i = 1, \ldots, n$. The problem can then be formulated as

$$\begin{aligned} \min \sum_{i=1}^{n-1} d_{x_i,x_{i+1}} + d_{x_n,x_1} & \\ \text{s.t. } \texttt{alldifferent}(x_1, x_2, \ldots, x_n), & \\ D(x_i) = \{1, 2, \ldots, n\}, & \qquad \forall i \in \{1, \ldots, n\}. \end{aligned}$$

In addition to global constraints, most CP systems contain other high-level modeling objects. For example, *set variables* are variables that will be assigned a set of elements [Puget, 1992; Gervet, 1994]. The domain of a set variable is a finite set of sets, that however often is exponentially large. Therefore, CP systems usually represent the domain of a set variable by a 'lower bound' of mandatory elements and an 'upper bound' of possible elements. In addition, a lower and upper bound on the cardinality of the set is maintained as part of the domain information. For example, if the domain of a set variable S is defined as $D(S) = [\{a,c\}, \{a,b,c,d,e\}]$ with $3 \leq |S| \leq 4\}$, the mandatory elements are $\{a,c\}$ while the possible elements are $\{a,b,c,d,e\}$ and in addition the cardinality must be between 3 and 4. That is, the domain implicitly includes the sets $\{a,b,c\}, \{a,c,d\}, \{a,c,e\}, \{a,b,c,d\}, \{a,b,c,e\}$, and $\{a,c,d,e\}$. As an illustration, we next model the weighted stable set problem using a set variable. Let $G = (V,E)$ be an undirected weighted graph with vertex set V, edge set E, and a 'weight' function $w : V \to \mathbb{R}$. The maximum weighted stable set problem asks to find a subset of vertices with maximum total weight, such that no two vertices in this subset share an edge. We introduce one set variable S representing the vertices that will form the stable set. The initial domain of this set variable is $D(S) = [\varnothing, V]$, and $0 \leq |S| \leq |V|$. That is, initially there are no mandatory elements, and all elements in V can potentially be in the stable set. The constraint programming model then becomes

$$\begin{aligned} \max \textstyle\sum_{i \in S} w_i & \\ \text{s.t. } (i \in S) \Rightarrow (j \notin S), & \qquad \forall (i,j) \in E, \\ D(S) = [\varnothing, V], 0 \leq |S| \leq |V|\,. & \end{aligned}$$

Note that this model makes use of logical implications to ensure that no two adjacent vertices belong to the stable set.

2.2 Solving

The solution process of constraint programming interleaves *constraint propagation* and *search*. During the search process, all possible variable-value combinations are

implicitly enumerated until we find a solution or prove that none exists. We say that this process constructs a *search tree*, in which the root node represents the original problem to be solved, and a branching decision partitions a node into at least two subproblems. To reduce the exponential number of combinations, *domain filtering* and *constraint propagation* is applied at each node of the search tree. A *domain filtering algorithm* operates on an individual constraint. Given a constraint, and the current domains of the variables in its scope, a domain filtering algorithm removes domain values that do not belong to a solution to the constraint. Such domain values are called *inconsistent* with respect to the constraint. Likewise, domain values that do belong to a solution are called *consistent* with respect to the constraint. Since variables usually participate in more than one constraint, the updated domains are propagated to the other constraints, whose domain filtering algorithms in effect become active. This process of constraint propagation is repeated for all constraints until no more domain values can be removed, or a domain becomes empty. Since the variable domains are finite, this process always terminates. Furthermore, it can be shown that under certain conditions on the domain filtering algorithms, the constraint propagation process always reaches the same fixed point, irrespective of the order in which the constraints are processed [Apt, 1999].

For example, consider the following constraint programming model (corresponding to a feasibility problem):

$$x_1 > x_2, \tag{1}$$

$$x_1 + x_2 = x_3, \tag{2}$$

$$\texttt{alldifferent}(x_1, x_2, x_3, x_4), \tag{3}$$

$$x_1 \in \{1,2\}, x_2 \in \{0,1,2,3\}, x_3 \in \{2,3\}, x_4 \in \{0,1\}. \tag{4}$$

If we process the constraints in the top-down order of the model, the constraint propagation first considers constraint (1), whose domain filtering algorithm removes the inconsistent values 2 and 3 from $D(x_2)$, i.e., $D(x_2) = \{0,1\}$ at this point. We then continue with constraint (2), for which all values are consistent. For constraint (3), observe that variables x_2 and x_4 now both have domain $\{0,1\}$, and thus saturate these values. Therefore, we can remove values 0 and 1 from the domains of the other variables in its scope, i.e., $D(x_1) = \{2\}$. As an immediate effect, the same algorithm will deduce that $D(x_3) = \{3\}$. At this point, we have processed all constraints, but note that since we have changed the domains of x_1, x_2, and x_3, we need to repeat the domain filtering for the constraints in which they participate. Considering again constraint (1), we find that all domain values are consistent. From constraint (2), however, we deduce that $D(x_2) = \{1\}$. Finally, reconsidering constraint (3) leads to $D(x_4) = \{0\}$. In summary, in this case the constraint propagation alone is able to find a solution to the problem, i.e., $x_1 = 2, x_2 = 1, x_3 = 3, x_4 = 0$.

In order to be effective, domain filtering algorithms should be computationally efficient, because they are applied many times during the solution process. Further, they should remove as many inconsistent values as possible. If a domain filtering algorithm for a constraint C removes *all* inconsistent values from the domains with

respect to C, we say that it makes C *domain consistent*.[1] In other words, all remaining domain values participate in at least one solution to C. More formally:

Definition 1 (Domain consistency). *A constraint C on the variables $x_1, \ldots, x_k$ is called* domain consistent *if for each variable x_i and each value $d_i \in D(x_i)$ $(i = 1, \ldots, k)$, there exist a value $d_j \in D(x_j)$ for all $j \neq i$ such that $(d_1, \ldots, d_k) \in C$.*

In practice, one usually tries to develop filtering algorithms that separate the check for consistency and the actual domain filtering, especially when processing global constraints. That is, we would like to avoid applying the algorithm that performs the consistency check to each individual variable-value pair, since that strategy can often be improved. Moreover, one typically tries to design incremental algorithms that re-use data structures and partial solutions from one filtering event to the next, instead of applying the filtering algorithm from scratch each time it is invoked.

Without effective domain filtering algorithms to reduce the search space, constraint programming would not be applicable in practice. For many practical problems, global constraints are the most effective, since they exploit the underlying combinatorial structure to identify more inconsistent values than a corresponding decomposition in 'elementary' constraints would identify. For example, consider a set of variables X that must take pairwise different values, while the domain of each variable $x \in X$ is $D(x) = \{1, \ldots, |X| - 1\}$. Clearly, as the number of variables exceeds the number of possible values that can be jointly assigned to them, the problem has no solution. However, if we model the problem using not-equal constraints $x_i \neq x_j$ for all pairs $x_i, x_j \in X$ with $i \neq j$, constraint propagation alone cannot deduce this fact. Namely, for each individual not-equal constraint $x_i \neq x_j$, the associated domain filtering algorithm will consider only the domains of x_i and x_j, and each value within these domains participates in a solution to that constraint. On the other hand, if we establish domain consistency for the global constraint $\texttt{alldifferent}(X)$, we would be able to immediately deduce that the constraint cannot be satisfied. In fact, Régin [1994] showed that domain consistency for `alldifferent` can be established in polynomial time by representing the constraint as a bipartite matching problem.

As a final remark on domain filtering, consider the constraint $(i \in S) \Rightarrow (j \notin S)$ as applied in the stable set model above. Such constraints, that involve the truth value of subexpressions, will internally be represented using Boolean variables by most CP systems. That is, one Boolean variable x represents whether the expression $(i \in S)$ is true or not, and another Boolean variable y represents the expression $(j \notin S)$. The associated domain filtering will then be performed on the relation $x \Rightarrow y$ and the relations $x = (i \in S)$ and $y = (j \notin S)$. However, such decomposition into a Boolean representation is not always necessary for set variables, as more efficient algorithms may be implemented directly, especially for global constraints on set variables. For example, the global constraint $\texttt{partition}(S_1, S_2, \ldots, S_n, V)$ requires that the set variables $S_1, S_2, \ldots, S_n$ form a partition of the set V. For this constraint, an efficient filtering algorithm can be designed based on a bipartite network flow representation [Sadler and Gervet, 2004; Régin, 2004].

[1] In the literature, domain consistency is also referred to as *hyper-arc consistency* or *generalized arc consistency* for historic reasons.

When a constraint programming model involves an objective function to be optimized, the default solving process applies the following straightforward bounding procedure. Consider an objective $\max f(X)$ where the objective function f can be an arbitrary expression mapped to a totally ordered set. Assuming that we have a known upper bound U on the objective, it will be treated as a constraint $f(X) \leq U$, and the associated domain filtering algorithms for this constraint will be applied. As soon as a solution satisfying all the constraints has been found, and that has objective value U', the right hand side of the 'objective' constraint will be updated as $f(X) \leq U'$, after which the search continues. Note that if the objective value of the incumbent solution is higher than the current upper bound, the associated domain filtering algorithm will detect an inconsistency and the solver will backtrack to a previous search state.

2.3 Optimization Constraints

As explained in the previous section, the CP solver evaluates the objective function as an individual constraint, where the potential solution set is formed by the Cartesian product of the domains of the variables in its scope. That is, by default it is not aware of any underlying combinatorial structure that may improve the bound of the objective, unless we add this structure explicitly to the constraint that represents the objective. Many works have studied how to improve the representation of the objective in constraint programming, and the resulting constraints are now often referred to as 'optimization constraints', or 'optimization oriented global constraints' [Focacci et al., 2002a].

Even though a formal definition of optimization constraints is not easy to provide, they can be thought of as weighted versions of global constraints. That is, for a global constraint $C(X)$ on a set of variables X, we can define a corresponding optimization constraint with respect to a 'weight' function $f(X)$ as

$$C(X) \wedge (f(X) \leq z),$$

where z is a variable representing the upper bound on the weight function. The optimization constraint thus ensures that we only allow solutions that respect constraint C and have a corresponding weight not more than z. The domain filtering associated with such reasoning is referred to as 'cost-based' domain filtering. Cost-based domain filtering algorithms usually work in two directions: One direction is to update the domain of z, i.e., improve the bound, based on the current domains of the variables X, while the second direction is to update the domains of the variables X based on the current bound of z. The latter direction is also referred to as 'back-propagation' in the literature.

While many optimization constraints utilize specialized combinatorial algorithms to perform cost-based filtering (see, e.g., [Régin, 2002; Van Hoeve et al., 2006; Katriel et al., 2007; Sellmann et al., 2007]), a generic methodology to designing cost-based filtering for optimization constraints was introduced in a sequence of papers by Focacci, Lodi, Milano, and Vigo [1999b] and Focacci, Lodi, and Milano

[1999a, 2002b,c]. In these works, the domain filtering is based on a linear relaxation of the associated global constraint together with a 'gradient function' $grad(x,v)$ that returns, for each possible variable-value pair (x,v) with $v \in D(x)$, an optimistic evaluation of the cost to be paid if v is assigned to x. For example, consider the constraint `alldifferent`$(x_1,\ldots,x_n)$, together with 'weights' w_{ij} for every pair (x_i,j) such that $j \in D(x_i)$ ($i \in \{1,\ldots,n\}$). We introduce binary variables y_{ij} for $i \in \{1,\ldots,n\}$ and $j \in D(x_i)$ such that

$$y_{ij} = 1 \Leftrightarrow (x_i = j), \text{ and}$$
$$y_{ij} = 0 \Leftrightarrow (x_i \neq j).$$

The weighted `alldifferent` constraint corresponds to the following Assignment Problem formulation:

$$z = \sum_{i=1}^{n} \sum_{j \in D(x_i)} w_{ij} y_{ij}$$
$$\sum_{j \in D(x_i)} y_{ij} = 1, \; \forall i \in \{1,\ldots,n\},$$
$$\sum_{i=1}^{n} y_{ij} \leq 1, \; \forall j \in \cup_{i=1}^{n} D(x_i),$$
$$y_{ij} \in \{0,1\}, \; \forall i \in \{1,\ldots,n\}, \forall j \in D(x_i),$$

where z represents the value of the objective function. It is well-known that the linear relaxation of the Assignment Problem yields an integer solution. Focacci et al. [1999a] propose to use the reduced costs obtained by solving this relaxation as the gradient function. That is, let $\overline{c}_{ij}$ represent the reduced cost of variable y_{ij} in a solution to the Assignment Problem relaxation with lower bound z^*, and let $z_{\min}$ be the current best solution value. Whenever $z^* + \overline{c}_{ij} > z_{\min}$, we can remove value j from $D(x_i)$. Note that in the Operations Research literature this technique is known under the name 'variable fixing' [Nemhauser and Wolsey, 1988]. This approach has been generalized to Lagrangian relaxations, where the gradient function is formed by the Lagrange multipliers, see for example [Hooker, 2006].

2.4 Search Strategies

Even though the search process of constraint programming has similarities with integer programming, there are several differences. The goal of the search process is to implicitly enumerate all possible variable-value combinations until a (provably optimal) solution has been found, or until it is proved that no solution exists. As stated before, this process is said to create a search tree, which is a particular rooted tree. The root node represents the original problem to be solved. At each node of the tree constraint propagation is applied, typically until a fixed point is reached. If this process does not prove infeasibility, and does not finish with a solution, the node is split into two or more subproblems. We must ensure that the union of the solution set of the subproblems is equal to the solution set of the problem represented by the node itself.

The splitting, or branching, procedure can take various forms, and an important difference with integer programming systems is that a user can define the branching decisions to be made. In many cases this can even be done at the modeling level, see, e.g., Van Hentenryck et al. [2000]. A branching decision is stated as a constraint, which will be added to the subproblem (its negation will be added to the other subproblem in case of a binary branching decision). For example, for a variable x with $v \in D(x)$, we can branch on the constraint $(x = v)$ versus $(x \neq v)$. This corresponds to the traditional enumeration of variable-value combinations.

In order to apply a particular branching constraint, we first need to identify one or more variables on which to define the constraint. When the branching decisions correspond to variable assignments $(x = v)$ versus $(x \neq v)$, the order in which variables, and corresponding values, are considered follows from so-called *variable selection* heuristics and *value selection* heuristics. That is, when the problem is not infeasible and one or more variables are still 'free' (i.e., they have have a domain with size larger than one), the variable selection heuristic will identify one variable x from the set of free variables. Similarly, the value selection heuristic will identify one value v from its domain $D(x)$. A standard heuristic in constraint programming is to choose first the variable with the smallest domain size (this is called fail-first), and assign it the smallest value in its domain. For more involved heuristics, we refer to [van Beek, 2006].

In addition to the branching decisions that define the shape of the search tree, the search process also depends on the order in which the different search tree nodes are visited. In integer programming, the order is almost always defined by some variant of a 'best-bound-first' strategy, in which the next node to explore is that with the most promising bound following from the linear relaxation. Even though this strategy is often superior in practice for integer programming problems, it has the potential drawback of creating an exponential number of active search nodes. Here, a node is considered active when it must be split further, i.e., it is not proved to be infeasible or suboptimal and it does not correspond to a solution. Best-bound-first and similar strategies can also be applied to constraint programming search trees. However, it is much more common to apply a depth-first search strategy instead. An important reason for this may be the fact that constraint programming does not contain 'primal heuristics' that can find feasible solutions quickly for a given search node. In fact, one may view the constraint programming search process itself as a primal heuristic. Furthermore, the memory consumption of depth-first is linear in the depth of the search tree, while this can be exponential for breadth-first search strategies (including best-bound-first).

Search strategies have been studied extensively in the context of constraint programming, and more generally in the context of artificial intelligence; see van Beek [2006] for an overview. One particular search strategy that will also be applied in this chapter is *limited discrepancy search*, introduced by Harvey and Ginsberg [1995]. It is based on the assumption that we have a good, but not perfect, branching heuristic available (e.g., corresponding to a variable and value selection heuristic). If the branching heuristic were perfect, the first visited leaf in the search tree (when applying depth-first search) would correspond to a solution to the problem (assuming it

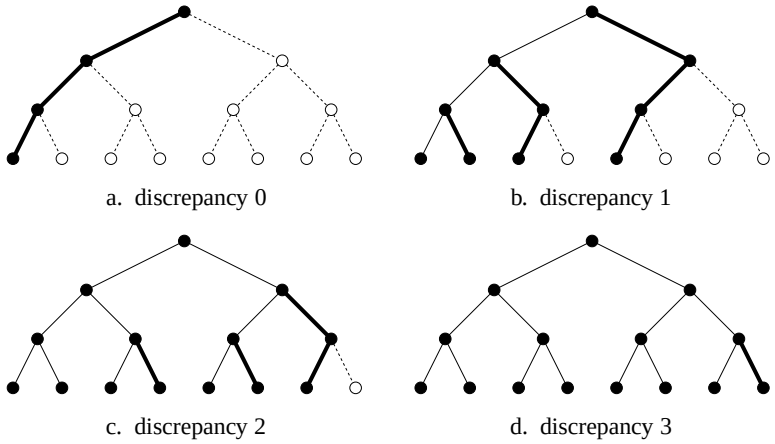


Fig. 1. Limited discrepancy search. For each discrepancy 0, 1, 2, 3, the top node of the tree indicates the root, visited nodes are filled, and bold arcs indicate the parts of the tree that are traversed newly.

exists). Harvey and Ginsberg [1995] argue that when the heuristic is equally likely to make a mistake at each search node, we should visit the leafs by increasing value of total discrepancy from the heuristic choice required to reach that leaf. An illustration is given in Figure 1. In the binary search tree of this figure, it is assumed that the left branch corresponds to the heuristic choice. Discrepancy 0 (in Figure 1.a thus corresponds to the first leaf found by depth-first search. The leafs visited for discrepancy 1 (in Figure 1.b correspond to all leafs that are reached when exactly one right branch is chosen, and so on.

Note that the 'Local Branching' search strategy in the context of integer programming resembles limited discrepancy search [Fischetti and Lodi, 2003]. Namely, given a current solution x^* and an integer k, Local Branching creates a subproblem in which k variables are allowed to deviate from the value taken in x^*. The solutions considered in that subproblem correspond to those having discrepancy 0 up to k, where x^* serves as the heuristic to be followed.

3 Semidefinite Relaxations in Constraint Programming

There exist two streams of research that combine semidefinite relaxations and constraint programming. The first stream considers constraint programming models of an arbitrary form, and assumes that any suitable semidefinite relaxation may be applied, depending on the problem at hand. It is in the same spirit as the 'optimization constraints' mentioned before, and this stream will be the main focus of the remainder of this chapter. The second stream considers constraint programming models of a specific form, and applies a dedicated semidefinite relaxation that maps exactly onto

the format of the constraint programming model. Examples are given at the end of this section.

In this section, we will first discuss how suitable semidefinite relaxations may be found for arbitrary constraint programming models. Unfortunately, it is not straightforward to obtain a computationally efficient semidefinite program that provides a tight solution for any given constraint programming model. However, for a number of combinatorial optimization problems such semidefinite relaxations do exist, see Laurent and Rendl [2004] and Chapter 27 'Combinatorial Optimization' of this volume for an overview. If such semidefinite relaxations are not available, we need to build up a relaxation from scratch. One possible way is to apply the generic framework by Laurent, Poljak, and Rendl [1997] that will be described in Section 3.1.

After the discussion of building semidefinite relaxations, we consider the application of such relaxations inside optimization constraints in Section 3.2. Next, we discuss how the solution to the semidefinite relaxation can be applied to guide the search process in Section 3.3. Finally, in Section 3.4 we discuss other applications to combining semidefinite relaxations and constraint programming, that are in the second stream mentioned above.

3.1 Building a Semidefinite Relaxation

Consider a constraint programming model consisting of a set of variables $x_1, \ldots, x_n$, a set of constraints and an objective function. If the domains $D(x_1), \ldots, D(x_n)$ are not binary, we first transform the variables and their domains into corresponding binary variables y_{ij} for $i = 1, \ldots, n$ and $j \in D(x_i)$:

$$\begin{aligned} x_i = j &\Leftrightarrow y_{ij} = 1, \\ x_i \neq j &\Leftrightarrow y_{ij} = 0. \end{aligned} \tag{5}$$

We will use the binary variables y_{ij} to construct a semidefinite relaxation, following Laurent, Poljak, and Rendl [1997]. Of course, the transformation has consequences for the constraints also, which will be discussed below. The method of Laurent et al. [1997] to transform a model with binary variables into a semidefinite relaxation is the following. For a positive integer N, let $\mathbf{d} \in \{0,1\}^N$ be a vector of binary variables representing all variables y_{ij} for $i = 1, \ldots, n$ and $j \in D(x_i)$. Construct the $(N+1) \times (N+1)$ variable matrix X as

$$X = \begin{pmatrix} 1 \\ \mathbf{d} \end{pmatrix} (1 \; \mathbf{d}^\mathsf{T}) = \begin{pmatrix} 1 & \mathbf{d}^\mathsf{T} \\ \mathbf{d} & \mathbf{d}\mathbf{d}^\mathsf{T} \end{pmatrix}. \tag{6}$$

We then constrain X to satisfy

$$X \succeq 0 \tag{7}$$

$$X_{ii} = X_{0i} \; \forall i \in \{1, \ldots, N\} \tag{8}$$

where the rows and columns of X are indexed from 0 to N. Condition (8) expresses the fact that $d_i^2 = d_i$, which is equivalent to $d_i \in \{0,1\}$. Note however that the latter constraint is relaxed by requiring X to be positive semidefinite.

The matrix X contains the variables to model our semidefinite relaxation. Obviously, the diagonal entries (as well as the first row and column) of this matrix represent the binary variables from which we started. Using these variables, we need to express (a part of) the original constraints as our semidefinite relaxation.

In case the binary variables are obtained from transformation (5), not all constraints may be trivially transformed accordingly. Especially because the original constraints may be of any form. The same holds for the objective function. On the other hand, as we are constructing a relaxation, we may choose among the set of constraints an appropriate subset to include in the relaxation. Moreover, the constraints itself are allowed to be relaxed.

In most practical cases, however, the suitability of applying an SDP relaxation heavily relies on the problem at hand. Instead of applying the above framework, a practitioner may be more interested in identifying a combinatorial substructure of the problem for which a known SDP relaxation exists and can be readily applied.

3.2 Semidefinite Relaxation as Optimization Constraint

Similar to linear programming relaxations, semidefinite programming relaxations can be embedded inside optimization constraints. As discussed before, the role of such a constraint is twofold. First, the relaxation can be used to improve the bound on the variable representing the corresponding objective. Second, the relaxation can be applied to perform cost-based domain filtering.

So far, semidefinite relaxations have only been applied in a constraint programming context to improve the bound on the objective. Cost-based domain filtering utilizing the semidefinite relaxation has not yet been pursued. Nevertheless, Helmberg [2000] has introduced variable fixing procedures for semidefinite programming, while Fleischman and Poggi de Aragão [2010] have presented similar variable fixing procedures for unconstrained quadratic programs. In principle, these methods could be readily applied as filtering algorithms inside an optimization constraint. However, there are some practical hurdles that may prevent such filtering to be worthwhile. Perhaps the most important one is the issue of incrementality. Recall that domain filtering algorithms are typically invoked many times during the solving process. Since it is not straightforward to efficiently re-use data from one propagation event to the next when using semidefinite programming, the relaxations typically have to be recomputed from scratch at each event. This can be very time-consuming, and may not offset the extra amount of search space reduction that can be gained from it.

3.3 Semidefinite Relaxation as Search Heuristic

Several works on optimization constraints have applied the embedded linear relaxation not only for cost-based filtering, but also to define a search heuristic. For example, Focacci, Lodi, and Milano [1999a] use the solution to the linear relaxation as a branching heuristic for Traveling Salesman Problems. As another example, Milano and Van Hoeve [2002] apply reduced costs following from a linear relaxation as a

search heuristic. Leahu and Gomes [2004] investigate the quality of linear relaxations as a search heuristic in detail.

Van Hoeve [2006, 2003] proposes to use the solution to an SDP relaxation as a heuristic to guide the CP search. In general, the solution to a semidefinite relaxation yields fractional values for its variable matrix. These fractional values can serve as an indication for the values that the original constraint programming variables take in a solution. Consider for example the matrix X of equation (6) above, and suppose it is obtained from non-binary original variables by transformation (5). Assume that variable X_{ii} corresponds to the binary variable y_{jk} (for some integer j and k), which corresponds to $x_j = k$, where x_j is a constraint programming variable and $k \in D(x_j)$. If variable X_{ii} is close to 1, then also y_{jk} is supposed to be close to 1, which corresponds to assigning $x_j = k$.

Hence, the variable and value selection heuristics for the constraint programming variables are based upon the fractional solution values of the corresponding variables in the semidefinite relaxation. A natural variable selection heuristic is to select first the constraint programming variable for which the corresponding variable in the semidefinite relaxations has a fractional solution that is closest to the corresponding integer solution. As value selection heuristic, we then select first the corresponding suggested value. As an alternative, we can design a randomized branching heuristic, in which a selected variable (or value) is accepted with probability proportional to the corresponding fractional value in the semidefinite relaxation.

Note that not all semidefinite relaxations may offer a precise mapping between a variable-value pair in the constraint programming model and a variable in the semidefinite relaxation. We will see an example of this in Section 6, where the constraint programming variables $x_1, \ldots, x_n$ all have domains $\{1, \ldots, k\}$ for some integer $k \geq 1$. The semidefinite relaxation of that application uses variables Z_{ij} that represent whether x_i and x_j are assigned the same value, irrespective of the eventual value. In such situations, we can still apply the solution to the semidefinite relaxation as a branching heuristic, for example by branching on $(x_i = x_j)$ versus $(x_i \neq x_j)$. However, additional branching decisions may be necessary to eventually assign a value to each variable.

3.4 Semidefinite Relaxations for Restricted Constraint Programming Problems

Several works have applied semidefinite relaxations to constraint programming problems of a specific form, including Boolean satisfiability and weighted constraint satisfaction problems (CSPs). The *Boolean satisfiability problem*, or SAT problem, consists of a set of Boolean variables and a conjunction of clauses, each of which is a disjunction of literals. The conjunction of clauses is called a formula. Each literal is a logical variable (x) or its negation ($\overline{x}$). The SAT problem is to determine whether there exists a variable assignment that makes the formula true (i.e., each clause is true). The k-SAT problem represents the SAT problem where the clauses are constrained to have length equal to k. The (2+p)-SAT problem is a SAT problem in which a fraction p of the clauses is defined on three literals, while a fraction $(1-p)$

is defined on two literals. Clearly, the SAT problem is a constraint programming problem of a restricted form, i.e., the variables take Boolean domains and constraints are restricted to take the form of clauses.

MAX-SAT is the optimization version of SAT. Given a formula, we want to maximize the number of simultaneously satisfied clauses. Given an algorithm for MAX-SAT we can solve SAT, but not vice-versa, therefore MAX-SAT is more complex than SAT. The distinction becomes obvious when considering the case when the clauses are restricted to two literals per clause (2-SAT): 2-SAT is solvable in linear time, while MAX-2-SAT is NP-hard [Garey and Johnson, 1979].

Goemans and Williamson [1995] were the first to apply semidefinite relaxations to the MAX-2-SAT and MAX-SAT problems for obtaining approximation algorithms with provable performance ratios. Other works following this approach include Karloff and Zwick [1997] for MAX-3-SAT and Halperin and Zwick [2001] for MAX-4-SAT. Warners [1999] applies semidefinite relaxations for MAX-2-SAT problems within a DPLL-based SAT solver (see also De Klerk and Warners [2002]). The experimental results show that the semidefinite relaxations provide very tight objective values, and are applicable in practice in terms of computational efficiency. De Klerk et al. [2000] study semidefinite relaxations for general SAT problems, while De Klerk and Van Maaren [2003] consider semidefinite relaxations for (2+p)-SAT problems. Anjos [2005] introduces improved semidefinite relaxations to SAT problems.

Lau [2002] considers *weighted CSPs*, that generalize the MAX-2-SAT problem above in two ways. First, variables can have arbitrary finite domains. Second, even though the constraints are still defined on two variables only, now a weight is associated to each constraint. That is, a constraint C on two variables x and y is defined as $C \subseteq D(x) \times D(y)$, and has an associated weight (a natural number). The goal is to maximize the sum of the weights of the satisfied constraints. Lau [2002] represents such weighted CSPs as a quadratic integer program, and formulates a corresponding semidefinite relaxation. He applies a randomized rounding procedure to prove worst-case bounds for fixed variable domain sizes 2 and 3. He also provides computational results, comparing his approach to greedy and randomized local search procedures.

A recent line of research studies problems of the form MAX k-CSP, where variables can take an arbitrary finite domain, each constraint is defined on k variables, and the goal is to maximize the number of satisfied constraints. Zwick [1998] also applies semidefinite relaxations to a similar problem referred to as MAX 3-CSP, but in that case the domains are Boolean, constraints take the form of a Boolean function on at most three variables, and each constraint has an associated weight. Therefore, these problems are more similar to weighted CSPs than to the recent interpretation of MAX k-CSP. Most of the recent works for MAX k-CSP consider the approximability of such problems, and apply semidefinite relaxations to obtain certain approximation ratios, e.g., [Charikar et al., 2009], [Guruswami and Raghavendra, 2008] and [Raghavendra, 2008]. Other works study the integrality gap following from the semidefinite relaxation, for similar constraint satisfaction problems, e.g., [Raghavendra and Steurer, 2009a,b].

4 Application to the Maximum Weighted Stable Set Problem

As a first application of integrating constraint programming and semidefinite programming we consider the hybrid approach of Van Hoeve [2006, 2003] for the stable set problem. The motivation for this work is based on two complementary observations: *i)* a standard CP approach can have great difficulty finding a good solution, let alone proving optimality, and *ii)* SDP relaxations may provide a good starting solution, but the embedding of SDP inside a branch-and-bound framework to prove optimality can be computationally too expensive.

4.1 Problem Description and Model Formulations

Recall from Section 2 that the weighted stable set problem is defined on an undirected weighted graph $G = (V,E)$, with 'weight' function $w : E \to \mathbb{R}$. Without loss of generality, we assume all weights to be non-negative. The problem is to find a subset of vertices $S \subseteq V$ of maximum total weight, such that no two vertices in S are joined by an edge in E.

The constraint programming model applied by Van Hoeve [2006] uses binary variables x_i representing whether vertex i is in S ($x_i = 1$) or not ($x_i = 0$), for all $i \in V$. The CP model is then formulated as an integer linear programming model:

$$\begin{aligned} \max \; & \sum_{i=1}^{n} w_i x_i \\ \text{s.t. } & x_i + x_j \leq 1, \; \forall (i,j) \in E, \\ & x_i \in \{0,1\}, \; \forall i \in V. \end{aligned} \tag{9}$$

A, by now classical, semidefinite relaxation for the maximum-weight stable set problem was introduced by Lovász [1979]. The value of that relaxation is called the theta number, and is denoted by $\vartheta(G)$ for a graph G. The theta number arises from several different formulations, see Grötschel, Lovász, and Schrijver [1988]. Van Hoeve [2006] uses the formulation that has been shown to be computationally most efficient among those alternatives [Gruber and Rendl, 2003]. Let us introduce that particular formulation (called ϑ_3 by Grötschel, Lovász, and Schrijver [1988]). Let $\mathbf{x} \in \{0,1\}^n$ be the vector of binary variables representing a stable set, where $n = |V|$. Define the $n \times n$ matrix $X = \xi\xi^\mathsf{T}$ where the vector ξ is given by

$$\xi_i = \frac{\sqrt{w_i}}{\sqrt{\sum_{j=1}^{n} w_j x_j}} x_i$$

for all $i \in V$. Furthermore, let the $n \times n$ cost matrix U be defined as $U_{ij} = \sqrt{w_i w_j}$ for $i, j \in V$. Observe that in these definitions we exploit the fact that $w_i \geq 0$ for all $i \in V$. The following semidefinite program

$$\begin{aligned} \max \quad & \operatorname{tr}(UX) \\ \text{s.t. } & \operatorname{tr}(X) = 1 \\ & X_{ij} = 0, \; \forall (i,j) \in E \\ & X \succeq 0 \end{aligned} \tag{10}$$

provides the theta number of G, see Grötschel et al. [1988]. Here $\mathrm{tr}(X)$ represent the trace of matrix X, i.e., the sum of its main diagonal elements. When (10) is solved to optimality, the diagonal element X_{ii} can be interpreted as an indication for the value that x_i ($i \in V$) takes in an optimal solution to the problem.

4.2 Evaluation of the Hybrid Approach

In the hybrid approach of Van Hoeve [2006], the semidefinite relaxation is solved once, at the root node of the search tree. The associated objective value is applied to bound the initial domain of the variable representing the objective in the CP model. Van Hoeve [2006] does not apply any additional cost-based domain filtering. Instead, the fractional solution to the semidefinite relaxation is applied as a variable and value selection heuristic. The variable x_i with the highest corresponding solution for X_{ii} will be selected first, and value 1 will be assigned to it first. Since the semidefinite relaxation often provides a very accurate variable-value selection heuristic, Van Hoeve [2006] applies limited discrepancy search to traverse the resulting search tree.

This approach is applied to solve random problem instances as well as structured instances arising from coding theory, and maximum clique problems. The hybrid approach is compared to a pure CP solver with a lexicographic variable selection strategy, choosing value 1 first. In almost all cases, the SDP solution provides a very accurate branching strategy, and the best solution is found at a very low discrepancy (recall that limited discrepancy search is applied). In fact, in many cases the tight bound obtained by the SDP relaxation suffices to prove optimality of the solution found with the SDP-based heuristic.

5 Accuracy of Semidefinite Relaxations as Search Heuristic

In this section, the accuracy of the semidefinite relaxation as a search heuristic is investigated in more detail. A similar investigation has been performed for linear programming relaxations by Leahu and Gomes [2004]. They identify that the heuristic quality of the LP solution is dependent on structural combinatorial properties of the problem at hand, which in their experiments is measured by the 'constrainedness' of the problem. More specifically, the problems that they consider, i.e., Latin Square completion problems, exhibit an easy-hard-easy phase transition when the problem becomes more constrained. The 'typically hardest' problem instances are those that originate from the critically constrained region corresponding to the easy-hard-easy phase transition. For instances that are outside this region, and that are typically less hard to solve, the linear programming relaxation provides quite accurate values. However, for problem instances from within the phase transition region, the information quality as search heuristic shows a sharp decrease. In other words, the quality of the relaxation degrades exactly for those instances it is most needed.

In light of these results, Gomes, Van Hoeve, and Leahu [2006] study the accuracy of semidefinite relaxations as search heuristic. One of the main motivations for this was to investigate whether semidefinite relaxations provide more robust search

heuristics, and of higher quality, than linear programming relaxations. The particular problem of study in [Gomes et al., 2006] is MAX-2-SAT, and SDP relaxations are contrasted with LP relaxations and complete (exact) solution methods for this problem.

A related work is that of Warners [1999] and De Klerk and Warners [2002], who propose and analyze a MAX-2-SAT solver that employs a semidefinite relaxation as well. However, they do not apply the SDP solution as a search heuristic. Instead, the branching rule is to choose first the variable with the maximal occurrence in the longest clauses. De Klerk and Van Maaren [2003] present another related work, in which the accuracy of the semidefinite relaxation to detect unsatisfiability for the (2+p)-SAT problem is experimentally evaluated. Finally, we note that Cheriyan et al. [1996] also apply linear programming and rounding techniques to solve MAX-2-SAT problems.

5.1 Problem Description and Model Formulations

Let the MAX-2-SAT problem consist of Boolean variables $x_1, x_2, \ldots, x_n$ and clauses $C_1, C_2, \ldots, C_m$ on these variables. We consider the following ILP formulation for the MAX-SAT problem from Goemans and Williamson [1994]. With each clause C_j we associate a variable $z_j \in \{0,1\}$, for $j = 1, \ldots, m$. Value 1 corresponds to the clause being satisfied and 0 to the clause not being satisfied. For each Boolean variable x_i we associate a corresponding variable y_i in the ILP, for $i = 1, \ldots, n$. Variable y_i can take the values 0 and 1, corresponding to x_i being false or true, respectively. Let C_j^+ be the set of indices of positive literals that appear in clause C_j, and C_j^- be the set of indices of negative literals (i.e., complemented variables) that appear in clause C_j. The problem can then be stated as follows:

$$\begin{aligned} \max \ & \sum_{j=1}^{m} z_j \\ \text{subject to } & \sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) \geq z_j, \ \forall j \in \{1, \ldots, m\} \\ & y_i, z_j \in \{0,1\}, \qquad \forall i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}. \end{aligned}$$

This model ensures that a clause is true only if at least one of the variables that appear in the clause has the value 1. Since we maximize $\sum_{j=1}^{m} z_j$ and z_j can be set to 1 only when clause C_j is satisfied, it follows that the objective function counts the number of satisfied clauses. By relaxing the integrality constraint, we obtain an LP relaxation for the MAX-SAT problem. This ILP formulation is equivalent to the ILP used in Xing and Zhang [2005] to compute the lower bound and to the ILP solved at each node by the MAX-SAT branch-and-cut algorithm in Joy et al. [1997].

Observe that there exists a trivial way to satisfy all the clauses by setting each variable y_i to 0.5. Using this assignment, the sum of literals for each clause is exactly 1, hence the clause can be satisfied and the objective function is equal to the number of clauses. The value 0.5 is not at all informative, lying half way between 0 and 1,

it gives no information whether the corresponding Boolean variable should be set to true or false. As the problem becomes more constrained (i.e., the number of clauses increases) the corresponding 2-SAT problem is very likely to be unsatisfiable, hence any variable assignment different than 0.5 would lead to a less than optimal objective value. Naturally, the LP solver finds the highest possible objective value (i.e., the number of clauses) when setting all variables to 0.5.

Gomes et al. [2006] apply the following semidefinite relaxation of MAX-2-SAT that was introduced by Goemans and Williamson [1995]. To each Boolean variable x_i ($i = 1,\ldots,n$), we associate a variable $y_i \in \{-1,1\}$. Moreover, we introduce a variable $y_0 \in \{-1,1\}$. We define x_i to be true if and only if $y_i = y_0$, and false otherwise.

Next, we express the truth value of a Boolean formula in terms of its variables. Given a formula c, we define its *value*, denoted by $v(c)$, to be 1 if the formula is true, and 0 otherwise. Hence,

$$v(x_i) = \frac{1+y_0y_i}{2}$$

gives the value of a Boolean variable x_i as defined above. Similarly,

$$v(\overline{x}_i) = 1 - v(x_i) = \frac{1-y_0y_i}{2}.$$

The value of the formula $x_i \vee x_j$ can be expressed as

$$\begin{aligned} v(x_i \vee x_j) &= 1 - v(\overline{x}_i \wedge \overline{x}_j) = 1 - v(\overline{x}_i)v(\overline{x}_j) = 1 - \frac{1-y_0y_i}{2}\frac{1-y_0y_j}{2} \\ &= \frac{1+y_0y_i}{4} + \frac{1+y_0y_j}{4} + \frac{1-y_iy_j}{4}. \end{aligned}$$

The value of other clauses can be expressed similarly. If a variable x_i is negated in a clause, then we replace y_i by $-y_i$ in the above expression.

Now we are ready to state the integer quadratic program for MAX-2-SAT:

$$\begin{aligned} &\max \sum_{c \in C} v(c) \\ &\text{s.t.} \quad y_i \in \{-1,1\} \ \forall i \in \{0,1,\ldots,n\}. \end{aligned} \tag{11}$$

It is convenient to rewrite this program as follows. We introduce an $(n+1)\times(n+1)$ matrix Y, such that entry Y_{ij} represents y_iy_j (we index the rows and columns of Y from 0 to n). Then program (11) can be rewritten as

$$\begin{aligned} &\max \operatorname{tr}(WY) \\ &\text{s.t.} \quad Y_{ij} \in \{-1,1\} \ \forall i,j \in \{0,1,\ldots,n\}, i \neq j, \end{aligned} \tag{12}$$

where W is an $(n+1)\times(n+1)$ matrix representing the coefficients in the objective function of (11). For example, if the coefficient of y_iy_j is w_{ij}, then $W_{ij} = W_{ji} = \frac{1}{2}w_{ij}$.

The final step consists in relaxing the conditions $Y_{ij} \in \{-1,1\}$ by demanding that Y should be positive semidefinite and $Y_{ii} = 1 \ \forall i \in \{0,1,\ldots,n\}$. The semidefinite relaxation of MAX-2-SAT can then be formulated as

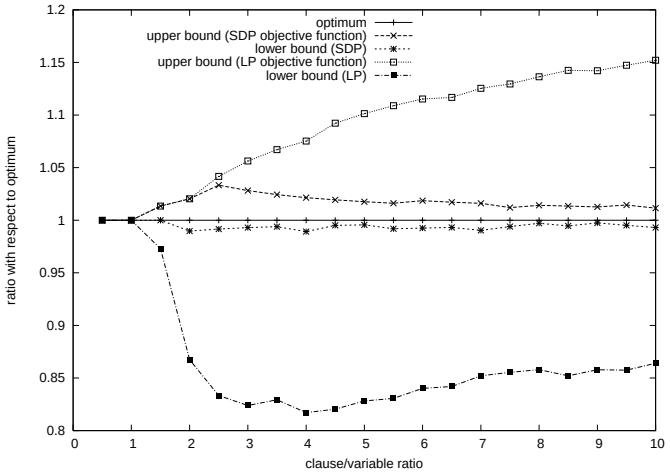


Fig. 2. Lower and upper bounds based on LP and SDP relaxations for MAX-2-SAT instances of increasing constrainedness.

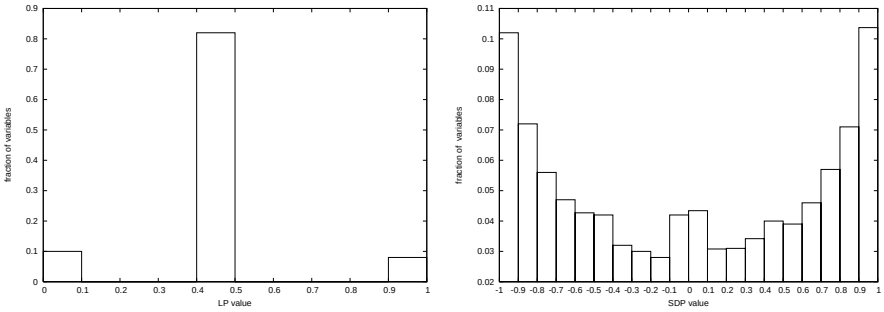
$$\begin{aligned} &\max \operatorname{tr}(WY) \\ &\text{s.t.} \quad Y_{ii} = 1 \quad \forall i \in \{0,1,\ldots,n\}, \\ &\qquad Y \succeq 0. \end{aligned} \tag{13}$$

Program (13) provides an upper bound on the solution to MAX-2-SAT problems. Furthermore, the values Y_{0i}, representing $y_0 y_i$, correspond to the original Boolean variables x_i ($i = 1,\ldots,n$). Namely, if Y_{0i} is close to 1, variable x_i is 'likely' to be true. Similarly, if Y_{0i} is close to -1, variable x_i is 'likely' to be false.

5.2 Experimental Evaluation

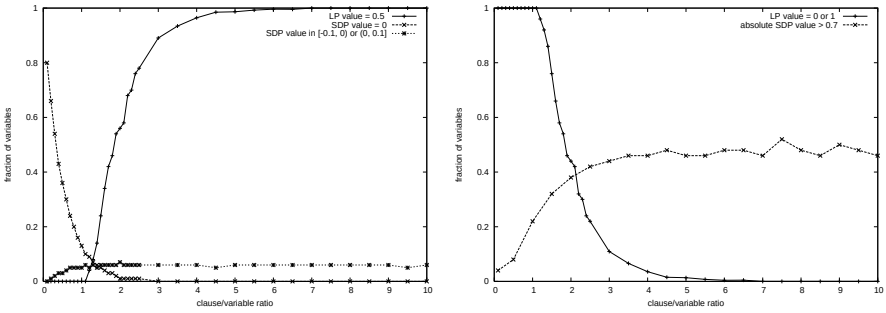
We next describe the experiments performed by Gomes et al. [2006] on randomly generated MAX-2-SAT instances. First, we consider the objective value of the LP and SDP relaxations across different constrainedness regions of the problem. Figure 2 presents the relative lower and upper bounds obtained by LP and SDP, for MAX-2-SAT instances on 80 variables where the constrainedness ratio (number of clauses over the number of variables) ranges from 0.5 to 10. The figure compares the ratio of these bounds to the optimal solution value. The lower bounds for LP and SDP are obtained by rounding the suggested fractional solution to the closest integer. As is clear from this plot, the semidefinite relaxation provides far better bounds than the linear relaxation. This confirms the observations made by Warners [1999] for MAX-2-SAT problems.

Next we consider the strengths of the relaxations in terms of heuristic guidance. This is done by first measuring the 'fractionality' of the solution to the relaxation. Namely, if a solution value of the relaxation is in the midpoint between the two integer endpoints (i.e., 0.5 for LP and 0.0 for SDP), we consider the suggested value



a. LP relaxation solution values b. SDP relaxation solution values

Fig. 3. Distribution of the values returned by the LP relaxation (a) and the SDP relaxation (b), averaged over instances with clause over variable ratio ranging from 0.5 to 10.



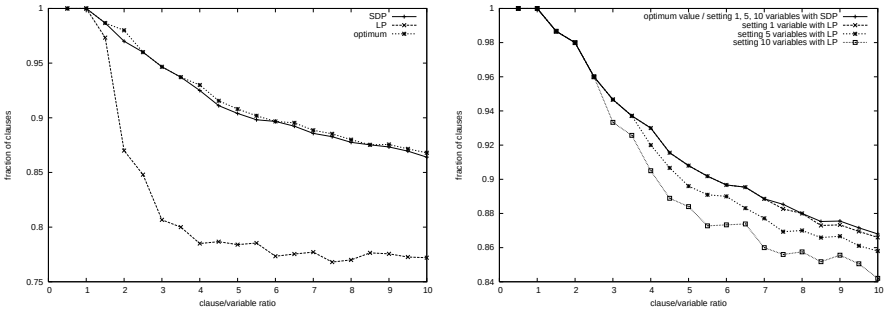
a. uninformative solution values b. informative solution values

Fig. 4. Fraction of variables having (a) uninformative value 0.5 computed by LP and 0 or smaller than 0.1 in absolute value computed by SDP, and (b) informative value 0 or 1 computed by the LP relaxation and above 0.7 in absolute value by the SDP relaxation.

uninformative. On the other hand, if the suggested value is close to an integer endpoint, we consider that value informative.

Figure 3 presents the distribution of the solution values returned by the LP and SDP relaxations, respectively. In Figure 3.a the distributions for the LP relaxation is given, clearly indicating that most values (more than 80%) are uninformative. In contrast, the distribution of the solution values for the SDP relaxation indicates that the majority of the suggested values is close to an integer value, and are much more informative. In this figure, the distribution averages the solutions over all instances where the ratio of the number of clauses over the number of variables ranges from 0.5 to 10.

Figure 4 depicts how the fractionality evolves with respect to the range of increasing constrainedness. In Figure 4.a the uninformative values are considered. It depicts the fraction of variables taking value 0.5 in the LP solution, and value 0, respectively in the intervals close to 0, i.e., [-0.1,0), (0,0.1], for the SDP solution.



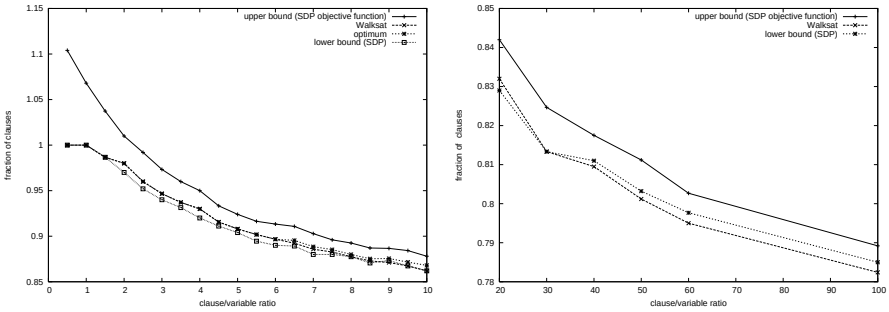
a. setting 84% of the variables b. setting 1, 5, and 10 variables

Fig. 5. Change in the number of satisfied clauses as we set (a) 84% of the variables and (b) 1, 5 and 10 variables using the LP and SDP solutions.

When the number of clauses increases (clause/variable ratio greater than 1), the LP solution values quickly become more uninformative. The SDP solution values of 0 show exactly the opposite, while the SDP values close to 0 remain fairly constant. In Figure 4.b considers the informative values, i.e., LP solution values 0 or 1, and SDP values close to -1 or 1 (with absolute value more than 0.7 to be precise). Again, the LP solution values quickly loose their informative quality beyond clause/variable ratio 1. At the same time, the informative quality of the SDP solution values increases up to that point, and remains constant afterward.

The following set of experiments investigate the actual accuracy of the heuristics, by evaluating the optimal objective function value when a number of variables is fixed to the suggested heuristic value, where the variables closest to integrality are chosen to be fixed first. In Figure 5.a, 84% of the variables are fixed according to the heuristic, and the exact solver Maxsolver is used to compute an optimal solution, completing the partial assignment. Below 84%, the SDP heuristic choice always provided an optimal solution, while at 84%, the SDP heuristic deviates slightly from the optimal solution. The LP solution values perform much worse, as is clear from the figure. In Figure 5.b, this is shown in more detail, as it indicates the effect of setting 1, 5, and 10 variables to the suggested heuristic value. Even when assigning only one variable, the LP heuristic already deviates from the optimal solution for higher clause/variable ratios.

Finally, the solution obtained by the SDP heuristic is compared to the solution obtained by Walksat, a state-of-the-art satisfiability solver based on local search. The results are indicated in Figure 6. It can be observed that Walksat and the SDP heuristic provide comparable solutions, and when the clause/variable ratio is relatively low (Figure 6.a Walksat usually finds slightly better solutions than the heuristic guided by the SDP solution. For larger clause/variable ratio, however, the solutions obtained with the SDP heuristic outperform Walksat. This is an interesting result, because for these problems, the time to solve the SDP relaxation was considerably smaller than



a. small clause/variable ratio b. large clause/variable ratio

Fig. 6. Comparing SDP and Walksat as a lower bound.

the time taken by Walksat, while at the same time the SDP relaxation also provides an upper bound.

In summary, these results further motivate the use of semidefinite relaxations as search heuristic for exact solving methods such as constraint programming.

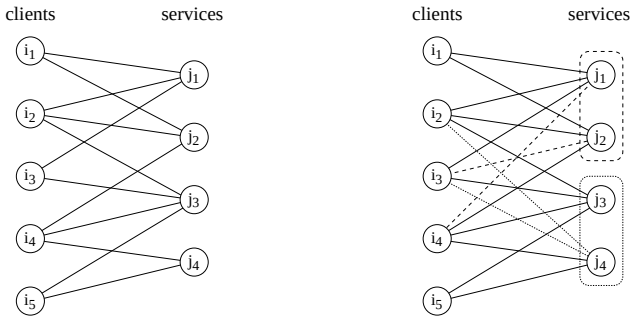
6 Telecommunications Application: Biclique Completion

Gualandi [2009] presents a hybrid semidefinite and constraint programming approach to a particular 'biclique completion problem', arising in the context of telecommunications. Let us first describe the application, which was first introduced by Faure, Chrétienne, Gourdin, and Sourd [2007].

Traditional communication services are mostly 'unicast', in which two nodes of a network are communicating between each other (for example a telephone conversation). In contrast, 'multicast' services interconnect multiple nodes in the network (for example a video-conference meeting). Specific efficient protocols for multicast services have been developed, and it is expected that the demand for multicast communication services will grow rapidly in the near future.

If we would choose to handle the multicast services optimally on an individual level, we would have to design and maintain a specific network configuration for each individual service, while ensuring global optimality of the network usage and provided services. Such a fine-grained individual approach is far from practical, as argued by [Faure et al., 2007]. Instead, multicast services are usually aggregated in clusters, and each cluster is considered as one meta-multicast service. For example, if several customers share a large number of services, they can be grouped together and use a network configuration that provides the information of all joint services to all customers. As a consequence, unnecessary information is sent between those customers and services in the cluster for which no relationship is required.

In [Faure et al., 2007], it is proposed to cluster the multicast sessions in such a way that the amount of unnecessary information sent through the network is limited,



a. Bipartite graph representation b. Optimal solution with two bicliques

Fig. 7. Clustering multicast services.

where this amount is measured by counting the number of client/service pairs between which unnecessary information is sent. An example is given in Figure 7. In this example, we are given a set of multicast services $I = \{i_1, i_2, i_3, i_4\}$ and a set of clients $J = \{j_1, j_2, j_3, j_4, j_5\}$. In Figure 7.a, the services and clients are depicted in a bipartite graph in which an edge (i, j) represents that client j requires service i. For example, client j_1 requires services i_1 and i_2. Figure 7.b represents the clustering of the services into two sets $\{i_1, i_2\}$ and $\{i_3, i_4\}$. This solution implies that unnecessary information is sent between the pairs (i_1, j_4), (i_2, j_3), (i_4, j_2) and (i_4, j_3), as indicated by the dashed edges in the figure. The total 'cost' of this solution is therefore 4. If we assume that the maximum number of clusters is 2, this solution is optimal.

Faure et al. [2007] also suggest other variants of the problem that consider more fine-grained measures for the information that is sent through the network. For example, one can take into account path lengths or capacity of certain connections. Some variants can be encoded as weighted versions of the problem. A hybrid semidefinite and constraint programming approach for such weighted biclique completion problems was studied by Gualandi and Malucelli [2010].

6.1 Problem Description

The specific variant considered by Gualandi [2009] is described more formally as follows [Faure et al., 2007]. Let $G = (I, J, D)$ be an undirected bipartite graph where vertices I represent a set of (multicast) services, vertices J represent a set of clients, and edge set $D \subset I \times J$ represents the required demands.

We define p clusters $T_1, \ldots, T_p$ as follows. A cluster $T_k = (I_k, J_k, D_k)$ is a subgraph of G where $I_k \subset I$, $J_k \subset J$, and $D_k = (I_k \times J_k) \cap D$. Let $K = \{1, \ldots, p\}$ denote the index set of the clusters. Since the problem will treat each cluster as a single multicast group, the cluster will in practice correspond to a biclique (a complete bipartite graph) on the vertex sets I_k and J_k. The cost of cluster T_k is $|I_k| \cdot |J_k| - |D_k|$, corresponding to the amount (number) of unnecessary information sent to the clients.

Given a number $p \in \{1, \ldots, |J| - 1\}$, the *multicast partition problem* consists of finding p clusters $T_1, \ldots, T_p$ such that $I_1, \ldots, I_p$ induce a partition of I, and the total

cost

$$\sum_{k=1}^{p} |I_k| \cdot |J_k| - |D_k|$$

is minimized. Given the nature of the problem, it is also referred to in [Faure et al., 2007] and [Gualandi, 2009] as a 'biclique completion' problem. Faure et al. [2007] show that the problem is NP-hard, even for $p = 2$.

6.2 Constraint Programming Model

The constraint programming model employed by Gualandi [2009] relies on a custom-made global constraint, that will be applied to characterize each cluster and the associated cost. Let $G = (S,T,E)$ be an undirected bipartite graph, and let $\overline{G} = (S,T,\overline{E})$ be the complement graph of G, i.e., $\overline{E} = (S \times T) \setminus E$. We introduce two set variables X and Y with initial domains $D(X) = [\varnothing, S]$, $D(Y) = [\varnothing, T]$, and $0 \leq |X| \leq |S|$, $0 \leq |Y| \leq |T|$. Variable X represents a set of nodes on one 'shore' of the graph, and all neighbors of X will be accumulated in Y. Further, let c be an integer variable with $0 \leq c \leq \left|\overline{E}\right|$. It corresponds to the number of edges in the complement graph $\overline{G}$ that are necessary to create a biclique from the subgraph induced by X and Y. The specific global constraint, called 'one-shore induced quasi-biclique constraint' takes the form

$$\texttt{osi-biclique}(X,Y,c,G)$$

and states that

$$Y = \cup_{i \in X} N(i), \text{ and}$$
$$c = |X| \cdot |Y| - |F|,$$

where $F = (X \times Y) \cap E$. Here $N(i)$ denotes the set of neighbors of a vertex i, i.e., $N(i) = \{ j \mid (i,j) \in E \}$. Gualandi [2009] describes and implements specific domain filtering algorithms for this constraint.

We next build the constraint programming model for the original problem on a graph $G = (I,J,D)$ with multicast services I, clients J, and demands D as specified above. We first introduce set variables $X_1, \ldots, X_p$ and $Y_1, \ldots, Y_p$, where a pair (X_k, Y_k) represents the vertices of a cluster $T_k = (I_k, J_k, D_k)$. We further introduce an integer variable c_k representing the cost of cluster T_k, and a variable z representing the objective function. The model then becomes

$$\begin{aligned}
\min\ & z = \sum_{k=1}^{p} c_k \\
\text{s.t. } & \texttt{partition}(X_1, \ldots, X_p, I), \\
& \texttt{osi-biclique}(X_k, Y_k, c_k, G), && \forall k \in K, \\
& D(X_k) = [\varnothing, I], 0 \leq |X_k| \leq |I|, && \forall k \in K, \\
& D(Y_k) = [\varnothing, J], 0 \leq |Y_k| \leq |J|, && \forall k \in K, \\
& 0 \leq c_k \leq \left|\overline{E}\right|, && \forall k \in K, \\
& 0 \leq z \leq \left|\overline{E}\right|.
\end{aligned}$$

Recall that the `partition` constraint was defined in Section 2.

6.3 Semidefinite Programming Model

The semidefinite relaxation for this problem proposed by Gualandi [2009] is closely related to the semidefinite relaxations for the MAX CUT problem [Goemans and Williamson, 1995] and the MAX k-CUT problem [Frieze and Jerrum, 1997]. Let us first describe the semidefinite relaxation for $p = 2$ and then extend this formulation for general p.

For $p = 2$, the objective of the multicast partition problem can be interpreted as minimizing the number of edges in the complement graph induced by the clusters $T_1 = (I_1, J_1, D_1)$ and $T_2 = (I_2, J_2, D_2)$. This is equivalent to maximizing the number of edges in the complement graph that are in the cut between the sets $(I_1 \cup J_1) \cap J_2$ and $(I_2 \cup J_1) \cap J_1$. That is, edges connected to clients that belong to both J_1 and J_2 are not considered in the cut.

For each vertex $i \in I$ we introduce a variable $x_i \in \{-1, 1\}$ representing whether i belongs to I_1 ($x_i = 1$) or to I_2 ($x_i = -1$). If two vertices $i, j \in I$ are in the same cluster, the product $x_i x_j$ equals 1, otherwise the product is equal to -1. We further introduce a variable $z_{ij} \in \{-1, 1\}$ for every edge $(i, j) \in \overline{E}$ representing whether (i, j) belongs to the cut ($z_{ij} = -1$) or not ($z_{ij} = 1$). Since an edge (i, j) does not belong to a cut if there exists a vertex $k \in N(j)$ such that i and k belong to the same cluster, or $x_i x_k = 1$, we have $z_{ij} = \max_{k \in N(j)} \{x_i x_k\}$. This relation can be linearized as $z_{ij} \geq x_i x_l$, for all $k \in N(j)$.

The multicast partition problem with $p = 2$ can now be formulated as the following quadratic program:

$$
\begin{aligned}
z_{\text{QP}} = \max \ & \tfrac{1}{2} \sum_{(i,j) \in \overline{E}} (1 - z_{ij}) \\
\text{s.t. } & z_{ij} \geq x_i x_k, && \forall (i, j) \in \overline{E}, k \in N(j), \\
& x_i \in \{-1, 1\}, && \forall i \in I, \\
& z_{ij} \in \{-1, 1\}, && \forall (i, j) \in \overline{E}.
\end{aligned}
$$

Note that the optimal solution value to the multicast partition problem is equal to $\left|\overline{E}\right| - z_{\text{QP}}$.

We next associate a unit vector $\mathbf{v}_i \in \mathbb{R}^{|I|+|J|}$ to each vertex $i \in I \cup J$, with the interpretation that i and j are in the same cluster if $\mathbf{v}_i \cdot \mathbf{v}_j = 1$. Let V be the matrix consisting of columns $\mathbf{v}_i$ for all $i \in I \cup J$, and let $Z = V^{\mathsf{T}} V$. The semidefinite relaxation thus becomes:

$$
\begin{aligned}
\max \ & \tfrac{1}{2} \sum_{(i,j) \in \overline{E}} (1 - Z_{ij}) \\
\text{s.t. } & Z_{ij} \geq Z_{ik}, && \forall (i, j) \in \overline{E}, k \in N(j), \\
& \operatorname{diag}(Z) = \mathbf{e}, \\
& Z \succeq 0.
\end{aligned}
$$

Here $\operatorname{diag}(Z)$ represents the vector formed by the diagonal elements of Z.

When the number of clusters p is more than 2, the model can be altered similar to the approach taken in [Frieze and Jerrum, 1997] for MAX k-CUT. We let $\mathbf{a}_1, \ldots, \mathbf{a}_p$

be unit vectors in $\mathbb{R}^{p-1}$ satisfying $\mathbf{a}_i \cdot \mathbf{a}_j = -\frac{1}{p-1}$ for $i, j \in \{1, \ldots, p\}, i \neq j$. These vectors represent the different clusters. For each vertex $i \in I$ we introduce a vector $\mathbf{x}_i$ taking its value in $\{\mathbf{a}_1, \ldots, \mathbf{a}_p\}$. That is, if two vertices i and j are in the same cluster we have $\mathbf{x}_i \cdot \mathbf{x}_j = 1$, otherwise we have $\mathbf{x}_i \cdot \mathbf{x}_j = -\frac{1}{p-1}$. We then introduce a variable $z_{ij} \in \{-\frac{1}{p-1}, 1\}$ for each edge $(i, j) \in \overline{E}$, representing whether (i, j) is in the cut ($z_{ij} = -\frac{1}{p-1}$) or not ($z_{ij} = 1$). Using these variables, the multicast partition problem for general p can be formulated as

$$\begin{aligned} \max\ & \tfrac{p-1}{p} \sum_{(i,j)\in\overline{E}} (1 - z_{ij}) \\ \text{s.t. } & z_{ij} \geq \mathbf{x}_i \cdot \mathbf{x}_k, && \forall (i,j) \in \overline{E}, k \in N(j), \\ & \mathbf{x}_i \in \{\mathbf{a}_1, \ldots, \mathbf{a}_p\}, && \forall i \in I, \\ & z_{ij} \in \{-\tfrac{1}{p-1}, 1\}, && \forall (i,j) \in \overline{E}. \end{aligned}$$

We can apply the same labeling technique as for the case $p = 2$ above to obtain the following semidefinite relaxation for general p:

$$\begin{aligned} z_{\text{SDP}} = \max\ & \tfrac{p-1}{p} \sum_{(i,j)\in\overline{E}} (1 - Z_{ij}) \\ \text{s.t. } & Z_{ij} \geq Z_{ik}, && \forall (i,j) \in \overline{E}, k \in N(j), \\ & \text{diag}(Z) = \mathbf{e}, \\ & Z_{ij} \geq -\tfrac{1}{p-1}, && \forall i, j \in I \cup J, i \neq j, \\ & Z \succeq 0. \end{aligned}$$

Observe that for $p = 2$, the vectors $\mathbf{a}_1$ and $\mathbf{a}_2$ are in fact scalars, i.e., $a_1 = -1$, $a_2 = 1$, in which case the latter model coincides precisely with the earlier model for $p = 2$.

The value of the semidefinite relaxation can be applied as a lower bound for the multicast partition problem using the relation $z^* \geq \left|\overline{E}\right| - \lfloor z_{\text{SDP}} \rfloor$, where z^* represents the optimal objective value for the multicast partition problem.

6.4 Evaluation of the Hybrid Approach

As stated above, Gualandi [2009] applies the semidefinite relaxation as a lower bound on the objective, but also to guide the constraint programming search process. To this end, entries in an optimal solution matrix Z^* to the semidefinite relaxation are interpreted as the likelihood that two vertices belong to the same cluster. That is, if Z^*_{ij} is close to $-\frac{1}{p-1}$, i and j are not likely to be in the same cluster, whereas they are if Z^*_{ij} is close to 1. The closer that Z^*_{ij} is to the midpoint $\frac{p}{2(p-1)}$, the more uncertain it is whether i and j belong to the same cluster.

The search heuristic first finds a pair of vertices (i, j) with $i, j \in I$, that are not yet assigned to any cluster, such that $|Z^*_{ij}| - \frac{p}{2(p-1)}$ is maximized. It then finds a variable X_k that contains at least one of these variables as a possible element. If $Z^*_{ij} > \frac{p}{2(p-1)}$

it will assign both i and j to X_k as branching decision. Otherwise, it will assign either i or j to X_k.

The overall hybrid approach has two variants. The first computes and exploits the semidefinite relaxation at each node in the search tree. The second variant only solves the semidefinite relaxation once at the root node, as in Van Hoeve [2006]. In the computational experiments reported by Gualandi [2009], these two approaches are compared against two other exact methods. The first uses a linearized quadratic integer programming model similar to [Faure et al., 2007], which is solved with IBM/ILOG CPLEX. The second is a pure constraint programming model, solved with Gecode, without the use of the semidefinite relaxation.

The computational results provide three main insights. First, the hybrid semidefinite and constraint programming approach is competitive to the integer programming approach (and in several cases better). Furthermore, in terms of efficiency, applying the semidefinite relaxation at each node in the search tree does not pay off. Instead, it was found that applying the semidefinite relaxation only once at the root node was more efficient. This strategy also outperformed the pure constraint programming approach.

7 Conclusion and Future Directions

In this chapter, we have described how constraint programming can be applied to model and solve combinatorial optimization problems, and how semidefinite programming has been successfully integrated into constraint programming. Specifically, we have shown that semidefinite relaxations can be a viable alternative to linear programming relaxations, that are commonly applied within optimization constraints. One of the main benefits of semidefinite relaxations in this context appears to be the accuracy when the solution to the semidefinite relaxation is applied as a search heuristic.

From a constraint programming perspective, arguably the most important question to be addressed is the application of semidefinite relaxations to design cost-based filtering algorithms for the variable domains, in addition to strengthening the bound on the objective that is currently done. Even though additional theory may be developed for this purpose, it is likely that specific algorithms must be designed and engineered, e.g., taking advantage of incremental data-structures, to make such application worthwhile in practice.

In order to make semidefinite programming relaxations more accessible to the constraint programming community, it would be worthwhile to investigate how the modeling and solving capabilities of constraint programming systems can be extended to facilitate this. For example, there exist various techniques to automatically create a linear programming relaxation from a given constraint programming model, see, e.g., [Refalo, 2000; Hooker, 2000, 2007]. Several systems, including the constraint programming solver Xpress Kalis [FICO, 2009] and the modeling language Zinc [Brand et al., 2008] provide an interface to embed and automatically exploit

linear programming relaxations. Having such functionality for semidefinite programming relaxations would be helpful, especially for designing hybrid methods.

In addition, it would be very useful to develop hybrid semidefinite programming and constraint programming approaches for more applications. Through studying more diverse applications, we not only have a chance of improving the state of the art in solving those, but we can also gain more insight in the theoretical and practical characteristics that make such approaches successful.

From a semidefinite programming perspective, it may be worthwhile to consider alternatives to the ILP-inspired branch-and-bound and branch-and-cut solving methods, especially because most semidefinite relaxations are still relatively expensive to compute. This chapter offers several options. For example, one may consider not solving a complete SDP relaxation at each search node, but rather at selected nodes of the search tree. Furthermore, alternative search strategies, as presented in this chapter, may be considered. And of course, the (cost-based) filtering algorithms may be applicable directly through variable fixing procedures inside an SDP-based solver.

Finally, another issue that deserves future investigations is the relationship between constraint programming, integer programming, and semidefinite programming when handling symmetry in combinatorial problems. There exists a vast literature on explicit symmetry breaking in constraint programming and integer programming, see Gent et al. [2006] and Margot [2010] for recent surveys. In contrast, certain types of symmetry breaking are implicitly accounted for in semidefinite relaxations; see for example Anjos and Vannelli [2008]. These complementary approaches could perhaps be combined very effectively.

In conclusion, even though several of the developments described in this chapter are still at an early stage, there clearly is a great potential for hybrid solution methods combining semidefinite programming and constraint programming.

Acknowledgement. I would like to thank Stefano Gualandi for helpful comments on an earlier draft of the chapter.

References

M.F. Anjos. An improved semidefinite programming relaxation for the satisfiability problem. *Mathematical Programming*, 102(3):589–608, 2005.

M.F. Anjos and A. Vannelli. Computing Globally Optimal Solutions for Single-Row Layout Problems Using Semidefinite Programming and Cutting Planes. *INFORMS Journal on Computing*, 20:611–617, 2008.

K.R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2): 179–210, 1999.

K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

S. Brand, G.J. Duck, J. Puchinger, and P.J. Stuckey. Flexible, Rule-Based Constraint Model Linearisation. In *Proceedings of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 4902 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2008.

M. Charikar, K. Makarychev, and Y. Makarychev. Near-optimal algorithms for maximum constraint satisfaction problems. *ACM Transactions on Algorithms*, 5(3):32–1—32–14, 2009.

J. Cheriyan, W.H. Cunningham, L. Tunçel, and Y. Wang. A Linear Programming and Rounding Approach to MAX 2-SAT. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:395–414, 1996.

R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

N. Faure, P. Chrétienne, E. Gourdin, and F. Sourd. Biclique completion problems for multicast network design. *Discrete Optimization*, 4:360–377, 2007.

FICO. Xpress-Kalis User guide, Fair Isaac Corporation, 2009.

M. Fischetti and A. Lodi. Local Branching. *Mathematical Programming*, 98(1–3):23–47, 2003.

D. Fleischman and M.V. Poggi de Aragão. Improved SDP Bounds on the Exact Solution of Unconstrained Binary Quadratic Programming. In *Optimization Days, Montreal*, 2010.

F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 1713 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 1999a.

F. Focacci, A. Lodi, M. Milano, and D. Vigo. Solving TSP through the integration of OR and CP techniques. *Electronic Notes in Discrete Mathematics*, 1:13–25, 1999b.

F. Focacci, A. Lodi, and M. Milano. Optimization-Oriented Global Constraints. *Constraints*, 7(3–4):351–365, 2002a.

F. Focacci, A. Lodi, and M. Milano. Embedding relaxations in global constraints for solving TSP and TSPTW. *Annals of Mathematics and Artificial Intelligence*, 34(4):291–311, 2002b.

F. Focacci, A. Lodi, and M. Milano. A hybrid exact algorithm for the TSPTW. *INFORMS Journal on Computing*, 14(4):403–417, 2002c.

A. Frieze and M. Jerrum. Improved Approximation Algorithms for MAX *k*-CUT and MAX BISECTION. *Algorithmica*, 18(1):67–81, 1997.

M.R. Garey and D.S. Johnson. *Computers and Intractibility*. Freeman, 1979.

I.P. Gent, K.E. Petrie, and J.-F. Puget. Symmetry in Constraint Programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 10. Elsevier, 2006.

C. Gervet. Conjunto: Constraint Logic Programming with Finite Set Domains. In *Proceedings of the International Logic Programming Symposium (ILPS)*, pages 339–358. MIT Press, 1994.

M.X. Goemans and D.P. Williamson. New $\frac{3}{4}$-Approximation Algorithms for the Maximum Satisfiability Problem. *SIAM Journal on Discrete Mathematics*, 7(4):656–666, 1994.

M.X. Goemans and D.P. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *Journal of the ACM*, 42(6): 1115–1145, 1995.

C.P. Gomes, W.J. van Hoeve, and L. Leahu. The Power of Semidefinite Programming Relaxations for MAX-SAT. In *Proceedings of the Third International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 3990 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2006.

M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Wiley, 1988.

G. Gruber and F. Rendl. Computational experience with stable set relaxations. *SIAM Journal on Optimization*, 13(4):1014–1028, 2003.

S. Gualandi. *k*-Clustering Minimum Biclique Completion via a Hybrid CP and SDP Approach. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 5547 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2009.

S. Gualandi and F. Malucelli. Weighted Biclique Completion via CP-SDP Randomized Rounding. In *Proceedings of the European Workshop on Mixed Integer Nonlinear Programming*, pages 223–230, 2010.

V. Guruswami and P. Raghavendra. Constraint Satisfaction over a Non-Boolean Domain: Approximation Algorithms and Unique-Games Hardness. In *Proceedings of the 11th International Workshop on Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques (APPROX)*, volume 1571 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2008.

E. Halperin and U. Zwick. Approximation Algorithms for MAX 4-SAT and Rounding Procedures for Semidefinite Programs. *Journal of Algorithms*, 40:184–211, 2001.

W.D. Harvey and M.L. Ginsberg. Limited Discrepancy Search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 607–615. Morgan Kaufmann, 1995.

C. Helmberg. Fixing Variables in Semidefinite Relaxations. *SIAM Journal on Matrix Analysis and Applications*, 21(3):952–969, 2000.

W.-J. van Hoeve. A hybrid constraint programming and semidefinite programming approach for the stable set problem. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2833 of *Lecture Notes in Computer Science*, pages 407–421. Springer, 2003.

W.-J. van Hoeve. Exploiting Semidefinite Relaxations in Constraint Programming. *Computers and Operations Research*, 33(10):2787–2804, 2006.

W.-J. van Hoeve and I. Katriel. Global Constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 6. Elsevier, 2006.

W.-J. van Hoeve, G. Pesant, and L.-M. Rousseau. On global warming: Flow-based soft global constraints. *Journal of Heuristics*, 12(4):347–373, 2006.

J. Hooker. *Logic-Based Methods for Optimization - Combining Optimization and Constraint Satisfaction*. Wiley, 2000.

J.N. Hooker. Operations Research Methods in Constraint Programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 15. Elsevier, 2006.

J.N. Hooker. *Integrated methods for optimization*. Springer, 2007.

S. Joy, J. Mitchell, and B. Borchers. A branch and cut algorithm for MAX-SAT and weighted MAX-SAT. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 35:519–536, 1997.

H. Karloff and U. Zwick. A 7/8-approximation algorithm for MAX 3SAT? In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 406–415. IEEE Computer Society, 1997.

I. Katriel, M. Sellmann, E. Upfal, and P. Van Hentenryck. Propagating Knapsack Constraints in Sublinear Time. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI)*, pages 231–236. AAAI Press, 2007.

E. de Klerk and H. van Maaren. On semidefinite programming relaxations of (2+p)-SAT. *Annals of Mathematics and Artificial Intelligence*, 37:285–305, 2003.

E. de Klerk and J.P. Warners. Semidefinite Programming Approaches for MAX-2-SAT and MAX-3-SAT: computational perspectives. In P.M. Pardalos, A. Migdalas, and R.E. Burkard, editors, *Combinatorial and Global Optimization*. World Scientific, 2002.

E. de Klerk, H. van Maaren, and J.P. Warners. Relaxations of the Satisfiability Problem Using Semidefinite Programming. *Journal of Automated Reasoning*, 24:37–65, 2000.

H.C. Lau. A New Approach for Weighted Constraint Satisfaction. *Constraints*, 7:151–165, 2002.

M. Laurent and F. Rendl. Semidefinite Programming and Integer Programming. In K. Aardal, G. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, Handbooks in Operations Research and Management Science. Elsevier, 2004. Also available as Technical Report PNA-R0210, CWI, Amsterdam.

M. Laurent, S. Poljak, and F. Rendl. Connections between semidefinite relaxations of the max-cut and stable set problems. *Mathematical Programming*, 77:225–246, 1997.

L. Leahu and C.P. Gomes. Quality of LP-based Approximations for Highly Combinatorial Problems. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3258 of *Lecture Notes in Computer Science*, pages 377–392. Springer, 2004.

L. Lovász. On the Shannon capacity of a graph. *IEEE Transactions on Information Theory*, 25:1–7, 1979.

F. Margot. Symmetry in Integer Linear Programming. In *50 Years of Integer Programming 1958–2008*, chapter 17. Springer, 2010.

M. Milano, editor. *Constraint and Integer Programming - Toward a Unified Methodology*, volume 27 of *Operations Research/Computer Science Interfaces*. Kluwer Academic Publishers, 2003.

M. Milano and W.J. van Hoeve. Reduced Cost-Based Ranking for Generating Promising Subproblems. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2470 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, 1988.

J.F. Puget. PECOS: a high level constraint programming language. In *Proceedings of the Singapore International Conference on Intelligent Systems (SPICIS)*, 1992.

P. Raghavendra. Optimal Algorithms and Inapproximability Results for Every CSP? In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*, pages 245–254. ACM, 2008.

P. Raghavendra and D. Steurer. Integrality Gaps for Strong SDP Relaxations of UNIQUE GAMES. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 575–585. IEEE Computer Society, 2009a.

P. Raghavendra and D. Steurer. How to Round Any CSP. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 586–594. IEEE Computer Society, 2009b.

P. Refalo. Linear Formulation of Constraint Programming Models and Hybrid Solvers. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 1894 of *Lecture Notes in Computer Science*, pages 369–383. Springer, 2000.

J.-C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI)*, volume 1, pages 362–367. AAAI Press, 1994.

J.-C. Régin. Cost-Based Arc Consistency for Global Cardinality Constraints. *Constraints*, 7: 387–405, 2002.

J.-C. Régin. Global Constraints and Filtering Algorithms. In M. Milano, editor, *Constraint and Integer Programming - Toward a Unified Methodology*, volume 27 of *Operations Research/Computer Science Interfaces*, chapter 4. Kluwer Academic Publishers, 2003.

J.-C. Régin. Modélisation et Contraintes Globales en Programmation par Contraintes. Habilitation thesis, University of Nice, 2004.

F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

A. Sadler and C. Gervet. Global Filtering for the Disjointness Constraint on Fixed Cardinality Sets. Technical Report TR-IC-PARC-04-02, IC-PARC, Imperial College, 2004.

M. Sellmann, T. Gellermann, and R. Wright. Cost-Based Filtering for Shorter Path Constraints. *Constraints*, 12(2):207–238, 2007.

P. van Beek. Backtracking search algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 4. Elsevier, 2006.

P. Van Hentenryck, L. Perron, and J.-F. Puget. Search and strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):285–320, 2000.

J. Warners. *Nonlinear Approaches to Satisfiability Problems*. PhD thesis, Technische Universiteit Eindhoven, 1999.

Z. Xing and W. Zhang. MaxSolver: An efficient exact algorithm for (weighted) maximum satisfiability. *Artificial Intelligence*, 164(1–2):47–80, 2005.

U. Zwick. Approximation Algorithms for Constraint Satisfaction Problems Involving at Most Three Variables per Constraint. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 201–210. ACM/SIAM, 1998.

Index