Exploiting Semidefinite Relaxations in Constraint Programming

W.J. van Hoeve

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

Constraint programming uses enumeration and search tree pruning to solve combinatorial optimization problems. In order to speed up this solution process, we investigate the use of semidefinite relaxations within constraint programming. In principle, we use the solution of a semidefinite relaxation to guide the traversal of the search tree, using a limited discrepancy search strategy. Furthermore, a semidefinite relaxation generally produces a tight bound for the solution value, which improves the pruning behaviour. Experimental results on stable set problem instances and maximum clique problem instances show that constraint programming can indeed greatly benefit from semidefinite relaxations.

Key words: constraint programming, semidefinite programming, search

1 Introduction

Constraint programming models for combinatorial optimization problems consist of variables on discrete domains, constraints binding those variables and an objective function to be optimized. In general, constraint programming solvers use domain value enumeration to solve combinatorial optimization problems. By propagation of the constraints (i.e. removal of inconsistent values), large parts of the resulting search tree may be pruned. Because combinatorial optimization problems are NP-hard in general, constraint propagation is essential to make constraint programming solvers practically applicable. Another essential part concerns the enumeration scheme, that defines and traverses a search tree. Variable and value ordering heuristics as well as tree traversal heuristics

Email address: W.J.van.Hoeve@cwi.nl (W.J. van Hoeve).
URL: http://homepages.cwi.nl/~wjvh/ (W.J. van Hoeve).

greatly influence the performance of the resulting constraint programming solver.

This work presents a method to exploit semidefinite relaxations in constraint programming. In particular, we use the solution of a semidefinite relaxation to define search tree ordering and traversal heuristics. Effectively, this means that our enumeration scheme starts at the suggestion made by the semidefinite relaxation, and gradually scans a wider area around this solution. Secondly, we use the solution value of the semidefinite relaxation as a bound for the objective function, which results in stronger pruning. By applying a semidefinite relaxation in this way, we hope to speed up the constraint programming solver significantly. These ideas were motivated by a previous work, in which a linear relaxation was proven helpful in constraint programming [21].

We implemented our method and provide experimental results on the stable set problem and the maximum clique problem, two classical combinatorial optimization problems. We compare our method with a standard constraint programming solver, and with specialized solvers for maximum clique problems. As computational results will show, our method obtains far better results than a standard constraint programming solver. However, on maximum clique problems, the specialized solvers appear to be much faster than our method.

The outline of the paper is as follows. The next section gives a motivation for the approach proposed in this work. Then, in Section 3 some preliminaries on constraint and semidefinite programming are given. A description of our solution framework is given in Section 4. In Section 5 we introduce the stable set problem and the maximum clique problem, integer optimization formulations and a semidefinite relaxation. Section 6 presents the computational results. Finally, we conclude in Section 7.

This paper is an extended and revised version of [17]. In the current version, a more general view on the proposed method is presented. Also, the subproblem generation framework has been replaced by limited discrepancy search on single values. Finally, more experimental results are presented, including instances of the DIMACS benchmark set for the maximum clique problem.

2 Motivation

NP-hard combinatorial optimization problems are often solved with the use of a polynomially solvable relaxation. In general, linear relaxations are chosen for this purpose. Also within constraint programming, linear relaxations are widely used, for instance to guide the search [1,9,21]. Let us first motivate why in this paper a semidefinite relaxation is used rather than a linear relaxation.

For some problems, for instance for the stable set problem, linear relaxations are not very tight and not informative. One way to overcome this problem is to identify and add inequalities that strengthen the relaxation. But it is time-consuming to identify such inequalities, and by enlarging the model the solution process may slow down.

On the other hand, several papers on approximation theory following [12] have shown the tightness of semidefinite relaxations. However, being tighter, semidefinite programs are more time-consuming to solve than linear programs in practice. Hence one has to trade strength for computation time. For some (large scale) applications, semidefinite relaxations are well suited to be used within a branch and bound framework (see for instance [19]). Moreover, our intention is not to solve a relaxation at every node of the search tree. Instead, we propose to solve only once a relaxation, before entering the search tree. Therefore, we are willing to make the trade-off in favour of the semidefinite relaxation.

Finally, to our knowledge the cross-fertilization of semidefinite programming and constraint programming has not yet been investigated. This paper, however, does aim at the cooperation of constraint programming and semidefinite programming.

3 Preliminaries

3.1 Constraint Programming

In this section we briefly introduce the basic concepts of constraint programming that are used in this paper. A thorough explanation of the principles of constraint programming can be found in [5].

A constraint programming model consists of a set of variables, corresponding variable domains, and a set of constraints C restricting those variables. In case of optimization problems, also an objective function is added (see Figure 1).

Basically, a constraint programming solver tries to find a solution of the model by enumerating all possible variable-value assignments such that the constraints are all satisfied. Because there are exponentially many possible assignments, constraint propagation is needed to prune large parts of the corresponding search tree. Constraint propagation tries to remove inconsistent values from variable domains before the variables are actually instantiated. Hence, one doesn't need to generate the whole search tree, but only a part of it, while still preserving a complete (or exact) solution scheme. The general

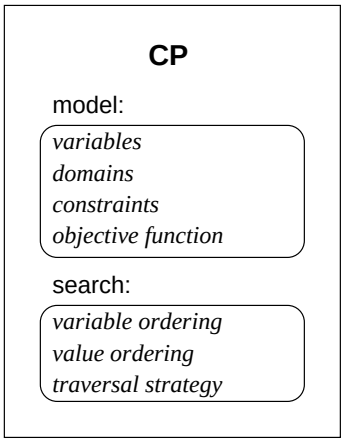

Fig. 1. Basic ingredients of constraint programming.

solution scheme is an iterative process in which branching decision are made, and the effects are propagated subsequently.

Variable and value ordering heuristics, that define the search tree, greatly influence the constraint propagation, and with that the performance of the solver. In general, constraint programming solvers use a lexicographic variable and value ordering, and depth-first search to traverse the tree. However, when good heuristics are available, they should be used.

When a perfect heuristic is followed, the first leaf of the search tree will be an optimal solution (although possibly unproven). Although perfect heuristics are often not available, some heuristics come pretty close. In such cases, one should try to deviate from the first heuristic solution as little as possible. This is done by traversing the search tree using a limited discrepancy search strategy (LDS) [15] instead of depth-first search.

LDS is organized in waves of increasing discrepancy. The first wave (discrepancy 0) exactly follows the heuristic. The next waves (discrepancy i, with $i > 0$), explore all the solutions that can be reached when i derivations from the heuristic are made. Typically, LDS is applied until a maximum discrepancy has been reached, say 3 or 4. Although being incomplete (or inexact), the resulting strategy often finds good solutions very fast, provided that the heuristic is informative. Of course LDS can also be applied until all possible discrepancies have been considered, resulting in a complete strategy.

3.2 Semidefinite Programming

In this section we introduce semidefinite programming as an extension of the more common linear programming. Both paradigms can be used to model polynomially solvable relaxations of NP-hard optimization problems. A large

number of references to papers concerning semidefinite programming are on the web pages of Helmberg [16] and Alizadeh [2]. A general introduction on semidefinite programming applied to combinatorial optimization is given by Goemans and Rendl [11].

In linear programming, combinatorial optimization problems are modeled in the following way:

$$\begin{aligned} &\max\ c^{\mathsf{T}}x \\ &\text{s.t. } a_j^{\mathsf{T}}x \le b_j \ (j = 1, \ldots, m) \\ &\qquad x \ge 0. \end{aligned} \tag{1}$$

Here $x \in \mathbb{R}^n$ is an n-dimensional vector of decision variables and $c \in \mathbb{R}^n$ a cost vector of dimension n. The m vectors $a_j \in \mathbb{R}^n$ $(j = 1, \ldots, m)$ and the m-dimensional vector $b \in \mathbb{R}^m$ define m linear constraints on x. In other words, this approach models problems using nonnegative vectors of variables.

Semidefinite programming makes use of positive semidefinite matrices of variables instead of nonnegative vectors. A matrix $X \in \mathbb{R}^{n \times n}$ is said to be positive semidefinite (denoted by $X \succeq 0$) when $y^{\mathsf{T}}Xy \ge 0$ for all vectors $y \in \mathbb{R}^n$. Semidefinite programs have the form

$$\begin{aligned} &\max\ \mathrm{tr}(CX) \\ &\text{s.t. } \mathrm{tr}(A_jX) \le b_j \ (j = 1, \ldots, m) \\ &\qquad X \succeq 0. \end{aligned} \tag{2}$$

Here $\mathrm{tr}(X)$ denotes the trace of X, which is the sum of its diagonal elements, i.e. $\mathrm{tr}(X) = \sum_{i=1}^{n} X_{ii}$. The cost matrix $C \in \mathbb{R}^{n \times n}$ and the constraint matrices $A_j \in \mathbb{R}^{n \times n}$ are supposed to be symmetric. The m reals b_j and the m matrices A_j define again m constraints.

We can view semidefinite programming as an extension of linear programming. Namely, when the matrices C and A_j $(j = 1, \ldots, m)$ are all supposed to be diagonal matrices [1], the resulting semidefinite program is equal to a linear program. In particular, then a semidefinite programming constraint $\mathrm{tr}(A_jX) \le b_j$ corresponds to the linear programming constraint $a_j^{\mathsf{T}}x \le b_j$, where a_j represents the diagonal of A_j.

Applied as a continuous relaxation (i.e. the integrality constraint on the vari-

[1] A diagonal matrix is a matrix with nonnegative values on its diagonal entries only.

ables is relaxed), semidefinite programming in general produces solutions that are much closer to the integral optimum than linear programming. Intuitively, this can be explained as follows. Demanding positive semidefiniteness of a matrix automatically implies nonnegativity of its diagonal. If this diagonal corresponds (as in the general case described above) to the nonnegative vector of the linear relaxation, the semidefinite relaxation is stronger than a linear relaxation. Unfortunately, it is not a trivial task to obtain a good (i.e. efficient) semidefinite program for a given problem.

Theoretically, semidefinite programs have been proved to be polynomially solvable using the so-called ellipsoid method (see for instance [13]). In practice, nowadays fast 'interior point' methods are being used for this purpose (see [3] for an overview).

4 Solution Framework

The skeleton of our solution framework is formed by the constraint programming enumeration scheme, or search tree, as explained in Section 3.1. Within this skeleton, we want to use the solution of a semidefinite relaxation to define the variable and value ordering heuristics. Hence, we assume a constraint programming model, consisting of variables, domains and constraints, together with an objective function. We also need to extract a semidefinite relaxation from our constraint programming model. As was indicated in Section 3.2, we need to transform the current model that uses a nonzero vector into a model that uses a positive semidefinite matrix to represent our variables. For a model with binary variables, one can always do this, as shown in [22]. Consequently, in the remainder of this paper, we restrict ourselves to constraint programming models using binary variables only.

The method presented in [22] is the following. Consider an array of binary variables $d \in \{0,1\}^n$. We can construct a $(n \times n)$ matrix X whose entries are defined by $X_{ij} = d_i d_j$. The integrality condition on d is equivalent to the condition $d_i^2 = d_i$ (for all i). This condition can be obtained by adding an extra row and column (indexed 0) to X, and linking them to the diagonal, i.e. $X_{ii} = d_i$ (for all i). Namely, then $X_{ii} = d_i d_i = d_i^2 = d_i$. Finally, one may relax the integrality condition by requiring

$$\begin{bmatrix} 1 & d^{\mathsf{T}} \\ d & X \end{bmatrix} \succeq 0, \text{ where } d_i = X_{ii}, \tag{3}$$

where the 1 in the leftmost corner is needed to obtain positive semidefiniteness. Note that we replace the integrality condition on d by requiring positive

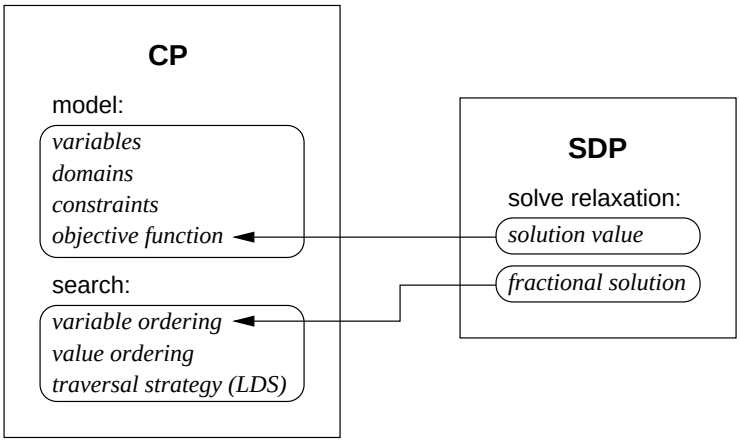

Fig. 2. Communication between constraint programming (CP) and semidefinite programming (SDP).

semidefiniteness on the above matrix. The resulting positive semidefinite matrix 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 original binary variables. Consequently, constraints from the constraint programming model may be added to the semidefinite relaxation, possibly after linearizing them, in accordance to program (2).

Now that the connection between the two models has been established, we show how to use the solution to the semidefinite relaxation, as depicted in Figure 2. In general, the solution to the semidefinite relaxation yields fractional values between 0 and 1 for the diagonal variables X_{ii}. Our variable and value ordering heuristics for the constraint programming variables are simply based upon these values. In our implementations (stable set problem and maximum clique problem), we select the first uninstantiated variable closest to 1. Our value ordering heuristic is to select value 1 before value 0, if possible.

We expect the semidefinite relaxation to provide promising values. Therefore the resulting search tree will be traversed using limited discrepancy search, defined in Section 3.1. The first heuristic solution (of discrepancy 0) to be followed is obtained by a randomized rounding procedure. Using the nondecreasing variable ordering, we accept a variable to be 1 with a probability equal to its fractional value. This procedure is repeated n times (where n is the number of variables), and the best solution is the heuristic solution to be followed.

A last remark concerns the solution value of the semidefinite relaxation, which is used as a bound on the objective function in the constraint programming model. This (often tight) bound on the objective function leads to more propagation and a smaller search space.

5 The Stable Set Problem and Maximum Clique Problem

This section describes the stable set problem and the maximum clique problem (see [25,6] for a survey), on which we have tested our algorithm. First we give their definitions, and the equivalence of the two problems. Then we will focus on the stable set problem, and formulate it as an integer optimization problem. From this, a semidefinite relaxation is inferred.

5.1 Definitions

Consider an undirected weighted graph $G = (V, E)$, where $V = \{1, \ldots, n\}$ is the set of vertices and E a subset of edges $\{(i, j)|i, j \in V, i \neq j\}$ of G, with $|E| = m$. To each vertex $i \in V$ a weight $w_i \in \mathbb{R}$ is assigned (without loss of generality, we can assume all weights to be nonnegative in this case).

A *stable set* is a set $S \subseteq V$ such that no two vertices in S are joined by an edge in E. The *stable set problem* is the problem of finding a stable set of maximum total weight in G. This value is called the *stable set number* of G and is denoted by $\alpha(G)$ [2]. In the unweighted case (when all weights are equal to 1), this problem amounts to the maximum cardinality stable set problem, which has been shown to be already NP-hard [24].

A *clique* is a set $C \subseteq V$ such that every two vertices in C are joined by an edge in E. The *maximum clique problem* is the problem of finding a clique of maximum total weight in G. This value is called the *clique number* of G and is denoted by $\omega(G)$ [3].

The complement graph of G is $\overline{G} = (V, \overline{E})$, with the same set of vertices $V = \{1, \ldots, n\}$, but with edge set $\overline{E} = \{(i, j)|i, j \in V, (i, j) \notin E, i \neq j\}$. It is well known that $\alpha(G) = \omega(\overline{G})$. Hence, a maximum clique problem can be translated into a stable set problem on the complement graph. We will do exactly this in our implementation, and focus on the stable set problem, for which good semidefinite relaxations exist.

[2] In the literature $\alpha(G)$ usually denotes the unweighted stable set number. The weighted stable set number is then denoted as $\alpha_w(G)$. In this work, it is not necessary to make this distinction.

[3] $\omega_w(G)$ is defined similar to $\alpha_w(G)$ and also not distinguished in this paper.

5.2 Integer Optimization Formulation

Let us first consider an integer linear programming formulation for the stable set problem. We introduce binary variables to indicate whether or not a vertex belongs to the stable set S. So, for n vertices, we have n integer variables x_i indexed by $i \in V$, with initial domains $\{0, 1\}$. In this way, $x_i = 1$ if vertex i is in S, and $x_i = 0$ otherwise. We can now state the objective function, being the sum of the weights of vertices that are in S, as $\sum_{i=1}^{n} w_i x_i$. Finally, we define the constraints that restrict two adjacent vertices to be both inside S as $x_i + x_j \leq 1$, for all edges $(i, j) \in E$. Hence the integer linear programming model becomes:

$$\begin{aligned} \alpha(G) = \max \quad & \textstyle\sum_{i=1}^{n} w_i x_i \\ \text{s.t. } & x_i + x_j \leq 1 \ \forall (i,j) \in E \\ & x_i \in \{0,1\} \ \forall i \in V. \end{aligned} \tag{4}$$

Another way of describing the same solution set is presented by the following integer quadratic program

$$\begin{aligned} \alpha(G) = \max & \textstyle\sum_{i=1}^{n} w_i x_i \\ \text{s.t. } & x_i x_j = 0 \ \forall (i,j) \in E \\ & x_i^2 = x_i \ \forall i \in V. \end{aligned} \tag{5}$$

Note that here the constraint $x_i \in \{0, 1\}$ is replaced by $x_i^2 = x_i$, as was also done in Section 4. This quadratic formulation will be used below to infer a semidefinite relaxation of the stable set problem.

In fact, both model (4) and model (5) can be used as a constraint programming model. We have chosen the first model, since the quadratic constraints take more time to propagate than the linear constraints, while having the same pruning power.

5.3 Semidefinite Programming Relaxation

The integer quadratic program (5) gives rise to a semidefinite relaxation introduced by Lovász [20] (see Grötschel et al. [13] for a comprehensive treatment). The value of the objective function of this relaxation has been named the *theta number* of a graph G, indicated by $\vartheta(G)$. For its derivation into a form similar

to program (2), we will follow the same idea as in Section 4 for the general case.

Let us start again from model (5). We can construct a matrix $X \in \mathbb{R}^{n\times n}$ by defining $X_{ij} = x_i x_j$. Let us also construct a $n \times n$ cost matrix W with $W_{ii} = w_i$ for $i \in V$ and $W_{ij} = 0$ for all $i \neq j$. Since $X_{ii} = x_i^2 = x_i$, the objective function becomes $\mathrm{tr}(WX)$. The edge constraints are easily transformed as $x_i x_j = 0 \Leftrightarrow X_{ij} = 0$. The integrality condition will be transformed as in Section 4. Namely, extend X with another row and column (both indexed by 0) that contain vector x, and define the $(n+1) \times (n+1)$ matrix Y as

$$Y = \begin{pmatrix} 1 & x^{\mathsf{T}} \\ x & X \end{pmatrix}$$

where $X_{ii} = x_i$. Finally, we will relax the integrality constraint on x by demanding Y to be a positive semidefinite matrix.

In order to maintain equal dimension to Y, a row and a column (both indexed by 0) should be added to W, all entries of which containing value 0. Denote the resulting matrix by $\tilde{W}$. The theta number of a graph G can now be described as

$$\begin{aligned} \vartheta(G) = \max \quad & \mathrm{tr}(\tilde{\mathrm{W}}\mathrm{Y}) \\ \text{s.t. } & Y_{ii} = \tfrac{1}{2}Y_{i0} + \tfrac{1}{2}Y_{0i} \;\; \forall i \in V \\ & Y_{ij} = 0 \qquad \forall (i,j) \in E \\ & Y \succeq 0. \end{aligned} \tag{6}$$

By construction, the diagonal value Y_{ii} serves as an indication for the value of variable x_i ($i \in V$) in a maximum stable set. In particular, this program is a relaxation for the stable set problem, i.e. $\vartheta(G) \geq \alpha(G)$. Note that program (6) can easily be rewritten into the general form of program (2). Namely, $Y_{ii} = \frac{1}{2}Y_{i0} + \frac{1}{2}Y_{0i}$ is equal to $\mathrm{tr}(AY)$ where the $(n+1) \times (n+1)$ matrix A consists of all zeroes, except for $A_{ii} = 1$, $A_{i0} = -\frac{1}{2}$ and $A_{0i} = -\frac{1}{2}$, which makes the corresponding b entry equal to 0. Similarly for the edge constraints.

The theta number also arises from other formulations, different from the above, see [13]. In our implementation we have used the formulation that has been shown to be computationally most efficient among those alternatives [14]. Let us introduce that particular formulation (called ϑ_3 in [13]). Again, let $x \in \{0,1\}^n$ be a vector of binary variables representing a stable set. Define the $n \times n$ matrix $X = \xi\xi^{\mathsf{T}}$ where $\xi_i = \frac{\sqrt{w_i}}{\sqrt{\sum_{j=1}^{n} w_j x_j}} x_i$. 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

$$\begin{aligned} \vartheta(G) = \max \quad & \operatorname{tr}(UX) \\ \text{s.t. } & \operatorname{tr}(X) = 1 \\ & X_{ij} = 0 \ \forall (i,j) \in E \\ & X \succeq 0 \end{aligned} \tag{7}$$

gives exactly the theta number of G. When (7) is solved to optimality, the scaled diagonal element $\vartheta(G)X_{ii}$ (a fractional value between 0 and 1) serves as an indication for the value of x_i ($i \in V$) in a maximum stable set (see for instance [14]). Again, it is not difficult to rewrite program (7) into the general form of program (2).

Program (7) uses matrices of dimension n and $m + 1$ constraints, while program (6) uses matrices of dimension $n + 1$ and $m + n$ constraints. This gives an indication why program (7) is computationally more efficient.

6 Computational Results

All our experiments are performed on a Sun Enterprise 450 (4 X UltraSPARC-II 400MHz) with maximum 2048 Mb memory size, on which our algorithms only use one processor of 400MHz at a time. As constraint programming solver we use the ILOG Solver library, version 5.1 [18]. As semidefinite programming solver, we use CSDP version 4.1 [7], with the optimized ATLAS 3.4.1 [28] and LAPACK 3.0 [4] libraries for matrix computations. The reason for our choices is that both solvers are among the fastest in their field, and because ILOG Solver is written in C++, and CSDP is written in C, they can be hooked together relatively easy.

As constraint programming model we have used model (4), as was argued in Section 5.2. We distinguish two algorithms to perform our experiments. The first algorithm is a sole constraint programming solver, which uses a standard labeling strategy. This means we use a lexicographic variable ordering, and we select domain value 1 before value 0. The resulting search tree is traversed using a depth-first search strategy. After each branching decision, its effect is directly propagated through the constraints.

The second algorithm is the one proposed in Section 4. It first solves the semidefinite program (7), and then calls the constraint programming solver. In

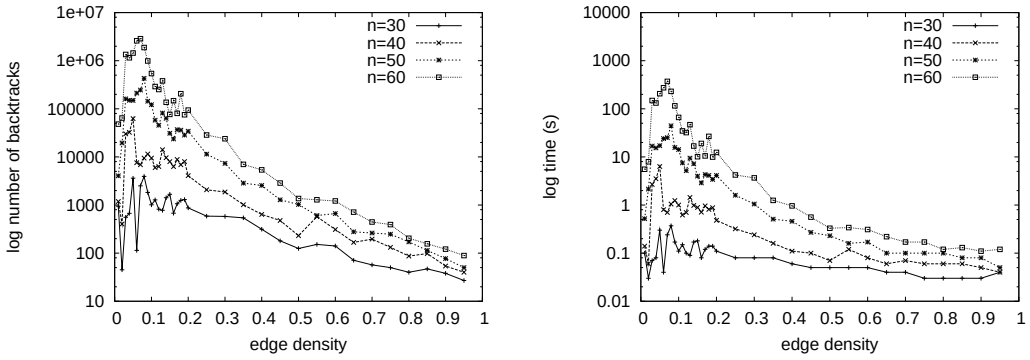

Fig. 3. Performance of the constraint programming solver on random instances with n vertices.

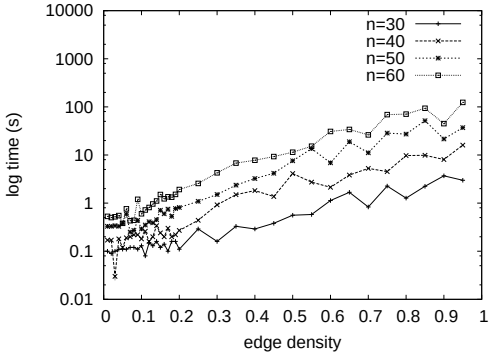

Fig. 4. Performance of the semidefinite programming solver on random instances with n vertices.

this case, we use a variable ordering defined by the solution of the semidefinite relaxation. The resulting search tree is traversed using a limited discrepancy search strategy.

6.1 Characterization of Problem Instances

We will first identify general characteristics of the constraint programming solver and semidefinite programming solver to obtain a maximum stable set on random graphs. It appears that both solvers are highly dependent on the edge density of the graph, i.e. $\frac{m}{\frac{1}{2}(n^2-n)}$ for a graph with n vertices and m edges. We therefore generated random graphs on 30, 40, 50 and 60 vertices, with density ranging from 0.01 up to 0.95. Our aim is to identify the hardness of the instances for both solvers, parametrized by the density. Based upon this information, we can make decisions on what kind of problems our algorithm is suitable for.

We have plotted the performance of both solvers in Figure 3 and Figure 4. For the constraint programming solver, we depict both the number of backtracks

and the time needed to prove optimality. For the semidefinite programming solver we only plotted the time needed to solve the relaxation. Namely, this solver does not use a tree search, but a so-called primal-dual interior point algorithm. Note that we use a log-scale for time and number of backtracks in these pictures.

From these figures, we can conclude that the constraint programming solver has the most difficulties with instances up to density around 0.2. Here we see the effect of constraint propagation. As the number of constraints increases, the search tree can be heavily pruned. On the other hand, our semidefinite relaxation suffers from every edge that is added. As the density increases, the semidefinite program increases accordingly, as well as its computation time. Fortunately, for the instances up to 0.2, the computation time for the semidefinite relaxation is very small. Consequently, our algorithm is expected to behave best for graphs that have edge density up to around 0.2. For graphs with a higher density, the constraint programming solver is expected to use less time than the semidefinite programming solver, which makes the application of our method unnecessary.

6.2 Random Weighted and Unweighted Graphs

Our first experiments are performed on random weighted and unweighted graphs. We generated graphs with 50, 75, 100, 125 and 150 vertices and edge density from 0.05, 010 and 0.15, corresponding to the interesting problem area. The results are presented in Table 1. Unweighted graphs on n vertices and edge density r are named 'gndr'. Weighted graphs are similarly named 'wg$n$dr'.

The first five columns of the table are dedicated to the instance, reporting its name, the number of vertices n and edges m, the edge density and the (best known) value of a stable set, α. The next three columns (CP) present the performance of the constraint programming solver, reporting the best found estimate of α, the total time and the total number of backtracks needed to prove optimality. The last five columns (SDP+CP) present the performance of our method, where also a column has been added for the discrepancy of the best found value (best discr), and a column for the time needed by the semidefinite programming solver (sdp time).

Table 1 shows that our approach always finds a better (or equally good) estimate for α than the standard constraint programming approach. This becomes more obvious for larger n. However, there are two (out of 30) instances in which our method needs substantially more time to achieve this result (g75d015 and wg75d010). A final observation concerns the discrepancy of the best found so-

Table 1
Computational results on random graphs, with n vertices and m edges. All times are in seconds. The time limit is set to 1000 seconds.

instance				**CP**				**SDP + CP**				
name	n	m	edge density	α	α	total time	back-tracks	α	best discr	sdp time	total time	back-tracks
g50d005	50	70	0.06	27	27	5.51	50567	27	0	0.26	0.27	0
g50d010	50	114	0.09	22	22	28.54	256932	22	0	0.35	0.36	0
g50d015	50	190	0.16	17	17	5.83	48969	17	0	0.49	0.49	0
g75d005	75	138	0.05	36	≥ 35	limit		36	0	0.72	0.73	0
g75d010	75	282	0.10	≥ 25	≥ 25	limit		≥ 25	5	1.4	limit	
g75d015	75	426	0.15	21	21	170.56	1209019	21	0	2.81	664.92	1641692
g100d005	100	254	0.05	43	≥ 40	limit		43	0	2.07	2.1	0
g100d010	100	508	0.10	≥ 31	≥ 30	limit		≥ 31	0	4.94	limit	
g100d015	100	736	0.15	≥ 24	≥ 24	limit		≥ 24	4	9.81	limit	
g125d005	125	393	0.05	≥ 49	≥ 44	limit		≥ 49	1	4.92	limit	
g125d010	125	791	0.10	≥ 33	≥ 30	limit		≥ 33	6	12.58	limit	
g125d015	125	1160	0.15	≥ 27	≥ 24	limit		≥ 27	1	29.29	limit	
g150d005	150	545	0.05	≥ 52	≥ 44	limit		≥ 52	3	10.09	limit	
g150d010	150	1111	0.10	≥ 38	≥ 32	limit		≥ 38	4	27.48	limit	
g150d015	150	1566	0.14	≥ 29	≥ 26	limit		≥ 29	8	57.86	limit	
wg50d005	50	70	0.06	740	740	4.41	30528	740	0	0.29	0.3	0
wg50d010	50	126	0.10	636	636	3.12	19608	636	0	0.41	0.41	0
wg50d015	50	171	0.14	568	568	4.09	25533	568	0	0.59	4.93	13042
wg75d005	75	128	0.05	1761	1761	744.29	4036453	1761	0	1.05	1.07	0
wg75d010	75	284	0.10	1198	1198	325.92	1764478	1198	13	1.9	924.2	1974913
wg75d015	75	409	0.15	972	972	40.31	208146	972	0	3.62	51.08	87490
wg100d005	100	233	0.05	2302	≥ 2176	limit		2302	0	2.59	2.62	0
wg100d010	100	488	0.10	≥ 1778	≥ 1778	limit		≥ 1778	2	6.4	limit	
wg100d015	100	750	0.15	≥ 1412	≥ 1412	limit		≥ 1412	2	15.21	limit	
wg125d005	125	372	0.05	≥ 3779	≥ 3390	limit		≥ 3779	3	5.39	limit	
wg125d010	125	767	0.10	≥ 2796	≥ 2175	limit		≥ 2796	0	18.5	limit	
wg125d015	125	1144	0.15	≥ 1991	≥ 1899	limit		≥ 1991	4	38.24	limit	
wg150d005	150	588	0.05	≥ 4381	≥ 3759	limit		≥ 4381	3	13.57	limit	
wg150d010	150	1167	0.10	≥ 3265	≥ 2533	limit		≥ 3265	9	40.68	limit	
wg150d015	150	1630	0.15	≥ 2828	≥ 2518	limit		≥ 2828	11	82.34	limit	

lutions. Our method appears to find those (often optimal) solutions at rather low discrepancies.

6.3 Graphs Arising from Coding Theory

The next experiments are performed on structured (unweighted) graphs arising from coding theory, obtained from [27]. We have used those instances that were solvable in reasonable time by the semidefinite programming solver (here reasonable means within 1000 seconds). For these instances, the value of α happened to be known already.

The results are reported in Table 2, which follows the same format as Table 1. It shows the same behaviour as the results on random graphs. Namely, our method always finds better solutions than the standard constraint programming solver, in less time or within the time limit. This is not surprising,

Table 2
Computational results on graphs arising from coding theory, with n vertices and m edges. All times are in seconds. The time limit is set to 1000 seconds.

instance					**CP**			**SDP + CP**				
name	n	m	edge density	α	α	total time	back-tracks	α	best discr	sdp time	total time	back-tracks
1dc.64	64	543	0.27	10	10	11.44	79519	10	0	5.08	5.09	0
1dc.128	128	1471	0.18	16	$\geq$ 16	limit		16	0	49.95	49.98	0
1dc.256	256	3839	0.12	30	$\geq$ 26	limit		30	0	882.21	882.33	0
1et.64	64	264	0.13	18	18	273.06	2312832	18	0	1.07	1.08	0
1et.128	128	672	0.08	28	$\geq$ 28	limit		$\geq$ 28	0	11.22	limit	
1et.256	256	1664	0.05	50	$\geq$ 46	limit		$\geq$ 50	0	107.58	limit	
1tc.64	64	192	0.10	20	$\geq$ 20	limit		20	0	0.78	0.79	0
1tc.128	128	512	0.06	38	$\geq$ 37	limit		38	0	8.14	8.18	0
1tc.256	256	1312	0.04	63	$\geq$ 58	limit		$\geq$ 63	4	72.75	limit	
1tc.512	512	3264	0.02	110	$\geq$ 100	limit		$\geq$ 110	2	719.56	limit	
1zc.128	128	2240	0.28	18	$\geq$ 18	limit		$\geq$ 18	4	129.86	limit	

because the edge density of these instances are exactly in the region in which our method is supposed to behave best (with the exception of 1dc.64 and 1zc.128), as analyzed in Section 6.1. Again, our method finds the best solutions at a low discrepancy. Note that the instance 1et.64 shows the strength of the semidefinite relaxation with respect to standard constraint programming. The difference in computation time to prove optimality is huge.

6.4 Graphs from the DIMACS Benchmarks Set

Our final experiments are performed on a subset of the DIMACS benchmark set for the maximum clique problem [8]. Although our method is not intended to be competitive with the best heuristics and exact methods for maximum clique problems, it is still interesting to see its performance on this standard benchmark set. As pointed out in Section 5, we have transformed these maximum clique problems to stable set problems on the complement graph.

The results are reported in Table 3, which again follows the same format as Table 1. The choice for this particular subset of instances is made by the solvability of an instance by a semidefinite programming solver in reasonable time (again, reasonable means 1000 seconds). For all instances with edge density smaller than 0.24, our method outperforms the standard constraint programming approach. For higher densities however, the opposite holds. This is exactly what could be expected from the analysis of Section 6.1. A special treatment has been given to instance MANN_a45. We stopped the semidefinite programming solver at the time limit of 1000 seconds, and used its intermediate feasible solution as if it were the optimal fractional solution. We then proceeded our algorithm for a couple of seconds more, to search for a solution up to discrepancy 1.

Table 3
Computational results on graphs from the DIMACS benchmark set for maximum clique problems, with n vertices and m edges. All times are in seconds. The time limit is set to 1000 seconds.

instance					**CP**			**SDP + CP**				
name	n	m	edge density	α	α	total time	back-tracks	α	best discr	sdp time	total time	back-tracks
hamming6-2	64	192	0.095	32	32	20.22	140172	32	0	0.68	0.69	0
hamming6-4	64	1312	0.651	4	4	0.28	804	4	0	27.29	28.10	706
hamming8-2	256	1024	0.031	128	≥ 128	limit		128	0	45.16	45.55	0
johnson8-2-4	28	168	0.444	4	4	0.05	255	4	0	0.35	0.35	0
johnson8-4-4	70	560	0.232	14	14	15.05	100156	14	0	4.82	4.83	0
johnson16-2-4	120	1680	0.235	8	≥ 8	limit		8	0	43.29	43.32	0
MANN_a9	45	72	0.072	16	16	162.81	1738506	16	1	0.17	82.46	411104
MANN_a27	378	702	0.010	126	≥ 103	limit		≥ 125	3	70.29	limit	
MANN_a45	1035	1980	0.004	345	≥ 156	limit		≥ 338	1	1047.06	limit	
san200_0.9_1	200	1990	0.100	70	≥ 45	limit		70	0	170.01	170.19	0
san200_0.9_2	200	1990	0.100	60	≥ 36	limit		60	0	169.35	169.51	0
san200_0.9_3	200	1990	0.100	44	≥ 26	limit		44	0	157.90	157.99	0
sanr200_0.9	200	2037	0.102	42	≥ 34	limit		≥ 41	4	131.57	limit	

Table 4
A comparison of different methods on graphs from the DIMACS benchmark set for maximum clique problems, with n vertices and m edges. All times are in seconds.

instance			**Östergård**		**Régin**			**CP + SDP**		
name	edge density	α	α	total time	α	total time	back-tracks	α	total time	back-tracks
hamming6-2	0.095	32	32	0.01	32	0.00	17	32	0.69	0
hamming6-4	0.651	4	4	0.01	4	0.00	42	4	28.10	706
hamming8-2	0.031	128	128	0.04	128	0.00	65	128	45.55	0
johnson8-2-4	0.444	4	16	0.01	4	0.00	14	4	0.35	0
johnson8-4-4	0.232	14	14	0.01	14	0.00	140	14	4.83	0
johnson16-2-4	0.235	8	8	0.27	8	11.40	250505	8	43.32	0
MANN_a9	0.073	16	16	0.01	16	0.00	50	16	82.46	411104
MANN_a27	0.010	126		> 10000	126	55.44	1258768	≥ 125	> 1000	
MANN_a45	0.004	345		> 10000	≥ 345	> 43200		≥ 338	> 1000	
san200_0.9_1	0.100	70	70	0.27	70	3.12	1040	70	170.19	0
san200_0.9_2	0.100	60	60	4.28	60	7.86	6638	60	169.51	0
san200_0.9_3	0.100	44		> 10000	44	548.10	758545	44	157.99	0
sanr200_0.9	0.102	42		> 10000	42	450.24	541496	≥ 41	> 1000	

In Table 4 we compare our method with two methods that are specialized for maximum clique problems. The first method was presented by Östergård [23], and follows a branch-and-bound approach. The second method is a constraint programming approach, using a special constraint for the maximum clique problem. This idea was introduced by Fahle [10] and extended and improved by Régin [26]. Since all methods are performed on different machines, we need to identify a time ratio between them. A machine comparison from SPEC [4] shows that our times are comparable with the times of Östergård. We have multiplied the times of Régin with 3, following the time comparison made in [26]. In

[4] `http://www.spec.org/`

general, our method is outperformed by the other two methods, although there is one instance on which our method performs best (san200_0.9_3).

7 Conclusion

We have presented a method to use semidefinite relaxations within constraint programming. The fractional solution values of the relaxation serve as an indication for the corresponding constraint programming variables. Moreover, the solution value of the relaxation is used as a bound for the corresponding constraint programming cost function.

We have implemented our method to find the maximum stable set in a graph. Experiments are performed on random weighted and unweighted graphs, structured graphs from coding theory, and on a subset of the DIMACS benchmarks set for maximum clique problems. Computational results show that constraint programming can greatly benefit from semidefinite programming. Indeed, the solution to the semidefinite relaxations turn out to be very informative. Compared to a standard constraint programming approach, our method obtains far better results. Specialized algorithms for the maximum clique problem however, still outperform our method, except for one problem instance.

Acknowledgements

Many thanks to Michela Milano, Monique Laurent and Sebastian Brand for fruitful discussions and helpful comments while writing (earlier drafts of) this paper.

References

[1] F. Ajili and H. El Sakkout. LP probing for piecewise linear optimization in scheduling. In *Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'01)*, pages 189–203, 2001.

[2] F. Alizadeh. The Semidefinite Programming Page. `http://new-rutcor.rutgers.edu/~alizadeh/sdp.html`.

[3] F. Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal on Optimization*, 5(1):13–51, 1995.

[4] E. Anderson, Z. Bai, C. Bischof, L.S. Blackford, J. Demmel, J.J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, third edition, 1999. `http://www.netlib.org/lapack/`.

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

[6] I.M. Bomze, M. Budinich, P.M. Pardalos, and M. Pelillo. The Maximum Clique Problem. In D.-Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 4. Kluwer, 1999.

[7] B. Borchers. A C Library for Semidefinite Programming. *Optimization Methods and Software*, 11(1):613–623, 1999. `http://www.nmt.edu/~borchers/csdp.html`.

[8] DIMACS. DIMACS maximum clique benchmark, 1993. `ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique`.

[9] H. El Sakkout and M. Wallace. Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling. *Constraints*, 5(4):359–388, 2000.

[10] T. Fahle. Simple and Fast: Improving a Branch-And-Bound Algorithm for Maximum Clique. In *10th Annual European Symposium on Algorithms (ESA 2002)*, volume 2461 of *LNCS*, pages 485–498. Springer Verlag, 2002.

[11] M. Goemans and F. Rendl. Combinatorial Optimization. In H. Wolkowicz, R. Saigal, and L. Vandenberghe, editors, *Handbook of Semidefinite Programming*, pages 343–360. Kluwer, 2000.

[12] 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.

[13] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. John Wiley & Sons, 1988.

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

[15] W. D. Harvey and M. L. Ginsberg. Limited Discrepancy Search. In C. S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1*, pages 607–615, 1995.

[16] C. Helmberg. Semidefinite Programming website. `http://www-user.tu-chemnitz.de/~helmberg/semidef.html`.

[17] W.J. van Hoeve. A Hybrid Constraint Programming and Semidefinite Programming Approach for the Stable Set Problem. In F. Rossi, editor, *Ninth International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833 of *LNCS*, pages –. Springer Verlag, 2003.

[18] ILOG. ILOG Solver 5.1, Reference Manual, 2001.

[19] S. E. Karisch, F. Rendl, and J. Clausen. Solving graph bisection problems with semidefinite programming. *INFORMS Journal on Computing*, 12(3):177–191, 2000.

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

[21] M. Milano and W.J. van Hoeve. Reduced cost-based ranking for generating promising subproblems. In P. van Hentenryck, editor, *Eighth International Conference on the Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *LNCS*, pages 1–16. Springer Verlag, 2002.

[22] 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.

[23] P.R.J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120:197–207, 2002.

[24] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[25] P.M. Pardalos and J. Xue. The Maximum Clique Problem. *SIAM Journal of Global Optimization*, 4:301–328, 1994.

[26] J.-C. Régin. Solving the Maximum Clique Problem with Constraint Programming. In *Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'03)*, pages 166–179, 2003.

[27] N.J.A. Sloane. Challenge Problems: Independent Sets in Graphs. `http://www.research.att.com/~njas/doc/graphs.html`.

[28] R.C. Whaley, A. Petitet, and J.J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. `http://math-atlas.sourceforge.net/`.