Pruning Moves

Matteo Fischetti[(*)] and Domenico Salvagnin[(◦)]

[(*)] *DEI, University of Padova, Italy*
[(◦)] *DMPA, University of Padova, Italy*
e-mail: matteo.fischetti@unipd.it, salvagni@math.unipd.it

May 4, 2008

Abstract

The concept of dominance among nodes of a branch-decision tree, although known since a long time, is not exploited by general-purpose Mixed-Integer Linear Programming (MILP) enumeration codes, due to the intrinsic difficulties arising when using the classical dominance definition in a completely general context.

The starting point of our work was the general-purpose dominance procedure proposed in the 80's by Fischetti and Toth, where the dominance test at a given branching node consists in the (possibly heuristic) solution of a restricted MILP only involving the fixed variables. Both theoretical and practical issues concerning this procedure are analyzed, and important improvements are proposed. In particular, we use the dominance test not only to fathom the current branching node, but also to derive variable configurations called "nogoods" (a concept borrowed from Constraint Programming) and, more generally, "improving moves" (a basic concept in the theory of test sets). These latter configurations, that we rename "pruning moves" so as to stress their use in a node fathoming context, are stored into an internal pool and used during the enumeration to fathom large sets of dominated solutions in a computationally effective way.

Computational results on a test-bed of MILP instances whose structure is amenable to dominance are reported, showing that the proposed method can lead to a considerable speedup when embedded in a commercial MILP solver.

Keywords: mixed-integer programming, constraint programming, dominance procedure, nogoods, test sets, knapsack problem, network design.

1 Introduction

The Mixed-Integer Linear Programming (MILP, for short) paradigm is among the most powerful and most used methods for modeling and solving both real-life and theoretical combinatorial optimization problems; see, e.g., [7, 18, 22]. Almost fifty years of active research in the field have produced huge improvements in the solving capability of the MILP codes, as reported e.g. in [2].

A concept playing a potentially relevant role in trying to keep the size of the enumeration tree as small as possible is that of *dominance*. Following e.g. Papadimitriou and Steiglitz [18], a branching node α is said to be dominated by another node β if every feasible solution in the subtree of α corresponds to a solution in the subtree of β having a better cost (tie breaks being handled by a suitable lexicographic rule). This concept seems to have been studied first by Kohler and Steiglitz [14], and has been developed in the subsequent years, most notably by Ibaraki [11]. However, although known since a long time, dominance criteria are not exploited in general-purpose MILP codes, due to number of important limitations of the classical definition. In fact, as stated, the dominance relationship is too vague and of difficult application in a generic MILP context—in principle, every node not leading to an optimal solution could be declared as being dominated.

In this paper we build on the general-purpose dominance procedure proposed in the late 80's by Fischetti and Toth [6], that overcomes some of the drawbacks of the classical dominance definition. The approach has its roots in the concept of dominance among "states" in dynamic programming, and works as follows. Given the current node α of the search tree, let J^α be the set of variables fixed to some value. We construct an auxiliary problem XP^α that looks for a new partial assignment involving the variables in J^α and such that (i) the objective function value is not worse than the one associated with the original assignment, and (ii) every completion of the old partial assignment is also a valid completion of the new one. If such a new partial assignment is found (and a certain tie-break rule is satisfied), one is allowed to fathom node α.

As far as we know, no attempt to actually use the above dominance procedure within a general-purpose MILP scheme is reported in the literature. This is due to the fact that the approach, as described, tends to be excessively time consuming—the reduction in the number of nodes does not compensate for the large overhead introduced. In the attempt of finding a viable way to implement the original scheme, we found that the concept of *nogood*, borrowed from Constraint Programming (CP), can play a crucial role for the practical effectiveness of the overall dominance procedure. Roughly speaking, a *nogood* is a partial assignment of the variables such that every completion is either infeasible (for constraint satisfaction problems) or nonoptimal (for constraint optimization problems). Though widely used in the CP context, the concept of nogood is seldom used in Mathematical Programming. One of the the first uses of nogoods in ILP was to solve verification problems (Hooker and Yan [9]) and fixed-charge network flow problems (Hooker et al. [13]). Further applications can be found in Codato and Fischetti [3], where nogoods are used to generate cuts for a MILP problem.

In the context of dominance, a nogood configuration is available, as a byproduct, whenever the auxiliary problem is solved successfully. More

generally, we show how the auxiliary problem can be used to derive "moves" that capture in much more effective way the presence of general integer (as opposed to binary) variables.

Improving moves are the basic elements of *test sets*. For an integer programming problem, a test set is defined as a set T of vectors such that every feasible solution x to the integer program is nonoptimal if and only if there exists an element $t \in T$ (the "improving move") such that $x + t$ is a feasible solution with strictly better objective function value; see, e.g., [8, 21, 23]. Within the classical test set environment, improving moves are meant to be used within a primal heuristic scheme to convert any feasible solution to an optimal one by a sequence of moves maintaining solution feasibility and improving in a strictly monotone way the objective value. Computing and using test sets for NP-hard problems is however by far too expensive in practice. In our approach, instead, improving moves are exploited heuristically within a node fathoming procedure—hence the name "pruning moves". More specifically, we generate pruning moves on the fly, when solving the dominance auxiliary problem XP^{α}, and store them into a *move pool* that is checked in a rather inexpensive way at each node of the branching tree. Computational results show that this approach can be very successful for problems whose structure is amenable to dominance.

The paper is organized as follows. In Section 2 we describe the Fischetti-Toth technique in more details, and address some issues related to the tie-break rule to be used in order to get a mathematically correct overall method. In Section 3 we introduce the concepts of nogood and pruning move, and show how to derive them in an effective way. In Section 4 we deal with some extensions of the basic scheme, whereas important implementation issues are addressed in Section 5. Computational results obtained on hard knapsack and network design instances are given in Section 6. We show that, for those problems, the proposed method can lead to a significant speedup when embedded within a general-purpose MILP solver (ILOG Cplex 11). Some conclusions are finally drawn in Section 7.

2 The Fischetti-Toth dominance procedure

In the standard Branch-and-Bound (B&B) or Branch-and-Cut (B&C) framework, a node is fathomed in two situations:

1. the LP relaxation of the node is infeasible; or
2. the optimal value of LP relaxation is not better than the value of the incumbent optimal solution.

There is however a third way of fathoming a node, by using the concept of *dominance*. According to [18], a dominance relation is defined as follows: if we can show that a descendant of a node β is at least as good as the best

descendant of a node α, then we say that node β *dominates* node α, meaning that the latter can be fathomed (in case of ties, an appropriate rule has to be taken into account in order to avoid fathoming cycles). Unfortunately, this definition may become useless in the context of general MILPs, where we do not actually know how to perform the dominance test without storing huge amounts of information for all the previously-generated nodes—which is often impractical.

Fischetti and Toth [6] proposed a different dominance procedure that overcomes many of the drawbacks of the classical definition, and resembles somehow the *logic cuts* introduced by Hooker et al. in [10] and the *isomorphic pruning* introduced recently by Margot [15, 16]. Here is how the procedure works.

Let P be the MILP problem

$$P: \quad \min\{c^T x : x \in F(P)\}$$

whose feasible solution set is defined as

$$F(P) := \{x \in \Re^n : \ Ax \le b, \ l \le x \le u, \ x_j \text{ integer for all } j \in J\} \qquad (1)$$

where $J \subseteq N := \{1, \cdots, n\}$ is the index set of the integer variables. For any $J' \subseteq J$ and for any $x' \in \Re^n$, let

$$c(J', x') := \sum_{j \in J'} c_j x'_j$$

denote the contribution of the variables in J' to the overall cost $c^T x'$. Now, let us suppose to solve P by an enumerative (B&B or B&C) algorithm whose branching rule fixes some of the integer-constrained variables to certain values. For every node k of the search tree, let $J^k \subseteq J$ denote the set of indices of the variables x_j fixed to a certain value x_j^k (say). Every solution $x \in F(P)$ such that $x_j = x_j^k$ for all $j \in J^k$ (i.e., belonging to the subtree rooted at node k) is called a *completion* of the partial solution associated at node k.

Definition 1. *Let α and β be two nodes of the search tree. Node β dominates node α if:*

1. $J^\beta = J^\alpha$
2. $c(J^\beta, x^\beta) \le c(J^\alpha, x^\alpha)$*, i.e., the cost of the partial solution at node β is not worse than that at node α*
3. *every completion of the partial solution associated with node α is also a completion of the partial solution associated with node β.*

According to the classical dominance theory, the existence of a node β unfathomed that dominates node α is a sufficient condition to fathom

node α. A key question at this point is: Given the current node α, how can we check the existence of a dominating node β? Fischetti and Toth answered this question by modeling the search of dominating nodes as a structured optimization problem, to be solved exactly or heuristically. For generic MILP models, this leads to the following *auxiliary problem* XP^α:

$$\min \sum_{j \in J^\alpha} c_j x_j$$

$$\sum_{j \in J^\alpha} A_j x_j \leq b^\alpha := \sum_{j \in J^\alpha} A_j x_j^\alpha \tag{2}$$

$$l_j \leq x_j \leq u_j, \quad j \in J^\alpha \tag{3}$$

$$x_j \text{ integer}, \quad j \in J^\alpha \tag{4}$$

If a solution x^β (say) of the auxiliary problem having a cost strictly smaller than $c(J^\alpha, x^\alpha)$ is found, then it defines a dominating node β and the current node α can be fathomed.

It is worth noting that the auxiliary problem is of the same nature as the original MILP problem, but with a smaller size and thus it is often easily solved (possibly in a heuristic way) by a general-purpose MILP solver. In a sense, we are using here the approach of "MIPping the dominance test" (i.e., of modeling it as a MILP), in a vein similar to the recent approaches of Fischetti and Lodi [4] (the so-called *local-branching* heuristic, where a suitable MILP model is used to improve the incumbent solution) and of Fischetti and Lodi [5] (where an ad-hoc MILP model is used to generate violated Chvátal-Gomory cuts). Also note that, as discussed in Section 4, the auxiliary problem gives a sufficient but not necessary condition for the existence of a dominating node, in the sense that some of its constraints could be relaxed without affecting the validity of the approach. In addition, inequalities (2) could be converted to equalities in order to reduce the search space and get a simpler, although possibly less effective, auxiliary problem.

The Fischetti-Toth dominance procedure, called LD (for *Local Dominance*) in the sequel, has several useful properties:

- there is no need to store any information about the set of previously-generated nodes;
- there is no need to make any time-consuming comparison of the current node with other nodes;
- a node can be fathomed even if the corresponding dominating one has not been generated yet;
- the correctness of the enumerative algorithm does not depend on the branching rule; this is a valuable property since it imposes no con-

straints on the B&B parameters—though an inappropriate branching strategy could prevent several dominated nodes to be fathomed;

- the LD test needs not be applied at every node; this is crucial from the practical point of view, as the dominance test introduces some overhead and it would make the algorithm too slow if applied at every node.

An important issue to be addressed when implementing the LD test is to avoid fathoming cycles arising when the auxiliary problem actually has a solution x^β different from x^α but of the same cost, in which case one is allowed to fathom node α only if a tie-break rule is used to guarantee that node β itself is not fathomed for the same reason. In order to prevent these "tautological" fathoming cycles the following criterion (among others) has been proposed in [6]: In case of cost equivalence, define as unfathomed the node β corresponding to the solution found by a *deterministic*[1] exact or heuristic algorithm used to solve the auxiliary problem. Unfortunately this criterion can be of difficult practical application, for two important reasons. First of all, it is not easy to define a "deterministic" algorithm for MILP. In fact, besides the possible effects of randomized steps, the output of the MILP solver typically depends, e.g., on the order in which the variables are listed on input, that can affect the choice of the branching variables as well as the internal heuristics. Moreover, even very simple "deterministic" algorithms may lead to wrong result, as shown in the following example.

Let P be the problem:

$$\begin{cases} & \min -x_1 - x_2 - x_3 - x_4 - 99x_5 \\ \text{s.t.} & x_1 + x_2 \leq 1 \\ & x_3 + x_4 \leq 1 \\ & x_4 + x_5 \leq 1 \\ & x \in \{0,1\}^5 \end{cases}$$

whose optimal solutions are $[1,0,1,0,1]$ and $[0,1,1,0,1]$, and let us consider the enumeration tree depicted in Figure 1. The deterministic algorithm used to perform the LD test is as follows: If the number of variables in the auxiliary problem is smaller than 3, use a greedy heuristic trying to fix variables to 1 in decreasing index order; otherwise use the same greedy heuristic, but in increasing index order.

[1] In this context, an algorithm is said to be *deterministic* if it always provides the same output solution for the same input set.

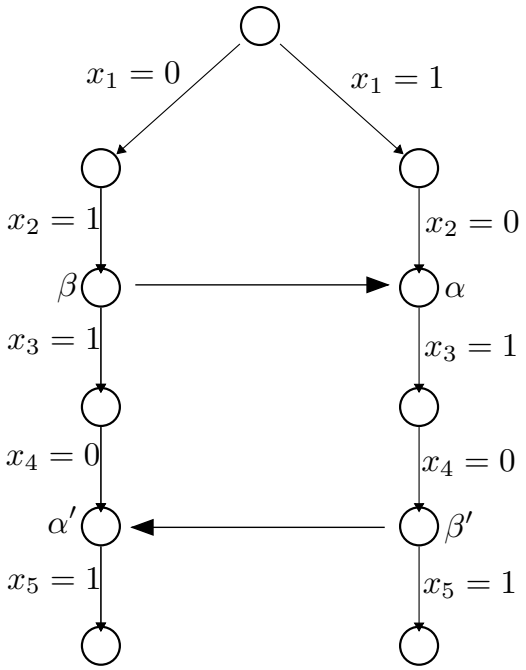

Figure 1: A nasty situation for LD test

When node α (that corresponds to the partial solution $x_1 = 1, x_2 = 0$ with cost -1) is processed, the following auxiliary model is constructed

$$\begin{cases} & \min -x_1 - x_2 \\ \text{s.t.} & x_1 + x_2 \leq 1 \\ & x_1, x_2 \in \{0, 1\} \end{cases}$$

and the deterministic heuristic returns the partial solution $x_2 = 1, x_1 = 0$ of cost -1 associated with node β, so node α is declared to be dominated by β and node α is fathomed assuming (correctly) that node β will survive the fathoming test. However, when the descendant node α' (that corresponds to the partial solution $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0$ with cost -2) is processed, the following auxiliary model is constructed

$$\begin{cases} & \min -x_1 - x_2 - x_3 - x_4 \\ \text{s.t.} & x_1 + x_2 \leq 1 \\ & x_3 + x_4 \leq 1 \\ & x_4 \leq 1 \\ & x_1, x_2, x_3, x_4 \in \{0, 1\} \end{cases}$$

and our deterministic heuristic returns the partial solution $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0$ of cost -2 associated with node β', so node α' is declared to be dominated by β' and node α' is fathomed as well. Therefore, in this case the enumerative algorithm cannot find any of the two optimal solutions, i.e., the LD tests produced a wrong answer.

In view of the considerations above, in our implementation we used a different tie-break rule, also described in [6], that consists in ranking cost-equivalent solutions in lexicographical order ($\prec$). To be more specific, in case of cost ties we fathom node α if and only if $x^\beta \prec x^\alpha$, meaning that the partial solution x^β associated with the dominating node β is lexicographically smaller[2] than x^α. Using this tie-break rule, it is possible to prove the correctness of the overall enumerative method.

Proposition 1. *Assuming that the projection of the feasible set $F(P)$ on the integer variable space is a bounded set, a B&B algorithm exploiting LD with the lexicographical tie-break rule returns the same optimal value as the classical B&B algorithm.*

Proof. Let x^* be the lexicographically minimal optimal solution, whose existence is guaranteed by the boundedness assumption and by the fact that $\prec$ is a well order. We need to show that no node α having x^* among its descendants (i.e. such that $x^*_j = x^\alpha_j$ for all $j \in J^\alpha$) can be fathomed by the LD test. Assume by contradiction that a node β dominating α exists, and define

$$z_j := \begin{cases} x^\beta_j & j \in J^* \\ x^*_j & j \notin J^* \end{cases}$$

where $J^* := J^\beta\,(= J^\alpha)$. In other words, z is a new solution obtained from x^* by replacing its dominated part with the dominating one. Two cases can arise:

1. $c(J^*, x^\beta) < c(J^*, x^\alpha)$: we have

$$c^T z = \sum_{j \in J^*} c_j x^\beta_j + \sum_{j \notin J^*} c_j x^*_j < \sum_{j \in J^*} c_j x^\alpha_j + \sum_{j \notin J^*} c_j x^*_j = c^T x^*$$

and

$$\sum_{j=1}^{n} A_j z_j = \sum_{j \in J^*} A_j x^\beta_j + \sum_{j \notin J^*} A_j x^*_j \leq \sum_{j \in J^*} A_j x^\alpha_j + \sum_{j \notin J^*} A_j x^*_j \leq b$$

so z is a feasible solution with a cost strictly smaller than x^*, which is impossible.

[2]We use the standard definition of lexicographic order on vectors of fixed size over a totally order set.

2. $c(J^*, x^\beta) = c(J^*, x^\alpha)$: using the same argument as in the previous case, one can easily show that z is an alternative optimal solution with $z \prec x^*$, also impossible.

□

It is important to notice that the above proof of correctness uses just two properties of the lexicographic order, namely:

well order: required for the existence of a minimum optimal solution;

inheritance: if x^α and x^β are two partial assignments such that $x^\alpha \prec x^\beta$, then the lexicographic order is not changed if we apply the same completion to both of them.

This observation will be used in section 4 to derive a more efficient tie-break rule.

3 Nogoods and pruning moves

The computational overhead related to the LD test can be reduced considerably if we exploit the notion of nogoods taken from Constraint Programming (CP). A *nogood* is a partial assignment of the problem variables such that every completion is either infeasible (for constraint satisfaction problems) or nonoptimal (for constraint optimization problems). The key observation here is that whenever we discover (through the solution of the auxiliary problem) that the current node α is dominated, we have in fact found a *nogood configuration* $[J^\alpha, x^\alpha]$ that we want to exclude from being re-analyzed at a later time.

Actually, when the LD test succeeds we have not just a dominated partial assignment (x^α), but also a dominating one (x^β). Combining the two we get a *pruning move* ($\Delta = [J^\alpha, x^\beta - x^\alpha]$), i.e., a list of variable changes that we can apply to a (partial) assignment to find a dominating one, provided that the new values assigned to the variables stay within the prescribed bounds. For binary MILPs, the concept of pruning move is equivalent to that of nogood, in the sense that a pruning move can be applied to a partial assignment if and only if the same assignment can be ruled out by a corresponding (single) nogood. For general integers MILPs, however, pruning moves can be much more effective than nogoods. This is easily seen by the following example. Suppose we have two integer variables $x_1, x_2 \in [0, U]$ and that the LD test produces the pair

$$x^\alpha = (1, 0) \quad x^\beta = (0, 1)$$

It is easy to see that, in this case, all the following (dominating, dominated)-pairs are valid:

$$\{((a, b), (a - 1, b + 1)) : a \in [1, U], b \in [0, U - 1]\}$$

The standard LD procedure would need to derive each such pair by solving a different auxiliary problem, while they can be derived all together by solving a single MILP leading to the pruning move $(-1, 1)$.

In our implementation, we maintain explicitly a pool of previously-found pruning moves and solve the following problem (akin to separation for cutting-plane methods) at each branching node α: Find, if any, a move $\Delta = [J', \delta]$ stored in the pool, such that $J' \subseteq J^\alpha$ and $l_j \leq x_j^\alpha + \delta_j \leq u_j$ for all $j \in J'$. If the test is successful, we can of course fathom node α without the need of constructing and solving the corresponding auxiliary problem XP^α.

It is worth noting that we are interested in minimal (with respect to set inclusion) pruning moves, so as to improve effectiveness of the method. To this end, before storing a pruning move in the pool we remove its components j such that $x_j^\alpha = x_j^\beta$ (if any).

It is also worth noting that a move $\Delta^1 = [J^1, \delta^1]$ implies (absorbs) a move $\Delta^2 = [J^2, \delta^2]$ in case

$$J^1 \subseteq J^2 \quad \text{and} \quad |\delta_j^1| \leq |\delta_j^2| \; \forall j \in J^1$$

This property can be exploited to keep the pool shorter without affecting its fathoming power.

At first glance, the use of a move pool can resembles classical state-based dominance tests, but this is really not the case since the amount of information stored is much smaller—actually, it could even be limited to be of polynomial size, by exploiting techniques such as relevance or length bounded nogood recording (see [12]).

4 Improving the auxiliary problem

The effectiveness of the dominance test presented in the previous section heavily depends on the auxiliary problem that is constructed at a given node α. In particular, it is advisable that its solution set is as large as possible, so as to increase the chances of finding a dominating partial solution. Moreover, we aim at finding a partial solution different from (and hopefully lexicographically better than) the one associated with the current node—finding the same solution x^α is of no use within the LD context. For these reasons, we next propose a number of improvements over the original auxiliary problem formulation.

Objective function

The choice of the lexicographic order as a mean to resolve ties, although natural and simple, is not well suited in practice.

In the most naïve implementation, there is a good chance of not finding a lexicographic better solution even if this exists, because we do not convey in any way to the solver the information that we are interested in lex-minimal solution. This is unfortunate, since we risk to waste a great computational effort.

Moreover, the lexicographic order cannot be expressed as a linear objective without resorting to huge coefficients: the only way to enforce the discovery of lexicographically better solutions is through ad-hoc branching and node selection strategies, that are quite intrusive and greatly degrade the efficiency of the solution process.

The solution we propose is to use a randomly generated second-level objective function, while using the lexicographic order only as a last resort, in the unlikely case where both the original and random objective functions yield the same value.

It is worth noting that:

- if we generate the random function at the beginning and keep it fixed for the whole search, then this function satisfies the two properties needed for the correctness of the algorithm;
- in order to guarantee that the optimal solution of the auxiliary problem will be not worse than the original partial assignment, we add the following *optimality* constraint:

$$\sum_{j \in J^\alpha} c_j x_j \leq \sum_{j \in J^\alpha} c_j x_j^\alpha$$

Local Search

As the depth of the nodes in the B&B increases, the auxiliary problem grows in size and becomes heavier to solve. Moreover, we are interested in detecting moves involving only a few variables, since these are more likely to help pruning the tree and are more efficient to search. For these reasons one can heuristically limit the search space of the auxiliary problem to alternative assignments not too far from the current one. To this end, we use a *local branching* [4] constraint defined as follows.

For a given node α, let $B^\alpha \subseteq J^\alpha$ be the (possibly empty) set of fixed binary variables, and define

$$U = \{j \in B^\alpha \mid x_j^\alpha = 1\} \quad \text{and} \quad L = \{j \in B^\alpha \mid x_j^\alpha = 0\}$$

Then we can guarantee a solution x of the auxiliary problem to be different from x^α in at most k binary variables through the following *local branching* constraint

$$\sum_{j \in U} (1 - x_j) + \sum_{j \in L} x_j \leq k$$

A similar reasoning could be extended to deal with general integer variables as well, although in this case the constraint is not as simple as before and requires the addition of certain auxiliary variables [4]. According to our computational experience, a good compromise is to consider a local branching constraint involving only the (binary or general integer) variables fixed to their lower or upper bound, namely

$$\sum_{j \in U} (u_j - x_j) + \sum_{j \in L} (x_j - l_j) \leq k$$

where

$$U = \{j \in J^\alpha \mid x_j^\alpha = u_j\} \quad \text{and} \quad L = \{j \in J^\alpha \mid x_j^\alpha = l_j\}$$

Right-hand side improvement

One could observe that we have been somehow over-conservative in the definition of the auxiliary problem XP^α. In particular, as noticed already in [6], in some cases condition

$$\sum_{j \in J^\alpha} A_j x_j \leq b^\alpha$$

could be relaxed without affecting the correctness of the method.

To illustrate this possibility, consider a simple knapsack constraint $4x_1 + 5x_2 + 3x_3 + 2x_4 \leq 10$ and suppose we are given the partial assignment $[1, 0, 1, *]$. The corresponding constraint in the auxiliary problem then reads $4x_1 + 5x_2 + 3x_3 \leq 7$. However, since the maximum load achievable with the free variables is 2, one can safely consider the relaxed requirement $4x_1 + 5x_2 + 3x_3 \leq 10 - 2 = 8$. Notice that the feasible partial solution $[0, 1, 1, *]$ is forbidden by the original constraint but allowed by the relaxed one, i.e., the relaxation does improve the chances of finding a dominating node. Another example arises for set covering problems, where the constraints are of the form $\sum_{j \in Q_i} x_j \geq 1$. Suppose we have a partial assignment x_j^* $(j \in J')$, such that $k := \sum_{j \in J'} x_j^* > 1$. In this case, the corresponding constraint in the auxiliary problem would be $\sum_{j \in J'} x_j \geq k$, although its relaxed version $\sum_{j \in J'} x_j \geq 1$ is obviously valid as well.

The examples above show however that the improvement of the auxiliary problem may require some knowledge of the particular structure of its constraints. Even more importantly, the right-hand side strengthening procedure above can interfere and become incompatible with the post-processing procedure that we apply to improve the moves. For this reason, in our implementation we decided to avoid any right-hand side improvement.

Local search on incumbents

A drawback of the proposed scheme is that node fathoming is very unlikely at the very beginning of the search. Indeed, at the top of the tree only few variables are fixed and the LD test often fails (this is also true if short moves exist, since their detection depends on the branching strategy used), while the move pool is empty.

To mitigate this problem, each time a new incumbent is found we invoke the following local search phase aimed at feeding the move pool.

- Given the incumbent x^*, we search the neighborhood $N_k(x^*)$ defined through the following constraints:

$$\sum_{j \in J} A_j x_j = \sum_{j \in J} A_j x_j^* \tag{5}$$

$$\sum_{j \in J: x_j^* = u_j} (u_j - x_j) + \sum_{j \in J: x_j^* = l_j} (x_j - l_j) \leq k \tag{6}$$

$$\sum_{j \in J} c_j x_j \leq \sum_{j \in J} c_j x_j^* \tag{7}$$

$$l_j \leq x_j \leq u_j, \quad j \in J$$

$$x_j \text{ integer}, \quad j \in J$$

  In other words, we look for alternative values of the integer variables x_j ($j \in J$) using each constraint—variable bounds excluded—in the same way as x^* (constraint (5)), having a Hamming distance from x^* not larger than k (constraint (6)), and with an objective value not worse than x^* (constraint (7)).

- We populate a solution list by finding multiple solutions to the MILP

$$\min\{\sum_{j \in J} c_j x_j : \ x \in N_k(x^*)\}$$

  by exploiting the multiple-solution mode available in our MILP solver [20].

- Given the solution list $L = (x^1, \ldots, x^p)$, we compare the solutions pairwise and generate a pruning move accordingly, to be stored in the move pool.

- If we find a better solution during the search, we use it to update the incumbent.

It is worth noting that the use of equalities (5) allows us to generate a pruning move for *every* pair of distinct solutions in the list, since for every pair we have a dominating and a dominated solution whose difference produces the pruning move.

5 Implementation

The enhanced dominance procedure presented in the previous sections was implemented in C++ on a Linux platform, and applied within a commercial MILP solver. Here are some implementation details that deserve further description.

An important drawbacks of LD tests is that their use can postpone the finding of a better incumbent solution, thus increasing the number of nodes needed to solve the problem. This behavior is quite undesirable, particularly in the first phase of the search when we have no incumbent and no nodes can be fathomed through bounding criteria. Our solution to this problem is to skip the dominance test until the first feasible solution is found.

The definition and solution of the auxiliary problem at every node of the search tree can become too expensive in practice. We face here a situation similar to that arising in B&C methods where new cuts are typically not generated at every node—though the generated cuts are exploited at each node. A specific LD consideration is that we better skip the auxiliary problem on nodes close to the top or the bottom of the search tree. Indeed, in the first case only few variables have been fixed, hence there is a little chance to find dominating partial assignments. In the latter case, instead, it is likely that the node would be fathomed anyway by standard bounding tests. Moreover, at the bottom of the tree the number of fixed variables is quite large and the auxiliary problem may be quite heavy to solve. In our implementation, we provide two thresholds on tree depth, namely $depth_{\min}$ and $depth_{\max}$, and solve the auxiliary problem for a node α only if $depth_{\min} \leq \text{depth}(\alpha) \leq depth_{\max}$. Moreover, we decided to solve the auxiliary problem at a node only if its depth is a multiple of a given parameter, say *depth_interval*.

In addition, as it is undesirable to spend a large computing time on the auxiliary problem for a node that would have been pruned anyway by the standard B&B rules, we decided to apply our technique just before branching—applying the LD test before the LP relaxation is solved turned out to be less effective.

In order to avoid spending too much computing time on pathologically hard auxiliary MILPs, we also set a node limit N_1 (say) for the auxiliary problem solution, and a node limit N_2 (say) for the local search on incumbents.

Finally, since the discovery of new pruning moves decreases as we proceed with the search, we set a upper bound M on the times the LD test is called: after this limit is reached, the pruning effect is left to the move pool only.

It is important to stress that, although the auxiliary problem is solved only at certain nodes, we check the current partial assignment against the move pool at every node, since this check is relatively cheap.

6 Computational Results

In our computational experiments we used the commercial solver ILOG Cplex 11.0 [20] with default options. All runs were performed on a Intel Q6600 2.4Ghz PC with 4GB of RAM, under Linux.

The definition of the test-bed for testing the potentiality of our approach is of course a delicate issue. As a matter of fact, one cannot realistically expect any dominance relationship to be effective on all types of MILPs. We face here a situation similar to that arising when testing techniques designed for highly-symmetric problems, such as the *isomorphic pruning* proposed recently by Margot [15,16]—although remarkably effective on some classes of problems, the approach is clearly of no use for problems that do not exhibit any symmetry.

Therefore we looked for classes of practically relevant problems whose structure can trigger the dominance relationship, and measured the speedup that can be achieved by using our specific LD procedure. In particular, we next give results on two combinatorial problems: knapsack problems [17] and network loading problems [1, 24]. While the first class of problems is quite natural for dominance tests (and could in principle be solved much more effectively by using specialized codes), the second one is representative of very important applications where the dominance property is hidden well inside the solution structure.

6.1 Knapsack problem

We generated hard single knapsack instances according to the so-called *spanner instance* method in combination with the *almost strongly correlated* profit generation technique; see Pisinger [19] for details.

The parameters of our LD procedure were set to:

$depth_{min}$**:** 5

$depth_{max}$**:** 0.8 times the number of integer variables

$depth_{interval}$**:** 6

k**:** 0.2 times the number of integer variables

N_1**:** 10

N_2**:** 5000

M**:** 1000

The results on hard single knapsack instances with 60 to 90 items are given in Table 1, where labels *"Dominance"* and *"Standard"* refer to the performance of the B&C scheme with and without the LD tests, and label

	Standard			Dominance			Ratio	
Problem	Nodes	Time (s)	Gap	Nodes	Time (s)	Gap	Nodes	Time
kp60_1	311,490	14.70	0.00	1,793	3.45	0.00	173.73	4.26
kp60_2	831,319	43.72	0.00	3,718	3.05	0.00	223.59	14.35
kp60_3	865,469	45.32	0.00	3,995	2.15	0.00	216.64	21.11
kp60_4	1,012,287	47.54	0.00	19,720	6.42	0.00	51.33	7.41
kp70_1	>12,659,538	>1,200.00	0.41	1,634,517	138.68	0.00	7.75	8.65
kp70_2	783,092	41.21	0.00	4,466	6.43	0.00	175.35	6.41
kp70_3	830,794	41.97	0.00	6,396	4.93	0.00	129.89	8.51
kp70_4	>13,226,464	>1,200.00	0.48	27,591	4.49	0.00	479.38	267.18
kp80_1	403,396	18.71	0.00	2,599	9.41	0.00	155.21	1.99
kp80_2	559,447	28.11	0.00	3,118	1.87	0.00	179.42	15.04
kp80_3	576,885	23.46	0.00	2,962	3.40	0.00	194.76	6.90
kp80_4	277,981	13.35	0.00	5,690	3.29	0.00	48.85	4.06
kp90_1	>16,013,282	>1,200.00	0.07	803,333	65.57	0.00	19.93	18.30
kp90_2	18,330,528	863.77	0.00	5,024	3.56	0.00	3,648.59	242.74
kp90_3	>15,273,264	>1,200.00	0.27	37,017	5.61	0.00	412.60	213.78
kp90_4	2,136,389	116.21	0.00	8,056	3.82	0.00	265.19	30.46
Average	>5,255,727	>381.13	-	160,165	16.63	-	398.89	54.45
Geom. mean	>1,761,164	>98.78	-	11,625	6.00	-	151.50	16.47

Table 1: Computational results for hard knapsack instances

"Ratio" refers to the ratios *Standard/Dominance*. For a fair comparison, the same seed was used to initialize the random generator in all runs. The performance figures used in the comparison are the number of nodes of the resulting search tree and the computing time (in CPU seconds). For these problems, the LD tests provided an overall speedup of 23 times and with substantially fewer nodes (the ratio being approximatively 1:33). Note that, in some cases, the ratios reported in the table are just lower bounds on the real ones, as the standard algorithm was stopped before completion due to time limit.

Additional statistics are reported in Table 2 where we provide, for each instance, the final size of the move pool, the percentage of the whole solution time spent either on pool management (*Pool Time*) or LD tests (*LD time*) along with their success rates (*Pool Success* and *LD Success*, respectively). The figures indicate that the number of pruning moves stored in the pool is always manageable. In addition, both the pool checks and the LD tests are rather successful as they allow for node fathoming approximatively 1/3 of the times they are applied—the total effect being of fathoming approximately 2/3 of the nodes. Although the relative overhead introduced by the LD procedure is significant (due to the fact that node relaxations are particularly cheap for knapsack problems), the overall benefit is striking, with an average speed-up of about 2-3 orders of magnitude.

Problem	Pool size	Pool Time	Pool Success	LD Time	LD Success
kp60_1	404	20.08%	38.95%	13.82%	35.00%
kp60_2	196	1.61%	34.97%	29.82%	47.06%
kp60_3	360	14.86%	37.66%	24.45%	16.67%
kp60_4	148	3.48%	38.18%	58.24%	16.80%
kp70_1	330	26.00%	29.62%	1.10%	50.50%
kp70_2	464	18.62%	39.97%	20.67%	40.59%
kp70_3	299	15.11%	40.58%	25.28%	30.43%
kp70_4	324	16.61%	29.31%	37.60%	59.90%
kp80_1	384	20.67%	42.89%	4.99%	37.05%
kp80_2	286	8.45%	37.87%	35.88%	25.45%
kp80_3	833	24.78%	43.96%	9.54%	35.50%
kp80_4	294	15.95%	43.02%	28.13%	25.13%
kp90_1	268	22.32%	37.55%	2.08%	28.70%
kp90_2	705	17.76%	38.56%	34.19%	58.15%
kp90_3	286	16.55%	34.20%	20.88%	41.20%
kp90_4	181	3.85%	33.97%	44.78%	42.20%
Average	360.13	15.42%	37.58%	24.47%	36.90%
Geom. mean	326.24	12.49%	37.33%	16.80%	34.59%

Table 2: Internal statistics for hard knapsack instances

6.2 Network loading problem

Network loading problems arise in telecommunications applications where demand for capacity for multiple commodities has to be realized by allocating capacity to the arcs of a given network. Along with a capacity plan a routing of all commodities has to be determined and each commodity must be routed from source to destination on a single path through the network. The objective is to minimize the costs of the installed capacity in the network, ensuring that all commodities can be routed from source to destination simultaneously.

Given a directed graph $G = (V, A)$, a set of commodities K (each commodity being described by a source node s^k, a destination node t^k, and a demand size d^k), a base capacity unit C and capacity installation costs c_{ij}, our network loading problem can be formulated as:

$$\min \sum_{(i,j)\in A} c_{ij} y_{ij}$$

$$\sum_{j\in V} x_{ij}^k - \sum_{j\in V} x_{ji}^k = \begin{cases} 1 & i = s^k \\ -1 & i = t^k \\ 0 & \text{otherwise} \end{cases} \qquad k \in K, i \in V \tag{8}$$

$$\sum_k d^k x_{ij}^k \le C y_{ij}, \qquad (i,j) \in A \tag{9}$$

$$x_{ij}^k \in \{0,1\}, y_{ij} \in \mathbb{Z}_0^+, \quad k \in K, (i,j) \in A \tag{10}$$

Random instances of the above network loading problem were generated as follows:

- In order to obtain a grid-like structure, we generated grid networks of dimensions 3x5 and 4x4 and randomly deleted arcs with probability 0.1. Each arc has base unit capacity of value 4 and cost 10.
- We generated a commodity of flow 10 for each pair of nodes with probability 0.2.

Some parameters of the LD procedure were changed with respect to the knapsack test-bed, due to the greatly increased size of the instances:

k**:** 0.01 times the number of integer variables

N_1**:** 100

N_2**:** 25000

The results of our experiments are given in Table 3. In this class of problems, the dominance procedure is still quite effective, with an overall speedup of 4 times and a node ratio of more than 5.

More statistics are reported in Table 4. The format of the table is the same as for knapsack problems. As expected, both the computational overhead of the tests and their rate of success are significantly smaller than in the knapsack case, but still very satisfactory.

6.3 Pool effectiveness

Finally, we tested the effectiveness of the main improvements we proposed to the original Fischetti-Toth scheme. Results are reported in Table 5 and Table 6 for knapsack and network loading instances, respectively. We compared our final code (*Dominance*) against two versions of the same code obtained by disabling the use of the move pool (versions *LD1* and *LD2*) and a version without the local search on incumbents (version *LD3*). We provide the absolute performance figures for *Dominance*, while we give relative performance for the other versions—the numbers in the table give the slowdown factors of the various versions with respect to our final code (the larger the worse). According to the tables, disabling the move pool while

	Standard			Dominance			Ratio	
Problem	Nodes	Time (s)	Gap	Nodes	Time (s)	Gap	Nodes	Time
g_15_17_43	1,954,292	797.01	0.00	242,693	163.80	0.00	8.05	4.87
g_15_17_45	>8,711,335	>3,600.00	0.34	1,544,646	845.00	0.00	5.64	4.26
g_15_17_51	>8,022,870	> 3,600.00	0.29	963,576	545.14	0.00	8.33	6.60
g_15_18_35	3,764,325	1,559.38	0.00	286,539	172.48	0.00	13.14	9.04
g_15_18_37	3,959,652	1,525.63	0.00	567,899	279.64	0.00	6.97	5.46
g_15_18_39	752,035	251.52	0.00	303,667	146.55	0.00	2.48	1.72
g_15_18_40	>10,156,564	>3,600.00	0.48	1,071,922	493.57	0.00	9.48	7.29
g_15_19_43	1,609,434	886.51	0.00	415,472	294.61	0.00	3.87	3.01
g_16_18_48	581,268	226.13	0.00	122,824	86.65	0.00	4.73	2.61
g_16_18_53	6,425,061	3,183.84	0.00	6,489	56.14	0.00	990.15	56.71
g_16_19_51	>7,222,780	>3,600.00	0.43	3,774,093	2,158.96	0.00	1.91	1.67
g_16_20_47	5,593,517	3,436.69	0.00	587,773	449.83	0.00	9.52	7.64
g_16_20_51	2,229,792	1,394.58	0.00	272,355	257.26	0.00	8.19	5.42
g_16_21_40	>6,187,221	>3,600.00	0.37	2,334,537	1,524.71	0.00	2.65	2.36
g_16_21_44	1,079,588	717.43	0.00	151,869	182.00	0.00	7.11	3.94
g_16_21_52	>4,565,496	>3,600.00	0.27	1,279,007	1,186.96	0.00	3.57	3.03
Average	>4,550,951.88	>2,223.67	-	870,335.06	552.71	-	67.86	7.85
Geom. mean	>3,354,264	>1,621.25	-	436,484	339.43	-	7.68	4.78

Table 3: Computational results for network loading problems

Problem	Pool size	Pool Time Ratio	Pool Success	LD Time Ratio	LD Success
g_15_17_43	48	0.73%	13.39%	26.20%	1.50%
g_15_17_45	61	1.08%	17.80%	8.43%	0.80%
g_15_17_51	167	2.19%	17.14%	5.79%	1.30%
g_15_18_35	55	0.94%	16.64%	15.84%	1.10%
g_15_18_37	53	1.15%	4.04%	3.27%	0.50%
g_15_18_39	108	1.52%	12.22%	16.93%	1.20%
g_15_18_40	89	1.66%	18.90%	5.17%	2.60%
g_15_19_43	135	1.68%	18.51%	8.97%	3.30%
g_16_18_48	78	0.95%	9.81%	28.68%	0.90%
g_16_18_53	84	0.17%	3.19%	59.68%	0.60%
g_16_19_51	178	2.45%	22.92%	1.49%	1.70%
g_16_20_47	56	0.92%	8.63%	8.76%	0.80%
g_16_20_51	69	0.81%	15.32%	18.28%	2.60%
g_16_21_40	67	0.92%	11.46%	2.77%	1.90%
g_16_21_44	108	0.78%	7.25%	25.27%	1.00%
g_16_21_52	71	0.77%	13.24%	5.58%	2.30%
Average	89.19	1.17%	13.15%	15.07%	1.51%
Geom. mean	82.13	1.01%	11.70%	9.89%	1.31%

Table 4: Internal statistics for network loading instances

retaining the limit M on the number of times the LD test is actually called (version *LD1*) is disastrous: not only we loose the fathoming effect on a large part of the tree, but we waste a large computing time in solving auxiliary problems discovering a same pruning move (or even a dominated one) over and over. Better results can be obtained if we remove limit M (version *LD2*): in this way we retain much of the fathoming effect, but at a much greater computational effort (*LD2* is about 5 times slower than the default version on knapsack problems, and reaches the 1-hour time limit in 11 out of 16 instances on network problems). The contribution of the local search on incumbents (version *LD3*) is more difficult to evaluate: while the number of nodes is always reduced by its use, the overall computing time is reduced only for network problems. A closer look to the single table entries shows however that local search is not effective only for easy problems, where its overhead is not balanced by the increased fathoming power, but it turns out to be very useful on harder instances (e.g., *kp70_1* or *kp90_1*).

	Dominance		LD1 ratios		LD2 ratios		LD3 ratios	
Problem	Nodes	Time (s)	Nodes	Time	Nodes	Time	Nodes	Time
kp60_1	1,793	3.45	104.23	3.09	3.66	1.36	1.20	0.23
kp60_2	3,718	3.05	179.47	11.96	4.61	4.77	1.56	0.76
kp60_3	3,995	2.15	143.03	14.41	5.01	4.67	1.97	0.82
kp60_4	19,720	6.42	19.34	3.07	4.33	10.45	1.67	0.85
kp70_1	1,634,517	138.68	>8.16	>8.65	>1.26	>8.65	7.21	6.25
kp70_2	4,466	6.43	133.94	5.05	5.58	2.70	1.26	0.30
kp70_3	6,396	4.93	117.94	7.94	5.31	4.43	1.35	0.44
kp70_4	27,591	4.49	>500.77	>267.18	2.85	17.85	2.33	1.36
kp80_1	2,599	9.41	105.96	1.46	4.97	0.72	1.38	0.11
kp80_2	3,118	1.87	103.60	9.35	4.19	4.70	1.30	0.69
kp80_3	2,962	3.40	119.87	4.49	5.14	1.79	1.44	0.26
kp80_4	5,690	3.29	39.23	3.56	4.78	4.36	1.54	0.70
kp90_1	803,333	65.57	>20.06	>18.30	10.66	16.30	14.49	10.52
kp90_2	5,024	3.56	3,354.31	221.01	4.45	4.02	1.58	0.67
kp90_3	37,017	5.61	>432.23	>213.78	3.67	20.43	1.58	0.94
kp90_4	8,056	3.82	240.35	27.55	3.32	4.58	1.00	0.64
Average	-	-	>351.41	>51.30	>4.61	>6.99	2.68	1.60
Geom. mean	-	-	>116.83	>13.14	>4.26	>4.85	1.88	0.74

Table 5: Comparison of different LD versions on knapsack problems: LD1 is the version without the move pool and with the same limit M on the number of times the LD test is called, while LD2 is still without the move pool, but which no such limit. LD3 is the version without local search on the incumbents. As in the previous table, label *Dominance* refers to the default LD version. > indicates a reached time limit.

	Dominance		LD1 ratios		LD2 ratios		LD3 ratios	
Problem	Nodes	Time (s)	Nodes	Time	Nodes	Time	Nodes	Time
g_15_17_43	242,693	164	4.36	2.73	>1.97	>21.98	0.82	0.90
g_15_17_45	1,544,646	845	>5.38	>4.26	>0.22	>4.26	1.26	1.22
g_15_17_51	963,576	545	>8.38	>6.60	>0.55	>6.60	1.99	1.82
g_15_18_35	286,539	172	6.80	4.89	1.48	15.78	1.08	1.02
g_15_18_37	567,899	280	2.13	1.77	1.25	12.43	0.47	0.55
g_15_18_39	303,667	147	2.19	1.63	1.11	7.99	0.97	0.89
g_15_18_40	1,071,922	494	>9.46	>7.29	>0.65	>7.29	1.44	1.35
g_15_19_43	415,472	295	3.39	2.68	1.04	10.83	0.98	0.93
g_16_18_48	122,824	87	9.51	5.49	>4.17	>41.55	4.01	2.63
g_16_18_53	6,489	56	>1,126.00	>64.13	28.33	33.69	3.84	1.07
g_16_19_51	3,774,093	2,159	>1.93	>1.67	>0.11	>1.67	1.65	1.57
g_16_20_47	587,773	450	>10.07	>8.00	>1.05	>8.00	2.96	2.61
g_16_20_51	272,355	257	3.46	2.43	>1.83	>13.99	1.63	1.35
g_16_21_40	2,334,537	1,525	>2.66	>2.36	>0.17	>2.36	2.42	2.36
g_16_21_44	151,869	182	7.18	4.25	>3.04	>19.78	1.76	1.31
g_16_21_52	1,279,007	1,187	>3.57	>3.03	>0.23	>3.03	1.01	0.97
Average	-	-	>75.40	>7.70	>2.95	>13.20	1.77	1.41
Geom. mean	-	-	>6.49	>4.14	>1.00	>9.24	1.51	1.29

Table 6: Comparison of different LD versions on network problems: LD1 is the version without the move pool and the same M as the default LD version, while LD2 is still without the move pool, but which no such limit. LD3 is the version without local search on the incumbents. As in the previous table, label *Dominance* refers to the default LD version. > indicates a reached time limit (for LD1 and LD2 the time limit is reached so often that the means are seriously underestimated)

7 Conclusions

In this paper we have presented a dominance procedure for general MILPs. The technique is an elaboration of an earlier proposal of Fischetti and Toth [6], with important improvements aimed at making the approach computationally more attractive in the general MILP context. In particular, the use of nogoods and of pruning moves that we propose in this paper turned out to be crucial for an effective use of dominance test within a general-purpose MILP code.

Computational results on knapsack and network loading problems have been presented, showing that the method can lead to speedups of up to 2 orders of magnitude on hard MILPs whose structure is amenable to dominance.

In our view, a main contribution of our work is the innovative use of improving moves. In the classical (yet computationally impractical) test set approach, these moves are used within a primal heuristic scheme leading eventually to an optimal solution. In our approach, instead, we heuristically generate improving moves on small subsets of variables by solving, on the fly, small MILPs. These moves are not used to improve the incumbent, as in the classical test set environment, but rather to fathom nodes in the enumeration tree—hence the name "pruning moves". If implemented in a proper way, this approach introduces an acceptable overhead even if embedded in a highly-efficient commercial MILP solver such as ILOG Cplex 11, and may produce a drastic reduction in the number of nodes to be enumerated when solving hard MILPs whose structure is amenable to dominance.

Further developments, mainly in the directions of adaptive parameter tuning and a tighter CP integration, are likely to make pruning moves even more appealing and practically useful in general purpose MILP or CP solvers.

Acknowledgments

This work was supported by the University of Padova "Progetti di Ricerca di Ateneo", under contract no. CPDA051592 (project: Integrating Integer Programming and Constraint Programming) and by the Future and Emerging Technologies unit of the EC (IST priority), under contract no. FP6-021235-2 (project ARRIVAL).

References

[1] A. Atamturk and D. Rajan. On splittable and unsplittable flow capacitated network design arc-set polyhedra. *Mathematical Programming*, (92):315–333, 2002.

[2] R. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. MIP: Theory and Practice - Closing the Gap. In *Proceedings of the 19th IFIP TC7 Conference on System Modelling and Optimization*, pages 19–50, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[3] G. Codato and M. Fischetti. Combinatorial Benders' cuts. In *IPCO 2005 Proceedings*, pages 178–195, 2004.

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

[5] M. Fischetti and A. Lodi. Optimizing over the first chvàtal closure. In M. Jünger and V. Kaibel, editors, *Integer Programming and Combinatorial Optimization, 11th International IPCO Conference, Berlin, Germany, June 8-10, 2005, Proceedings*, volume 3509 of *Lecture Notes in Computer Science*, pages 12–22. Springer, 2005.

[6] M. Fischetti and P. Toth. A New Dominance Procedure for Combinatorial Optimization Problems. *Operations Research Letters*, 7:181–187, 1988.

[7] R. Garfinkel and G. Nemhauser. *Integer Programming*. John Wiley & Sons, New York, 1972. Series in Decision and Control.

[8] J. E. Graver. On the foundations of linear and integer linear programming i. *Mathematical Programming*, 9(1):207–226, 1975.

[9] J. N. Hooker and H. Yan. Logic circuit verification by Benders' decomposition. In *V. Saraswat and P. Van Hentenryck (eds.), Principles and Practice of Constraint Programming*, pages 267–288, Cambridge, MA, USA, 1995. The Newport Papers, MIT Press.

[10] J. N. Hooker, H. Yan, I. E. Grossmann, and R. Raman. Logic cuts for processing networks with fixed charges. *Computers & OR*, 21(3), 1994.

[11] T. Ibaraki. The Power of Dominance Relations in Branch-and-Bound Algorithms. *J. ACM*, 24(2):264–279, 1977.

[12] R. J. B. Jr. and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI/IAAI, Vol. 1*, pages 298–304, 1996.

[13] H. J. Kim and J. N. Hooker. Solving fixed-charge network flow problems with a hybrid optimization and constraint programming approach. *Annals of Operations Research*, 115:95–124, 2002.

[14] W. H. Kohler and K. Steiglitz. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *J. ACM*, 21(1):140–156, 1974.

[15] F. Margot. Pruning by isomorphism in branch-and-cut. *Mathematical Programming*, 94(1):71–90, 2002.

[16] F. Margot. Exploiting orbits in symmetric ILP. *Mathematical Programming*, 98(1–3):3–21, 2003.

[17] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, New York, 1990.

[18] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover, 1982.

[19] D. Pisinger. Where are the hard knapsack problems? *Comput. Oper. Res.*, 32(9):2271–2284, 2005.

[20] I. S.A. *CPLEX: ILOG CPLEX 11.0 User's Manual and Reference Manual*, 2007. `http://www.ilog.com`.

[21] H. E. Scarf. Neighborhood systems for production sets with indivisibilities. *Econometrica*, 54(3):507–532, 1986.

[22] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, 1998.

[23] R. Thomas and R. Weismantel. Test sets and inequalities for integer programs. *Integer Programming and Combinatorial Optimization*, pages 16–30, 1996.

[24] S. P. van Hoesel, A. M. Koster, R. L. van de Leensel, and M. W. Savelsbergh. Polyhedral results for the edge capacity polytope. *Mathematical Programming*, 92:335–358, 2002.