A Local Dominance Procedure for Mixed-Integer Linear Programming

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 22, 2007

Abstract

Among the hardest Mixed-Integer Linear Programming (MILP) problems, the ones that exhibit a symmetric nature are particularly important in practice, as they arise in both theoretical and practical combinatorial optimization problems. A theoretical concept that generalizes the notion of symmetry is that of *dominance*. This concept, although known since a long time, is typically not used in general-purpose MILP codes, due to the intrinsic difficulties arising when using the classical definitions in a completely general context.

In this paper we study a general-purpose dominance procedure proposed in the 80's by Fischetti and Toth, that overcomes some of the main drawbacks of the classical dominance criteria. Both theoretical and practical issues concerning this procedure are considered, and important improvements are proposed. Computational results on a test-bed made of hard (single and multiple) knapsack problems are reported, showing that the proposed method can lead to considerable speedup when embedded in a general-purpose MILP solver.

Keywords: integer programming, constraint programming, dominance procedure, nogoods, knapsack problem

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, 19, 23]. 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]. However, there are still important problems of even small size that are not solved to proven optimality in a reasonable amount of time by the current state-of-the-art commercial MILP solvers. Among these problems, the ones that exhibit a symmetric nature are particularly important in practice as they arise, e.g., in scheduling, routing, packing, and network design problems. For these problems the usual Branch & Bound (B&B) and Branch & Cut (B&C) methods are not effective, as the implicit symmetry implies

that it is not possible to prune large subtrees of the enumeration tree by using bounding criteria.

A technique for handling highly-symmetric problems called *isomorphic pruning* has been proposed recently by François Margot [15, 16]. This technique exploits the symmetry group of an Integer Linear Programming (ILP) problem to prune nodes of the search tree and to generate cuts. Although remarkably effective on some classes of problems, this method has two main drawbacks, namely: (1) one needs to compute *a priori* the symmetry group of the ILP problem at hand; and (2) one is no longer free to choose the branching strategy within the ILP solver. Possible remedies have been proposed very recently in [11] and [18].

A theoretical concept that generalizes the notion of symmetry is that of *dominance*. 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 [9]. However, although known since a long time, dominance criteria are not fully exploited in general-purpose MILP codes, due to number of important limitations of the classical definition. In particular, it is not easy to compute a dominance relationship for a generic MILP problem, and the naïve application of the classical concept requires the storage of huge "state" information that makes it impractical.

In this paper we study a 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 is local in nature, 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 build up an auxiliary problem XP^α that looks for a new partial assignment involving the variables in J^α 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.

The present 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 concept of *nogood*, borrowed from Constraint Programming (CP), that turns out to be of crucial importance for the practical effectiveness of the overall dominance procedure. Briefly speaking, a *nogood* is a partial assignment of the variables such that every completion is either infeasible (for constraint satisfaction problems) or non-optimal (for constraint optimization problems). Though widely used in the CP context, the concept of nogood is quite new in mathematical programming. One of the the first uses of nogoods in ILP was to solve verification problems [8] and fixed-charge network flow problems [13]. Further applications can be found in [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. These nogoods are stored in a *pool*, that is checked before constructing and solving the auxiliary problem at a given node, with a typically substantial computational saving. In Section 4 we deal with some extensions of the basic scheme. Implementation issues are described in Section 5. Computational results obtained on hard knapsack instances are given in Section 6, showing that the proposed method can lead to a significant speedup when embedded within a general-purpose MILP solver. Some conclusions are finally drawn in Section 7.

2 The Fischetti-Toth dominance procedure

In the standard B&B (or B&C) framework, a node is fathomed in two situations:

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

There is however a third way of pruning a node, by using the concept of *dominance*. According to [19], 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 (and more "local") dominance procedure which overcomes many of the drawbacks of the classical definition, and resembles somehow the *isomorphic-pruning* introduced recently by Margot [15]. 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

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

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. $c(J^\beta, x^\beta) \leq c(J^\alpha, x^\alpha)$*, 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.*

Clearly, 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*:

$$\begin{cases} XP^\alpha: & \min \sum_{j \in J^\alpha} c_j x_j \\ \text{s.t.} & \sum_{j \in J^\alpha} A_j x_j \leq b^\alpha := \sum_{j \in J^\alpha} A_j x_j^\alpha \\ & x_j \text{ integer for all } j \in J^\alpha \end{cases} \tag{2}$$

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.

The Fischetti-Toth dominance procedure, called *Local Dominance* (LD) procedure in the sequel to stress its local nature, 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 constraints 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 misleading 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.

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

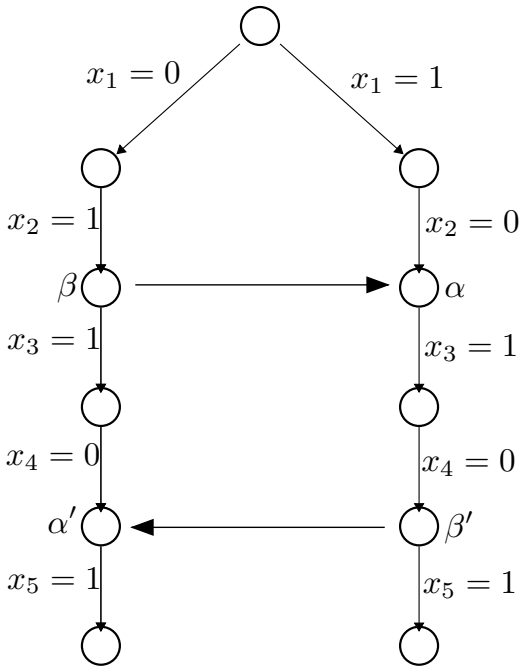

Figure 1: A nasty situation for LD test

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.

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, the 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:

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

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

$$c^T z = \sum_{j \in J^*} c_j x_j^\beta + \sum_{j \notin J^*} c_j x_j^* < \sum_{j \in J^*} c_j x_j^\alpha + \sum_{j \notin J^*} c_j x_j^* = c^T x^*$$

and

$$\sum_{j=1}^{n} A_j z_j = \sum_{j \in J^*} A_j x_j^\beta + \sum_{j \notin J^*} A_j x_j^* \leq \sum_{j \in J^*} A_j x_j^\alpha + \sum_{j \notin J^*} A_j x_j^* \leq b$$

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

2. $c(J^*, x^\beta) = c(J^*, x^\alpha)$: 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 Borrowing nogoods from Constraint Programming

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 non-optimal (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.

There are two possible ways of exploiting nogoods in the context of MILP solvers:

- Generate a constraint cutting the nogood configuration off, so as to prevent it appears again in later solutions. This is always possible (at least for binary variables) through a local branching constraint [4], and leads to the so-called *combinatorial Benders' cuts* studied by Codato and Fischetti [3].

- Maintain explicitly a pool of previously-found nogood configurations and solve the following problem (akin to separation) at each node α to be tested: Find, if any, a nogood configuration $[J', x']$ stored in the pool, such that $J' \subseteq J^\alpha$ and $x'_j = x^\alpha_j$ 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 auxiliary problem XP^α. This method can cope very easily with general-integer variables.

In our implementation we use the nogood pool option, that according to our computational experience outperforms the cut options. It is worth noting that we are interested in minimal (with respect to set inclusion) nogoods, so as to improve both for efficiency and effectiveness of the method. Indeed, if node α dominates node β and $J' := \{j \in J^\alpha : x^\alpha_j \neq x^\beta_j\}$, then clearly the restriction of x^α onto J' dominates the restriction of x^β onto J'. As a result, if the auxiliary problem produces a dominating assignment x^α, we can safely remove from the nogood configuration all components j such that $x^\alpha = x^\beta$.

If applied at every node, our procedure guarantees automatically the minimality of the nogood configurations found. If this is not the case, instead, minimality is no longer guaranteed, and can be enforced by a simple post-processing step before storing any new nogood in the pool.

At first glance, the use of a nogood 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 [10]).

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 crucial 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 implemented 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 for an effective solution of the auxiliary problems.

In the most naïve implementation, there is a good chance of not finding a lex-better solution even if it exists, because we are not telling the solver in any way that we are interested in a lex-minimal solution. This is unfortunate, since we risk to spoil 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 lex-better solutions is through ad-hoc branching and node selection strategies, which greatly degrades the efficiency of the solution process (and are quite intrusive).

The solution we propose is to use a randomly generated second-level objective function and use the lexicographic order only as a last resort, in the unlikely case where both the objectives (original and random) 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 (for a minimization problem):

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

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 nogood configurations 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 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 some auxiliary variables [4].

Right-hand-side improvement

In the construction of the auxiliary problem XP^α we have been quite conservative in the definition of the constraints. In particular, as noticed already in [6], condition

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

is a very restrictive requirement that can often 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 suggest that the improvement of the auxiliary problem requires some knowledge of the particular structure of its constraints and can be time consuming for the general linear constraint. For this reason, we propose the following very simple rule, which is fast and does not require the analysis of the constraint structure: *If all variables involved in a constraint have been fixed, use the right hand side value of the original problem in the auxiliary one.*

Although simple, this rule proved quite effective in practice and it is used by default in our implementation.

5 Implementation

The enhanced dominance procedure presented in the previous sections was implemented in C++ on a Linux platform, and applied within a simple B&B scheme. We decided not to use a more elaborated B&C scheme because our primary interest was to test the impact of the LD procedure within a "clean" environment without unpredictable and uncontrollable side effects—as is often the case when using more sophisticated MILP solvers. A similar design decision was taken, e.g., in [12] when testing the effectiveness of branching on general disjunctions. Here are some implementation issues that deserve further description.

Framework

We have implemented three different branching strategies:

maximum integrality violation: choose the fractional variable closer to 0.5 (take the one with the smallest index in case of ties).

minimum index branching: branch on the fractional variable with smallest index.

pseudocosts: standard pseudocosts branching as described in [1].

The simple minimum index branching strategy proved the most effective on the classes of problems on which we tested the dominance procedure, so it was the one chosen for the benchmarks.

As to node selection, we use a mix of plunging/best bound strategies [1]. In particular, we choose a child or sibling of the current node whenever possible (plunging), and revert to the best bound strategy on backtracking.

At each node, the linear programming relaxation of the original MILP is solved by a commercial LP solver; primal simplex is used on the root node, dual simplex on all other nodes.

No specific primal heuristic was implemented; however, at the beginning of the root node we simply run a commercial MILP solver with a short time limit to hopefully gather a first incumbent solution.

We did not perform preprocessing (bound strengthening, variable aggregation/substition, etc.) before starting our B&B algorithm. Rather, we apply the so-called *AC-3 constraint propagation* [**?**] scheme at every node of the tree; this propagation consists of bound strengthening on the variables as described in [22].

Finally we apply reduced cost fixing at each node.

LD parameters

One of the main 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 algorithm, when we have no incumbent and no nodes can be fathomed through bounding criteria. A practical 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 a few variables have been fixed, hence there are little chances to find dominating partial assignments. In the latter case, instead, it is likely that the node would be pruned 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 the tree depth of the nodes, namely $depth_{\min}$ and $depth_{\max}$, and a node α is tested for dominance only if $depth_{\min} \leq \text{depth}(\alpha) \leq depth_{\max}$.

In addition, as it 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 only just before branching—applying the LD test before the LP relaxation is solved is not worthwhile in practice.

Finally, we decided to solve the auxiliary problem at every k-th node (k being a fixed *skip factor* parameter), and we set a limit on the computing time spent by the black-box MILP solver used for solving it.

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

6 Computational Results

In our computational experiments we used the commercial code ILOG Cplex 10.0 [21] to solve the linear relaxations and the auxiliary problems, with default options. All runs were performed on a AMD Athlon64 X2 4200+ PC with 4GB of RAM, under Linux, with a node limit of 2,000,000 nodes.

As to the LD procedure, we used the following parameters:

- $depth\_min = 0.2$ times the total number of integer variables;
- $depth\_max = 0.7$ times the total number of integer variables;
- $k = 5$ (skip factor)

- $neighboorhood\_size = 0.2$ times the total number of binary variables, used in the local branching constraint.

The first class of problems we tested was the one used by Margot in [15, 16] to prove the effectiveness of his isomorphic pruning technique. However, our computational results showed a disappointing "zero dominated nodes" result on these instances, which is easily explained by the fact that the symmetry there is inherently global, while LD is a local procedure. Thus we looked for classes of problems where equivalent/dominated partial solutions are as independent as possible of the rest of the assignment. Single and multiple knapsack problems [17] are among these problems. We therefore generated hard single knapsack instances according to the so-called *spanner instances* method in combination with the *almost strongly correlated* profit generation technique; see Pisinger [20] for details. Multiple knapsack instances were generated in a similar way, by choosing a same capacity for all the containers.

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 our B&B scheme with and without the LD tests, and label *"Ratio"* refers to the ratios *Standard/Dominance*. 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 standard B&B scheme could not solve any instance within the imposed node limit. On the other side, the LD tests enabled us to solve all the instances to proven optimality, with an average speedup of 8 times and with substantially fewer nodes (the ratio being 1/40). Note that the ratios reported in the table are just lower bounds on the real ones, because the standard algorithm was stopped before completion.

As to hard multiple knapsack problems, we have generated instances with a number of items ranging from 20 to 25 and a number of knapsacks ranging form 3 to 6. The LD parameters were unchanged. The results on multiple knapsack problems are given in Table 2.

In the multiple knapsack case, the LD procedure was not as effective as in the single case, yet worthwhile to apply, yielding improvements both in the number of nodes processed (75% drop) and in the solution time (average speedup of 3 times).

Finally, we tested the effectiveness of the main improvements we proposed to the original Fischetti-Toth LD scheme. Results on a subset of single and multiple knapsack problems are reported in Table 3, where we compared our final code (*Dominance*) against a code obtained by disabling just one of the main LD features. According to the table, the following comments can be drawn.

- The nogood pool was really crucial for the effectiveness of the method—

	Standard		Dominance		Ratio	
Problem	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time
kp60_1.lp	2,000,000	304	5,453	3	366.77	101.33
kp60_2.lp	2,000,000	292	29,271	32	68.33	9.13
kp60_3.lp	2,000,000	292	14,439	12	138.51	24.33
kp60_4.lp	2,000,000	294	20,207	22	98.98	13.36
kp60_5.lp	2,000,000	293	10,467	7	191.08	41.86
kp70_1.lp	2,000,000	293	443,323	121	4.51	2.42
kp70_2.lp	2,000,000	313	24,061	28	83.12	11.18
kp70_3.lp	2,000,000	304	13,007	8	153.76	38.00
kp70_4.lp	2,000,000	311	42,403	40	47.17	7.78
kp70_5.lp	2,000,000	298	6,057	4	330.20	74.50
kp80_1.lp	2,000,000	309	9,983	5	200.34	61.80
kp80_2.lp	2,000,000	315	11,293	6	177.10	52.50
kp80_3.lp	2,000,000	318	5,145	2	388.73	159.00
kp80_4.lp	2,000,000	317	12,751	7	156.85	45.29
kp80_5.lp	2,000,000	311	19,793	16	101.05	19.44
kp90_1.lp	2,000,000	319	96,257	220	20.78	1.45
kp90_2.lp	2,000,000	326	17,527	15	114.11	21.73
kp90_3.lp	2,000,000	307	27,853	34	71.81	9.03
kp90_4.lp	2,000,000	326	29,987	30	66.70	10.87
kp90_5.lp	2,000,000	340	101,927	159	19.62	2.14
Total	40,000,000	6,182	941,204	771	42.50	8.02

Table 1: Computational results for hard single knapsack instances

	Standard		Dominance		Ratio	
Problem	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time
mkp_20_4_1.lp	359,447	46	100,895	34	3.56	1.35
mkp_20_4_3.lp	140,215	18	51,681	15	2.71	1.20
mkp_20_5_2.lp	2,000,000	555	445,411	162	4.49	3.43
mkp_20_5_3.lp	1,957,569	628	419,003	180	4.67	3.49
mkp_20_6_1.lp	1,926,067	645	499,765	289	3.85	2.23
mkp_25_3_1.lp	62,519	13	19,817	6	3.15	2.17
mkp_25_3_2.lp	67,631	18	25,813	6	2.62	3.00
mkp_25_4_1.lp	171,199	51	63,043	21	2.72	2.43
mkp_25_4_2.lp	2,000,000	520	547,367	131	3.65	3.97
Total	8,684,647	2,494	2,172,795	844	4.00	2.95

Table 2: Computational results for hard multiple knapsack problems

without it, the considerable saving in the number of nodes would actually increase the overall solution time.

- The use of the random objective function was very effective on the single knapsack instances, and slightly worse on the multiple ones.
- The right-hand-side strengthening was very effective on multiple knapsack instances, and obviously useless on the single knapsack instances where fixing every variable in a constraint means actually solving the problem (the slight differences in running times and nodes being due to the random nature of the objective function).
- The local branching constraint added to the auxiliary problem does not appear to play a role for the problems in our test-bed, where the solution of the auxiliary problem is in any case very fast—we expect however that these constraints can turn out to be useful for different classes of problems.

	Standard		LD1 (−nogood)		LD2 (−rand_obj)		LD3 (−rhs_str)		LD4 (−local_branch)		Dominance	
Problem	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time	Nodes	Time (s)	Nodes	Time (s)	Nodes	Time
kp60_1.lp	2,000,000	304	7,269	9	59,115	24	5,649	4	6,585	4	5,453	3
kp60_2.lp	2,000,000	292	77,909	804	162,705	373	33,103	28	31,493	30	29,271	32
kp60_3.lp	2,000,000	292	20,629	68	249,071	545	12,733	9	19,887	14	14,439	12
kp60_4.lp	2,000,000	294	35,313	338	80,747	183	20,365	22	17,847	14	20,207	22
kp60_5.lp	2,000,000	293	16,859	49	205,395	436	11,087	5	10,295	5	10,467	7
kp70_1.lp	2,000,000	293	169,663	3,869	2,000,000	2,768	375,921	125	357,279	132	443,323	121
kp70_2.lp	2,000,000	313	52,415	406	148,807	422	33,223	33	28,489	21	24,061	28
kp70_3.lp	2,000,000	304	14,549	36	254,603	312	10,745	7	11,049	6	13,007	8
kp70_4.lp	2,000,000	311	164,775	847	665,303	1,558	45,217	36	45,341	37	42,403	40
kp70_5.lp	2,000,000	298	8,753	20	291,089	316	6,935	4	7,053	4	6,057	4
Total	20,000,000	2,994	568,134	6,446	4,116,835	6,937	554,978	273	535,318	267	608,688	277
mkp_20_4_1.lp	359,447	46	228,767	195	109,529	28	256,285	82	97,365	34	100,895	34
mkp_20_4_3.lp	140,215	18	74,721	55	60,141	12	107,747	28	63,407	17	51,681	15
mkp_20_5_2.lp	2,000,001	555	1,161,819	1,088	420,483	127	1,992,815	571	532,893	210	445,411	162
mkp_20_5_3.lp	1,957,569	628	1,346,157	1,240	406,305	106	1,705,747	593	443,181	197	419,003	180
mkp_20_6_1.lp	1,926,067	645	756,211	1,135	514,037	289	1,111,899	676	524,667	404	499,765	289
Total	6,383,299	1,892	3,567,675	3,713	1,510,495	562	5,174,493	1,950	1,661,513	862	1,516,755	680

Table 3: Comparison of different LD versions: LD1 is the version without the nogood pool (but with the LD test done at every node); LD2 uses the original objective function instead of its random replacement; LD3 does not strengthen the right-hand-side of the auxiliary problem; and LD4 does not impose the local branching constraint. As in the previous table, label *Dominance* refers to the default LD version with all improvements turned on.

7 Conclusions

In this paper we have presented a local 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 we propose in this paper turned out to be crucial for an effective use of dominance test within a general-purpose MILP code.

Promising computational results on single and multiple knapsack problems have been presented. Further developments, mainly in the directions of adaptive parameter tuning and a tighter CP integration, are likely to make this technique even more appealing and practically worthwhile in a general-purpose MILP or CP solver.

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] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Oper. Res. Lett*, 33(1):42–54, 2005.

[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. 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.

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

[10] 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.

[11] V. Kaibel, M. Peinhardt, and M. E. Pfetsch. Orbitopal fixing. In *IPCO 2007 Proceedings*, 2007.

[12] M. Karamanov and G. Cornuéjols. Branching on general disjunctions. Technical report, Tepper School of Business, 2005.

[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] J. Ostrowski, J. Linderoth, F. Rossi, and S. Smriglio. Orbital branching. In *IPCO 2007 Proceedings*, 2007.

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

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

[21] I. S.A. *CPLEX: ILOG CPLEX 10.0 Users Manual and Reference Manual*, 2006. `http://www.ilog.com`.

[22] M. W. P. Savelsbergh. Preprocessing and probing for mixed integer programming problems. *ORSAjoc*, 6:445–454, 1994.

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