

Nogood Learning for Mixed Integer Programming

Tuomas Sandholm

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
sandholm@cs.cmu.edu

Rob Shields

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
rshields@cs.cmu.edu

Abstract

Nogood learning has proven to be an effective CSP technique critical to success in today’s top SAT solvers. We extend the technique for use in combinatorial optimization problems, as opposed to mere constraint satisfaction. In particular, we examine 0-1 integer programming (0-1 IP). Our technique generates globally valid cutting planes for the 0-1 IP search algorithm from information learned through constraint propagation. Nogoods (cutting planes) are generated not only from infeasibility but also from bounding. All of our techniques are geared toward yielding tighter LP upper bounds, and thus smaller search trees. Experiments suggest that nogood learning does not help in optimization because few cutting planes are generated, and they are weak. We explain why, and identify problem characteristics that affect the effectiveness. We then generalize the technique to mixed-integer programming. Finally, we lay out directions along which the techniques can potentially be made helpful.

1 Introduction

Nogood learning is a powerful technique for reducing search tree size in constraint satisfaction problems (CSPs) (e.g., [Dechter, 1990; Frost and Dechter, 1994; Richards and Richards, 1996; Chai and Kuehlmann, 2003]). Whenever an infeasibility is found, reasoning is used to identify a subset of the variable assignments from the path (the nogood) that caused the infeasibility. The nogood is stored; the rest of the tree search does not have to consider paths that include the assignments of that nogood. Modern complete propositional satisfiability solvers use nogood learning; it enables them to solve orders of magnitude larger problems (e.g., [Marques-Silva and Sakallah, 1999; Moskewicz *et al.*, 2001]).

We present, to our knowledge, the first propagation-based nogood learning methods for optimization problems. Optimization problems are more general than CSPs: they have an objective to be maximized in addition to having constraints that must be satisfied. We focus on the most prevalent optimization framework, mixed integer programming (MIP), which is domain independent and has a very broad range of applications in scheduling, routing, facility location, combinatorial auctions, etc. The high-level perspective is that our techniques hybridize two powerful search paradigms: constraint programming and MIP. Other—complementary—ways of hybridizing the two have been proposed (e.g., [Bockmayr and Kasper, 1998; Hooker *et al.*, 1999; Focacci *et al.*, 1999; Bockmayr and Eisenbrand, 2000; Hooker, 2002]).

A *mixed integer program (MIP)* is defined as follows.

Definition 1 *Given an n -tuple c of rationals, an m -tuple b of rationals, and an $m \times n$ matrix A of rationals, find the n -tuple*

x such that $Ax \leq b$, and $c \cdot x$ is maximized.

If the decision variables are constrained to be integers ($x \in Z^n$ rather than allowing reals), then we have an *integer program (IP)*. If we further require that the decision variables are binary ($x \in \{0, 1\}^n$), then we have a *0-1 IP*. While (the decision version of) MIP is \mathcal{NP} -complete, there are sophisticated techniques that can solve very large instances in practice. We now briefly review those techniques. We build our methods on top of them.

In *branch-and-bound* search, the best solution found so far (*incumbent*) is stored. Once a node in the search tree is generated, an upper bound on its value is computed by solving a relaxed version of the problem, *while honoring the commitments made on the search path so far*. The most common method for doing this is to solve the problem while only relaxing the integrality constraints of all undecided variables; that *linear program (LP)* can be solved fast in practice, e.g., using the simplex algorithm (or a polynomial worst-case time interior-point method). A path terminates if 1) the upper bound is at most the value of the incumbent (search down that path cannot produce a solution better than the incumbent), 2) the LP is infeasible, or 3) the LP returns an integral solution. Once all paths have terminated, the incumbent is optimal.

A more modern algorithm for solving MIPs is *branch-and-cut* search, which first achieved success on the traveling salesman problem [Padberg and Rinaldi, 1987; 1991], and is now the core of the fastest general-purpose MIP solvers. It is like branch-and-bound, except that in addition, the algorithm generates *cutting planes* [Nemhauser and Wolsey, 1999]. They are linear constraints that, when added to the problem at a search node, result in a tighter LP polytope (while not cutting off the optimal integer solution) and thus a lower upper bound. The lower upper bound in turn can cause earlier termination of search paths, thus yielding smaller search trees.

The next section presents our approach. Section 3 presents experiments and explains the performance. Section 4 generalizes our approach from 0-1 IPs to MIP. Section 5 concludes and lays out potentially fruitful future directions.

2 Nogood learning for 0-1 IP

The main idea of our approach (for 0-1 integer programming) is to identify combinations of variable assignments that cannot be part of an optimal solution. Any such combination is a *nogood*. The high-level motivation is that generating and storing nogoods allows the tree search algorithm to avoid search paths that would include the variable assignments of any stored nogood. This reduces search tree size.

To extend nogood learning from CSPs to optimization (IP), there are two challenges: generating nogoods and using them. Each challenge involves subtle and interesting issues. We first present a method for generating nogoods in this setting through constraint propagation. We then present techniques for generating cutting planes for the branch-and-cut algorithm

from those nogoods. Overall, our technique leads to tighter LP bounding, and thus smaller search trees.

2.1 Propagation rules to detect implications

As a building block, we need rules to detect the implications of decisions made on the search path. We therefore present an adaptation of constraint propagation to 0-1 IP.¹

First, consider a simple example: $ax \leq b, a \geq 0, x \in \{0, 1\}$. Clearly, if $b < a$, then $x = 0$. Furthermore, if $b < 0$, then the constraint is not satisfiable by any value of x . More generally, say we have $ax \leq \phi(), x \in \{0, 1\}$, for some function ϕ . If $a \geq 0$, we can reason as follows.

- If the *upper bound* on $\phi()$ is negative, then no assignment of x will satisfy the constraint.
- Otherwise, if a is greater than the upper bound of $\phi()$, then $x \leftarrow 0$.

If $a < 0$ we can make a similar statement:

- If a is greater than the upper bound of $\phi()$, then no assignment of x will satisfy the constraint.
- Otherwise, if the *upper bound* on $\phi()$ is negative, then $x \leftarrow 1$.

This is central to our constraint propagation scheme. Each time a variable is fixed, we loop through all constraints and check to see whether any of the above conditions are met. If a constraint is deemed to be unsatisfiable, then we have found a conflict, and there cannot exist a feasible solution in this node's subtree. If we have found no conflicts, but have instead proven that a variable must be fixed to satisfy the constraint, then we propagate that change as well.

The rest of this subsection lays out this procedure in more detail. Each IP constraint i can be written as $\sum_{j \in N} a_{ij}x_j \leq b_i$, where N is the index set of variables. In order to examine a particular variable $x_{\hat{j}}$ with respect to constraint i , the constraint can be rewritten as $a_{i\hat{j}}x_{\hat{j}} \leq b_i - \sum_{j \in N \setminus \hat{j}} a_{ij}x_j$. This is the same form as the inequality examined above. Now,

$$\begin{aligned} \phi_{i\hat{j}}(x) &= b_i - \sum_{j \in N, j \neq \hat{j}} a_{ij}x_j \\ &= b_i - \sum_{j \in N_i^+, j \neq \hat{j}} |a_{ij}|x_j + \sum_{j \in N_i^-, j \neq \hat{j}} |a_{ij}|x_j \end{aligned}$$

where $N_i^+ = \{j \in N : a_{ij} > 0\}$, $N_i^- = \{j \in N : a_{ij} < 0\}$.

If we can determine an upper bound U_{ij} for this expression, we can use the above process to perform constraint propagation on the IP. The expression

$U_{i\hat{j}} = b_i - \underline{s}_i(\{j | j \in N_i^+, j \neq \hat{j}\}) + \overline{s}_i(\{j | j \in N_i^-, j \neq \hat{j}\})$ yields an upper bound as long as $\underline{s}_i(S) \leq \sum_{j \in S} |a_{ij}|x_j \leq \overline{s}_i(S)$ for all x .

With no other knowledge of the problem structure, we can use $\underline{s}_i(S) = \sum_{j \in S} |a_{ij}|l_j$, $\overline{s}_i(S) = \sum_{j \in S} |a_{ij}|u_j$, where l_j and u_j are the lower and upper bounds on x_j , respectively, at the current node of the search tree. Since we are dealing with 0-1 IP, $l_i = 0$ and $u_i = 1$ unless the variable x_i has been fixed. If x_i has been fixed, then $l_i = u_i = x_i$.

We can now state the constraint propagation procedure:²

¹Propagation of linear constraints has been explored previously, in the context of bounds consistency [Harvey and Schimpf, 2002].

²This is very similar to that used for nogood learning in CSPs. It can be sped up by watching the set of variables that are candidates to become implied shortly [Chai and Kuehlmann, 2003].

for all unsatisfied³ constraints i do

for all unfixed⁴ variables \hat{j} do

if $\hat{j} \in N_i^+$ then

if $U_{i\hat{j}} < 0$ then we have detected a *conflict*

else if $a_{i\hat{j}} > U_{i\hat{j}}$ then $u_{\hat{j}} \leftarrow 0$

else if $\hat{j} \in N_i^-$ then

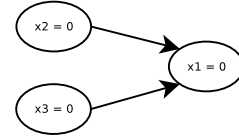
if $a_{i\hat{j}} > U_{i\hat{j}}$ then we have detected a *conflict*

else if $U_{i\hat{j}} < 0$ then $l_{\hat{j}} \leftarrow 1$

2.2 Implication graph and its maintenance

We also need a way to track the implications that have been made during the search path. For example, say the search has taken a branch $x_2 = 0$ and a branch $x_3 = 0$. Say the constraint propagation process then comes across constraint $x_1 - x_2 - x_3 \leq 0$. Clearly, x_1 must be 0 because $\langle x_2 = 0, x_3 = 0 \rangle$. However, we would like to capture more than just $x_1 = 0$; we would also like to capture the fact that the assignment $x_1 = 0$ was due to $\langle x_2 = 0, x_3 = 0 \rangle$.

To keep track of implications and their causes, our algorithm constructs and maintains⁵ an *implication graph*, a directed graph, in much the same way as a modern DPLL SAT solver. We add a node to it for each variable assignment (either due to branching or to implication). We also add a node whenever we detect a conflict. Denote by i the constraint that caused the assignment or conflict by implication. For each fixed variable x_j with a nonzero coefficient in i , we add an edge from the node corresponding to x_j to the node we just created. At this point our implication graph looks as follows.



2.3 Nogood identification and cutting plane generation

Whenever a conflict is detected (i.e., the node is ready to be pruned), we use the implication graph to identify *nogoods*, i.e., combinations of variable assignments that cannot be part of any feasible solution. Consider drawing a cut in the implication graph which separates all decision nodes from the conflict node. For every edge which crosses the cut, take the assignment from the source node of the edge. The resulting set of assignments cannot result in a feasible solution; the conflict will always be implied. Therefore, this set of assignments constitutes a nogood. Any such cut will produce a nogood; several methods for finding strong cuts have been studied by the SAT community (e.g., [Marques-Silva and Sakallah, 1999; Moskewicz *et al.*, 2001]) and can be applied in our setting directly. (In the experiments, we use the UIP technique to generate a nogood.)

Finally, we will use the identified nogood(s) to produce cutting plane(s) for the 0-1 IP problem. (These cuts are *global*, that is, they are valid throughout the search tree, not

³A constraint is *unsatisfied* if it is not yet guaranteed to be true given the set of fixed/implied variables at the current node.

⁴A variable is *unfixed* if $l_j < u_j$.

⁵This is easy to maintain with a single graph if depth-first search order is used. For search algorithms in the breadth-first family, such as A* (aka. best-first search), a separate graph is maintained for each active search path (i.e., each node on the open list).

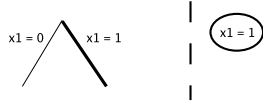
only in the current subtree. Thus it is not necessary to remove them as the search moves outside of the subtree.) We break the variables involved in the nogood into two sets: V_0 contains the variables that are fixed to 0 (by branching or implication), and V_1 contains the variables that are fixed to 1. Consider the case where all variables involved in the nogood were fixed to 0; we would like to constrain the problem so that at least one of those variables is nonzero: $\sum_{j \in N_0} x_j \geq 1$. Conversely, if all the variables involved in the nogood were fixed to 1, then we would like to constrain the problem so that for at least one variable, the complement of the variable is nonzero: $\sum_{j \in N_1} (1 - x_j) \geq 1$. Putting these together, a nogood generates the cutting plane $\sum_{j \in V_0} x_j - \sum_{j \in V_1} x_j \leq 1 - |V_1|$.

2.4 A small example

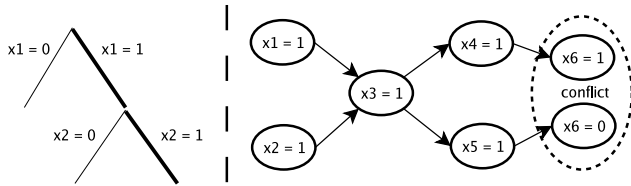
For illustration of the concepts, consider the following 0-1 IP.

$$\begin{array}{rcll}
 \max & x_1 & +1.1x_2 & +1.2x_3 & +x_4 & +x_5 & +x_6 & & \\
 \text{s.t.} & -x_1 & -x_2 & & & & & & -1 \\
 & & & +x_3 & & & & & 0 \\
 & & & -x_3 & +x_4 & & & & 0 \\
 & & & -x_3 & & +x_5 & & & 0 \\
 & & & & -x_4 & +x_6 & & & 0 \\
 & & & & -x_5 & -x_6 & & & -1 \\
 & & & & & & x_j \in & \{0, 1\} &
 \end{array}$$

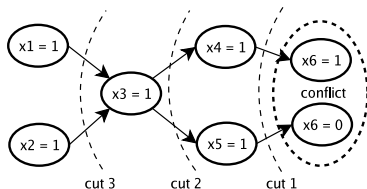
First, we solve the LP relaxation, which gives us an objective value of 3.7, and solution vector $x_1 = 0.5, x_2 = 1, x_3 = 0.5, x_4 = 0.5, x_5 = 0.5, x_6 = 0.5$. We branch on x_1 , and take the up branch ($x_1 = 1$). Constraint propagation finds no new assignments (besides the branch decision itself).



The LP relaxation results in an objective value of 3.65 and solution vector $x_1 = 1, x_2 = 0.5, x_3 = 0.5, x_4 = 0.5, x_5 = 0.5, x_6 = 0.5$. We branch on x_2 , and take the up branch ($x_2 = 1$). Performing constraint propagation on $x_2 = 1$ leads to the implied assignment $x_3 = 1$ (by the first constraint in the problem). Propagating $x_3 = 1$ leads to implied assignments $x_4 = 1$ and $x_5 = 1$ (by the second and third constraints, respectively). Finally, $x_4 = 1$ implies $x_6 = 1$ by the fourth constraint, and $x_5 = 1$ implies $x_6 = 0$ by the fifth constraint. We have thus detected a conflict on variable x_6 .



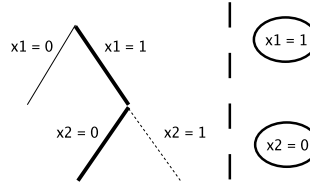
Now we find cuts in the graph that separate the conflict from the source nodes (which correspond to branching decisions). Not all cuts need be generated; in our example, say the algorithm generates three of them:



We translate the cuts into cutting planes for IP:

- Cut 1 in the graph generates nogood $\langle x_4 = 1, x_5 = 1 \rangle$, which yields the cutting plane $x_4 + x_5 \leq 1$.
- Cut 2 generates nogood $\langle x_3 = 1 \rangle$, which yields $x_3 \leq 0$.
- Cut 3 generates nogood $\langle x_1 = 1, x_2 = 1 \rangle$, which yields $x_1 + x_2 \leq 1$. However, this cutting plane is futile because it contains all of the branching decisions from the search path. Since search paths are distinct, this combination of variable assignments would never occur in any other part of the search tree anyway.

At this point the algorithm has proven that the current node is infeasible; there is no point in continuing down this search path. Therefore, the search moves on by popping another node from the open list. Say it pops the node corresponding to path $x_1 = 1, x_2 = 0$.⁶ Constraint propagation on $x_2 = 0$ yields no new assignments.



If we solve the LP for this node *without* the cutting planes we generated, the LP has an objective value of 3.1 and solution vector $x_1 = 1, x_2 = 0, x_3 = 0.5, x_4 = 0.5, x_5 = 0.5, x_6 = 0.5$. This would require further branching. However, solving the LP with the addition of our cutting planes, yields a tighter relaxation: an objective value of 3.0 and solution vector $x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1, x_5 = 0, x_6 = 1$. The solution is integral, so no further branching down that path is needed. Our cut generation process has thus produced a tighter LP bound that made the search tree smaller.

2.5 Generating additional conflicts and cutting planes from pruning by bound

In informed tree search, such as branch-and-cut, nodes can also be pruned by bounding. Denote by g the objective function contribution from the variables that have been decided by branching or propagation. Denote by h an upper bound on the rest of the problem—this is usually obtained by solving the LP involving the undecided variables and measuring their contribution to the objective. Finally, denote by f the current global lower bound (e.g., obtained from the incumbent). Then, if $g + h \leq f$,⁷ the current search node (and the subtree under it which has not yet been generated) is pruned.

Our nogood learning mechanism, as described so far, will only detect conflicts that stem from infeasibility. Further reduction in tree size can be achieved by also detecting conflicts that stem from bounding.

We address this by considering the current global lower bound as an additional constraint on the problem: given the objective function $\sum c_j x_j$ and the current global lower bound f , our algorithm considers the *objective bounding constraint* $\sum c_j x_j \geq f$ when performing constraint propagation. This simple technique will, in effect, treat as infeasible any assignment that cannot be extended into a solution that is better than f . This allows our cutting plane generation to occur in more

⁶E.g., depth-first branch-and-cut search would pick this node.

⁷If the lower bound comes from an actual incumbent solution, a strict inequality can be used.

nodes of the search tree and it also allows for cutting planes to be generated that could not have been generated with the vanilla version of our algorithm described above.⁸

2.6 Exploiting problem structure

We can achieve stronger constraint propagation (via tighter bounds on ϕ) using knowledge of the problem structure. This leads to more implications being drawn, more frequent or earlier conflict detection, and ultimately smaller search trees.

For example, consider the case where we have a set of variables J defined to be mutually exclusive in the problem: $\sum_{j \in J} x_j \leq 1$.⁹ Then, for *any* constraint i , we can redefine the helper function \bar{s}_i to get a lower upper bound U_{ij} on ϕ .

$$\bar{s}_i(S) = \sum_{j \in S \setminus J} |a_{ij}| u_j + \max_{j \in S \cap J} \{|a_{ij}| u_j\}$$

This tighter definition can easily be generalized to settings with multiple (potentially overlapping) sets J_1, \dots, J_r where at most one variable in each set can take value 1.

As another example, consider the case where we have a set of variables K defined such that at least one of them has to be “on”, i.e., $\sum_{j \in K} x_j \geq 1$.¹⁰ Then, for any constraint i ,¹¹

$$\underline{s}_i(S) = \begin{cases} \sum_{j \in S} |a_{ij}| l_j, & \text{if } K \not\subseteq S \\ \sum_{j \in S \setminus K} |a_{ij}| l_j + \min_{j \in K} \{|a_{ij}|\}, & \text{otherwise} \end{cases}$$

This tighter definition can easily be generalized to settings with multiple (potentially overlapping) sets K_1, \dots, K_r where at least one variable in each set has to take value 1.

Furthermore, if the problem exhibits the structures of both of the examples above, then both functions \bar{s}_i and \underline{s}_i in the revised forms above, should be used. An important special case of this is the case where exactly one of a set of variables has to be “on”, which corresponds to $J = K$.

Both of these examples can be viewed as special cases of using one knapsack constraint to deduce bounds on another, as presented in [Trick, 2003]. However, the actual method used to determine the bounds is completely different.

2.7 Backjumping

We can also generalize the backjumping idea from SAT to optimization. The main issue is that, unlike CSPs, optimization problems, such as integer programs, are typically not solved using depth-first search. Our solution works as follows, and does not rely on any particular search order. If, at any point of the search, we detect a nogood that contains no decisions or

⁸Cutting planes generated from the objective bounding constraint can cut off regions of the polytope that contain feasible integer solutions. No such points can be optimal, so branch-and-cut still produces correct results. However, the leading commercial MIP solvers (CPLEX and XPress-MP) assume that no cutting plane is used that cuts off feasible integer solutions. In such solvers, some functionality (parts of the preprocessor) needs to be turned off to accommodate these more aggressive cutting planes.

⁹This structure exists, for example, in the clique-based formulation of the independent set problem.

¹⁰This structure exists, for example, in the set covering problem.

¹¹This assumes that no variables in M have been fixed to 1 at the current search node. If a variable in M has been fixed to 1, then the knowledge that at least one member of K must be 1 is not useful.

implications made after the k th branching decision in the current path, we determine the ancestor, η , of the current node at depth k , and discard η and all its descendants. (This is valid since they are now known to be infeasible or suboptimal.)

There is an alternative way to accomplish this, which is easier to implement in the confines of the leading commercial MIP solvers. Consider the moment from the example above when η has been identified. Then, instead of explicitly discarding η and its descendants, simply mark η . Then, whenever a node comes up for expansion from the open list, immediately check whether that node or any node on the path from that node to the root has been marked; if so, the current node is infeasible or suboptimal and can be discarded (without solving its LP relaxation).

In the experiments, we do not use backjumping. This is because for any node on the open list that could be removed via backjumping reasoning, that node will be pruned if it is popped off the open list because the node’s LP will turn out infeasible (because it includes the cutting plane that would have allowed backjumping¹²). The cost of omitting backjumping is the need to potentially solve those nodes’ LPs. On the other hand, the overhead of backjumping (discussed above) is saved.

3 Experiments and analysis

We conducted experiments by integrating our techniques into ILOG CPLEX 9.1. In order to not confound the findings with undocumented CPLEX features, we turned off CPLEX’s pre-solve, cutting plane generation, and primal heuristics for all of the experiments. The platform was a 3.2 GHz Dual Core Pentium 4 based machine running 64-bit Fedora Core 3 Linux.

The first test problem was the combinatorial exchange winner determination problem [Sandholm *et al.*, 2002]. It can be formulated as a MIP, with a binary variable x_j for each bid, objective coefficients p_j corresponding to the prices of the bids, and quantity q_{ij} of each item i contained in bid j : $\max \sum_j p_j x_j$ such that $\forall i, \sum_j q_{ij} x_j = 0$. We generated instances randomly using the generator described in [Sandholm, 2003] (it uses graph structure to guide the generation; the prices and quantities can be positive or negative). We varied problem size from 50 items, 500 bids to 100 items, 1000 bids. For each size, we generated 150 instances.

The second problem was modeled after a class of combinatorial exchanges encountered in sourcing of truckload transportation services. There is a single buyer who wants to buy one unit of each item. The items are divided into regions. There is some number of suppliers; each supplier places singleton (i.e., non-combinatorial) bids on each of a subset of the items. A supplier can bid in multiple regions, but only bids on a randomly selected set of items in the region. Each bid is an ask in the exchange, so the price is negative. Finally, there are constraints limiting the number of suppliers that can win any business. There is one such constraint overall, and one for each region. The MIP formulation is as above, with the addition of a new binary variable for each supplier and for each supplier-region pair, the constraints over those variables, and constraints linking those binary variables to the bid vari-

¹²Even if cutting plane pool management is used to only include some of the cutting planes in the LP, the children of the node will be detected infeasible during constraint propagation.

ables. We varied problem size from 50 items, 500 bids to 100 items, 1000 bids. For each size, we generated 150 instances.

To test on a problem close to that on which nogood learning has been most successful, we experimented with a modified form of 3SAT. Each instance had 100 variables and 430 randomly generated clauses (yielding the hard ratio of 4.3). A subset of the variables (ranging from 10% to 100%) had positive objective coefficients; others' coefficients were 0.

Finally, we tested on all the MIPLIB [Bixby *et al.*, 1998] instances that only have 0-1 variables and for which an optimal solution is known.

Surprisingly, nogood learning does not seem to help in optimization: it provides little reduction in tree size (Table 1). Fortunately, the time overhead (measured by doing all the steps, but not inserting the generated cutting planes) averaged only 6.2%, 4.7%, 25.3%, and 12.8% on the four problems. We now analyze why nogood learning was ineffective.

	Tree reduction	Path pruned due to			Leaf yields nogood		Relevancy rate
		inf	bound	int	inf	bound	
Exchange	0.024%	0.001%	99.94%	0.060%	45.29%	0.000%	0.068%
Transport	0.005%	0.000%	100.0%	0.000%	42.80%	0.000%	0.047%
3SAT	2.730%	10.02%	89.98%	0.001%	75.83%	0.026%	5.371%
MIPLIB	0.017%	0.008%	99.99%	0.000%	37.66%	0.000%	0.947%

Table 1: Experiments. inf = infeasibility; int = integer solution from LP. Relevancy rate = of nodes that had at least one cutting plane, how many had at least one cutting plane binding at the LP optimum.

First, few nogoods are generated. Nodes are rarely pruned by infeasibility (Table 1), where our technique is strongest. In the 3SAT-based instances, pruning by infeasibility was more common, and accordingly our technique was more effective.

When nodes were pruned by bounding, our technique was rarely able to generate a nogood (Table 1). The objective bounding constraint requires so many variables to be fixed before implications can be drawn from it that in practice implications will rarely be drawn even when the LP prunes the node by bounding. This stems from the objective including most variables. The exception is those 3SAT-based instances where a small portion of the variables were in the objective. This explains why the implication graph was slightly more effective at determining a nogood from bounding in 3SAT, particularly the instances with few variables in the objective.

Second, the generated cutting planes are weak: they do not significantly constrain the LP polytope. The cutting planes effectively require at least one of a set of conditions (e.g., variable assignments) to be false. So, the larger the set of conditions, the weaker the cutting plane. On our problems, few inferences can be drawn until a large number of variables have been fixed. This, coupled with the fact that the problems have dense constraints (i.e., ones with large numbers of variables), leads to a high in-degree for nodes in the implication graph. This leads to a large number of edges in the conflict cut, and thus a large number of conditions in the nogood. Therefore, by the reasoning above, the cutting plane will be weak. This is reflected by the *relevancy rate* in Table 1. The cutting planes are least weak on the 3SAT-based instances; this is unsurprising since the 3SAT constraints are sparse compared to those in the other problems.

4 Generalizations

We now generalize our techniques to MIPs (that may include integer and real-valued variables in addition to binaries) and to branching rules beyond those that branch on individual variable assignments.

The key to the generalization is a generalized notion of a nogood. Instead of the nogood consisting of a set of variable assignments, we say that the *generalized nogood (GN)* consists of a set of conditions. The set of conditions in a GN cannot be satisfied in any solution that is feasible and better than the current global lower bound.

Nothing we have presented assumes that branches in the tree are binary; our techniques apply to multi-child branches.

4.1 Identifying generalized nogoods (GNs)

Even if the MIP includes integer or real variables, the techniques we presented for propagating binary variable assignments still work for propagating binary variable assignments.

The next case to handle is branching on artificial bounds on integer variables. For example, the search might branch on $x_j \leq 42$ versus $x_j \geq 43$. Such branching can be handled by changing our propagation algorithm to be the following. (For simplicity, we write the algorithm assuming that each variable has to be nonnegative. This is without loss of generality because any MIP can be changed into a MIP with nonnegative variables by shifting the polytope into the positive orthant.)

```

for all unsatisfied constraints  $i$  do
  for all unfixed variables  $\hat{j}$  do
    if  $\hat{j} \in N_i^+$  then
      if  $U_{i\hat{j}} < a_{i\hat{j}}l_{\hat{j}}$  then we have detected a conflict
      else if  $\left\lfloor \frac{U_{i\hat{j}}}{a_{i\hat{j}}} \right\rfloor < u_{\hat{j}}$  then  $u_{\hat{j}} \leftarrow \left\lfloor \frac{U_{i\hat{j}}}{a_{i\hat{j}}} \right\rfloor$ 
    else if  $\hat{j} \in N_i^-$  then
      if  $U_{i\hat{j}} < a_{i\hat{j}}u_{\hat{j}}$  then we have detected a conflict
      else if  $\left\lceil \frac{U_{i\hat{j}}}{a_{i\hat{j}}} \right\rceil > l_{\hat{j}}$  then  $l_{\hat{j}} \leftarrow \left\lceil \frac{U_{i\hat{j}}}{a_{i\hat{j}}} \right\rceil$ 

```

The procedure above works also for branches that do not concern only one variable, but an arbitrary hyperplane in the decision variable space. The hyperplane that is added in the branch decision is simply included into the constraint set of the child. The propagation is no different.

In our generalized method, not all nodes in the implication graph represent variable assignments; some represent half-spaces (a bound change or, more generally, a hyperplane).

If the branch decision states that moving from a parent to a child involves adding a *collection* of bound changes / hyperplanes (an important case of the former is *Special Ordered Set* branching [Beale and Tomlin, 1970]), then we can simply treat each of them separately using the procedure above.

4.2 Generating cutting planes from GNs

When we identify a GN from the implication graph, we are identifying a set of hyperplane constraints (with variable bounds and variable assignments as special cases) that cannot be mutually satisfied in any feasible solution better than the current lower bound. Thus the analog to conflict clauses in SAT is no longer direct as it was in 0-1 IP. Therefore, generating useful cutting planes is more challenging.

From the GN we know that at least one of the contained hyperplane constraints should be violated. Therefore, the feasible region of the LP can be reduced to be the intersection of the current feasible region of the LP and any linear relaxation of the disjunction of the infeasible IP region of the first constraint, the infeasible IP region of the second constraint, etc. (The tightest such relaxation is the convex hull of the disjunction.) The cutting planes that our method will generate are

facets of the linear relaxation. Not all of them need to be generated for the technique to be useful in reducing the size of the search tree through improved LP upper bounding. Any standard technique from *disjunctive programming* [Balas, 1979] can be used to generate such cutting planes from the descriptions of the infeasible IP regions.

To see that this is a generalization of our 0-1 IP technique, consider a 0-1 IP in which we have found a conflict. Say that the nogood generated from this conflict involves three variables being fixed to 0 (i.e., three hyperplane constraints): $\langle x_1 \leq 0, x_2 \leq 0, x_3 \leq 0 \rangle$. The infeasible IP regions of the three constraints are $x_1 \geq 1$, $x_2 \geq 1$, and $x_3 \geq 1$, respectively. Their disjunction is $\{x : x_1 \geq 1 \vee x_2 \geq 1 \vee x_3 \geq 1\}$, which has LP relaxation $x_1 + x_2 + x_3 \geq 1$. This is the cutting plane that our generalized method would generate. This is also the same cutting plane that our original method for 0-1 IPs would generate from the nogood $\langle x_1 = 0, x_2 = 0, x_3 = 0 \rangle$.

5 Conclusions and future research directions

Nogood learning is an effective CSP technique critical to success in leading SAT solvers. We extended the technique for use in combinatorial optimization—in particular, mixed integer programming (MIP). Our technique generates globally valid cutting planes for a MIP from nogoods learned through constraint propagation. Nogoods are generated from infeasibility and from bounding. Experimentally, our technique did not speed up CPLEX. This is due to few cutting planes being generated and the cutting planes being weak. We explained why. Potentially fruitful directions remain, however.

First, at any node that is pruned during search, one could use the LP to generate an *irreducibly inconsistent set* (IIS). The IIS is a nogood, so one can generate a cutting plane from it. For 0-1 IP, [Davey *et al.*, 2002] produce an IIS with the smallest overlap with variable assignments on the search path, and use that as the cutting plane. (Such techniques do not subsume ours: no IIS-based cutting plane would help in the example of Section 2.4 because any IIS would include all the decisions on the path.) For MIPs, we can use the same LP technique to generate a GN that shares the smallest number of variable bound changes with the path (thus it will help prune at many places of the tree that are not on the current path). Then we can use the techniques from the generalizations section to generate cutting planes from that GN.

Second, our machinery for implication graph construction can also be used for dynamic tightening of variable bounds as the search proceeds. Specifically, the inferences we draw during constraint propagation can be explicitly applied to the LP for the current node and its descendants. These bound tightenings are beyond those implied by the LP relaxation. Thus the technique yields tighter LP bounds, and smaller search trees. [Harvey and Schimpf, 2002] proposed similar techniques for bound consistency, without experiments.

Third, one could apply *lifting* (i.e., including additional variables, with the largest provably valid coefficients, into the constraint) to the cutting planes we are generating in order to strengthen them (i.e., cut off more of the LP polytope). Our cutting planes are similar to *knapsack cover inequalities* [Nemhauser and Wolsey, 1999], for which lifting is relatively straightforward. Experiments should be conducted to determine whether nogood learning enhanced by lifting would improve speed.

Fourth, there may be real-world problems where the techniques, even as-is, yield a drastic speedup; one can construct instances where they reduce tree size by an arbitrary amount. The experiments suggest more promise when nodes are often pruned by infeasibility (rather than bounding or integrality), when the objective is sparse, or when constraints are sparse.

References

- [Balas, 1979] Egon Balas. Disjunctive programming. *Annals of Discrete Mathematics*, 5:3–51.
- [Beale and Tomlin, 1970] E Beale and J Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. *Intl. Conf. on OR*.
- [Bixby *et al.*, 1998] Robert Bixby, Sebastian Ceria, Cassandra McZeal, and Martin Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 54:12–15.
- [Bockmayr and Eisenbrand, 2000] A Bockmayr and F Eisenbrand. Combining logic and optimization in cutting plane theory. *Workshop on Frontiers of Combining Systems, LNCS 1794*.
- [Bockmayr and Kasper, 1998] A Bockmayr, T Kasper. Branch and infer: A unifying framework for integer and finite domain constraint programming. *INFORMS J on Computing*, 10:287–300.
- [Chai and Kuehlmann, 2003] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. *Design Automation Conference*.
- [Davey *et al.*, 2002] Bruce Davey, Natasha Boland, and Peter Stuckey. Efficient intelligent backtracking using linear programming. *INFORMS J on Computing*, 14:373–386.
- [Dechter, 1990] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312.
- [Focacci *et al.*, 1999] F Focacci, A Lodi, and M Milano. Cost-based domain filtering. *CP*.
- [Frost and Dechter, 1994] Daniel Frost and Rina Dechter. Dead-end driven learning. *AAAI*.
- [Harvey and Schimpf, 2002] Warwick Harvey and Joachim Schimpf. Bounds consistency techniques for long linear constraints. *CP workshop on Techniques for Implementing Constraint Programming Systems*.
- [Hooker *et al.*, 1999] John Hooker, Greger Ottosson, Erlendur Thorsteinsson, H-J Kim. On integrating constraint propagation and linear programming for combinatorial optimization. *AAAI*.
- [Hooker, 2002] John Hooker. Logic, optimization and constraint programming. *INFORMS J of Computing*, 14:295–321.
- [Marques-Silva and Sakallah, 1999] J Marques-Silva and K Sakallah. GRASP-A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521.
- [Moskewicz *et al.*, 2001] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. *Design Automation Conference*.
- [Nemhauser and Wolsey, 1999] George Nemhauser and Laurence Wolsey. *Integer and Combinatorial Optimization*.
- [Padberg and Rinaldi, 1987] Manfred Padberg and Giovanni Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6:1–7.
- [Padberg and Rinaldi, 1991] Manfred Padberg, Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100.
- [Richards and Richards, 1996] E Thomas Richards and Barry Richards. Nogood learning for constraint satisfaction. *CP*.
- [Sandholm *et al.*, 2002] Tuomas Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. Winner determination in combinatorial auction generalizations. *AAMAS*.
- [Sandholm, 2003] Tuomas Sandholm. Winner determination in combinatorial exchanges. Slides from the FCC Combinatorial Bidding Conference. <http://wireless.fcc.gov/auctions/conferences/combin2003/presentations/Sandholm.ppt>.
- [Trick, 2003] Micheal Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118:73–84.