Nogood Learning for Mixed Integer Programming

Tuomas Sandholm **Rob Shields**

November 2006
CMU-CS-06-155

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

The bulk of this work was funded by, and conducted at, CombineNet, Inc., Fifteen 27th St., Pittsburgh, PA 15222. The work to finish this was funded by the National Science Foundation under ITR grants IIS-0121678 and IIS-0427858.

Keywords: Search, integer programming, mixed integer programming, combinatorial optimization, conflict analysis, infeasibility analysis, nogoods, nogood learning, clause learning, cutting planes, branch-and-bound, branch-and-cut, combinatorial auctions.

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 integer programming and mixed integer programming. Our technique generates globally valid cutting planes for the 0-1 IP search algorithm from information learned through constraint propagation (bounds 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 our nogood learning does not help in integer programming because few cutting planes are generated, and they are weak. We explain why, and identify problem characteristics that affect the effectiveness. We show how problem structure, such as mutual exclusivity of indicator variables, or at least one of a set of indicator variables having to be "on", can be used to enhance the technique. We show this also for instances that exhibit multiple occurrences of each of the two structures. We then generalize the technique to mixed-integer programming. Then we compare our techniques to Achterberg's parallel invention of an almost identical approach. This comparison yields conclusions about what techniques within the nogood learning framework for (mixed) integer programming are essential for obtaining speedup. Finally, we lay out several directions for future research down this new and potentially promising avenue.

1 Introduction

Nogood learning is a powerful technique for reducing search tree size in constraint satisfaction problems (CSPs) (e.g., [11, 14, 28, 8]). 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., [21, 24]).

We present a propagation-based nogood learning method for mixed integer programming (MIP). 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, which is domain independent and has a very broad range of applications in scheduling, routing, facility location, combinatorial auctions, etc. We designed the idea in June 2003, and since then have built an implementation of it on top of ILOG CPLEX. The same idea has been developed independently and in parallel by Achterberg, with an implementation on top of his MIP solver, SCIP [1]. (We will discuss the similarities and differences between our work and his in the related research section.) 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 also been proposed (e.g., [7, 17, 13, 6, 16]).

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 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 [26, 27], 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* [25]. 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 rest of this paper is organized as follows. Section 2 presents our approach in the context of 0-1 IPs. Section 3 covers experiments and explains the performance. Section 4 shows how special problem structures can be exploited to enhance the technique. Section 5 generalizes our approach from 0-1 IPs to MIP. Section 6 discusses related research. Section 7 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

[1]Propagation of linear constraints has been explored previously, in the context of bounds consistency [15].

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

$$\sum_{j \in N} a_{ij} x_j \leq b_i \tag{1}$$

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

$$a_{i\hat{j}} x_{\hat{j}} \leq b_i - \sum_{j \in N \setminus \hat{j}} a_{ij} x_j \tag{2}$$

This is the same form as the inequality examined above. Now,

$$\begin{aligned} \phi_{i\hat{j}}(x) &= b_i - \sum_{j \in N, j \neq \hat{j}} a_{ij} x_j & (3) \\ &= b_i - \sum_{j \in N_i^+, j \neq \hat{j}} |a_{ij}| x_j + \sum_{j \in N_i^-, j \neq \hat{j}} |a_{ij}| x_j & (4) \end{aligned}$$

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

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

$$U_{i\hat{j}} = b_i - \underline{s}_i(\{j | j \in N_i^+, j \neq \hat{j}\}) + \overline{s}_i(\{j | j \in N_i^-, j \neq \hat{j}\}) \tag{5}$$

yields an upper bound as long as

$$\underline{s}_i(S) \leq \sum_{j \in S} |a_{ij}| x_j \leq \overline{s}_i(S) \tag{6}$$

for all x.

With no other knowledge of the problem structure, we can use

$$\underline{s}_i(S) = \sum_{j \in S} |a_{ij}| l_j \tag{7}$$

and

$$\overline{s}_i(S) = \sum_{j \in S} |a_{ij}| u_j \tag{8}$$

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]

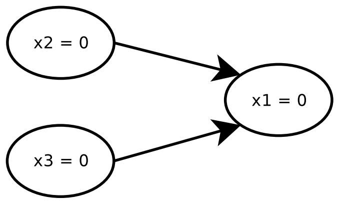
[2]This is very similar to that used for nogood learning in CSPs. It can be sped up by watching the set of variables that are candidates to become implied shortly [8].

```
for all unsatisified[3] constraints $i$ do
  for all unfixed[4] 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 maintains[5] 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

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

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

[5]This 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).

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., [21, 24]) and can be applied in our setting directly. (In the experiments, we use the 1UIP 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 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:

$$\sum_{j \in N_0} x_j \geq 1 \tag{9}$$

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:

$$\sum_{j \in N_1} (1 - x_j) \geq 1 \tag{10}$$

Putting these together, a nogood generates the cutting plane

$$\sum_{j \in V_0} x_j - \sum_{j \in V_1} x_j \leq 1 - |V_1| \tag{11}$$

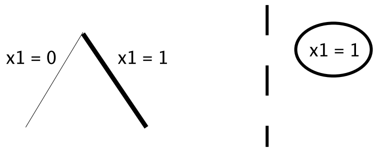
2.4 A small example

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

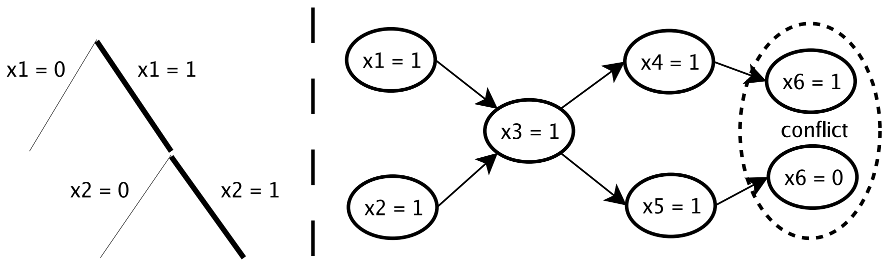
$$\begin{array}{lccccccccc}
\max & x_1 & +1.1x_2 & +1.2x_3 & +x_4 & +x_5 & +x_6 & & \\
s.t. & -x_1 & -x_2 & +x_3 & & & & \geq & -1 \\
 & & & -x_3 & +x_4 & & & \geq & 0 \\
 & & & -x_3 & & +x_5 & & \geq & 0 \\
 & & & & -x_4 & & +x_6 & \geq & 0 \\
 & & & & & -x_5 & -x_6 & \geq & -1 \\
 & & & & & & x_j &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

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).

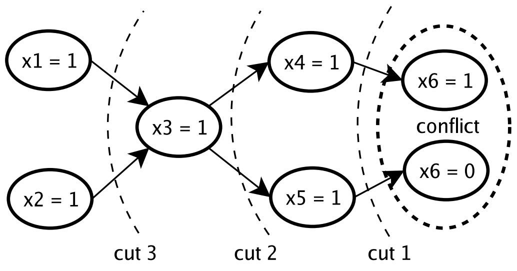
The LP relaxation results in an objective value of 3.65 and solution vector $x_1 = 1, x_2 = 0.5, x_3 = 0.5, x_4 = 0.5, x_5 = 0.5, x_6 = 0.5$. We branch on x_2, and take the up branch ($x_2 = 1$). Performing constraint propagation on $x_2 = 1$ leads to the implied assignment $x_3 = 1$ (by the first



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.



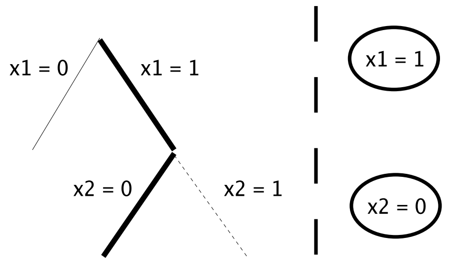
Now we find cuts in the graph that separate the conflict from the source nodes (which correspond to branching decisions). Not all cuts need be generated; in our example, say the algorithm generates three of them:



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*

$$\sum c_j x_j \geq f \tag{12}$$

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 nodes of the search tree and it also allows for cutting planes to be

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

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

generated that could not have been generated with the vanilla version of our algorithm described above.[8]

2.6 Generating additional conflicts and cutting planes from LP infeasibility

We implemented all the techniques presented in the paper so far. With these techniques, conflicts and cutting planes are generated based on propagation of variable bounds.

Another form of propagator that is present in MIP solvers is the linear program (LP) solver. Sometimes the LP returns infeasibility even at nodes where the bounds propagator does not catch the infeasibility. Therefore, one could generate additional conflicts and cutting planes from LP infeasibility.

For example, at any node that is pruned during search due to LP infeasibility or objective constraint violation, one could use the LP to generate an *irreducibly inconsistent set (IIS)*, i.e., a set of constraints that make the LP infeasible such that if any constraint is removed from the set, the conflict would disappear. Any IIS is a nogood, so one can generate a cutting plane from it. For 0-1 IP, Davey et al. produce an IIS with the smallest overlap with variable assignments on the search path, and use that as the cutting plane [9].

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.

We did not implement any techniques for generating conflicts and cutting planes from LP infeasibility because we wanted to keep the run-time per search node very small. However, that is a very promising direction for future research. We will come back to this in the “Related research” section and in the “Conclusions and future research” section.

2.7 Backjumping

We can also generalize the backjumping idea from SAT to mixed integer programming. 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 implications made after the kth 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

[8]Cutting 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.

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. Any node on the open list that could be removed via backjumping reasoning will be pruned anyway once it is popped off of the open list: the node's LP will be found infeasible. This is guaranteed by the presence of the cutting plane(s) which would have allowed backjumping in the first place.[9] 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. CPLEX's default node selection strategy was used in all of the experiments. In order to not confound the findings with undocumented CPLEX features, we turned off CPLEX's presolve, cutting plane generation, and primal heuristics. 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 [31]. 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 [30] (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 variables. 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 ran experiments on 3SAT instances converted to MIP. Each instance had 100 variables and 430 randomly generated clauses (yielding the hard ratio of 4.3). We converted each clause (i.e., constraint) of the 3SAT instance to a MIP constraint in the natural straightforward way. For example, clause $(x_1 \vee \neg x_7 \vee \neg x_9)$ is converted into the constraint $x_1 + (1 - x_7) + (1 - x_9) \geq 1$. In order to explore both the optimization power and the constraint satisfaction power of our approach, we made some

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

of the instances optimization instances rather than pure constraint satisfaction instances. Specifically, a subset of the variables (ranging from 0% to 100% in increments of 10%, with 100 instances at each setting) had positive objective coefficients; other variables' coefficients were 0.

Finally, we tested on all the MIPLIB 3.0 [5] instances that only have 0-1 variables and for which an optimal solution is known. There are 20 such instances.

Surprisingly, nogood learning does not seem to help in MIP: 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	Path pruned due to			Leaf yields nogood		Relevancy
	reduction	infeasibility	bound	integrality	infeasibility	bound	rate
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. Relevancy rate = of nodes that had at least one of our cutting planes, how many had at least one of them 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, column 7). 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 (variable assignments in the case of 0-1 IP) 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.

In order to further explore the behavior of our techniques, we ran additional experiments on SAT-based instances. Using the instance generator discussed above, we varied the *clause density* (ratio of clauses to variables), *objective function density* (fraction of variables that had a nonzero objective function coefficient; the nonzero coefficients were uniformly randomly drawn from $(0, 1]$), and *clause length* (number of literals per clause). 1000 instances were generated at each data point. The results are shown in Figure 1.

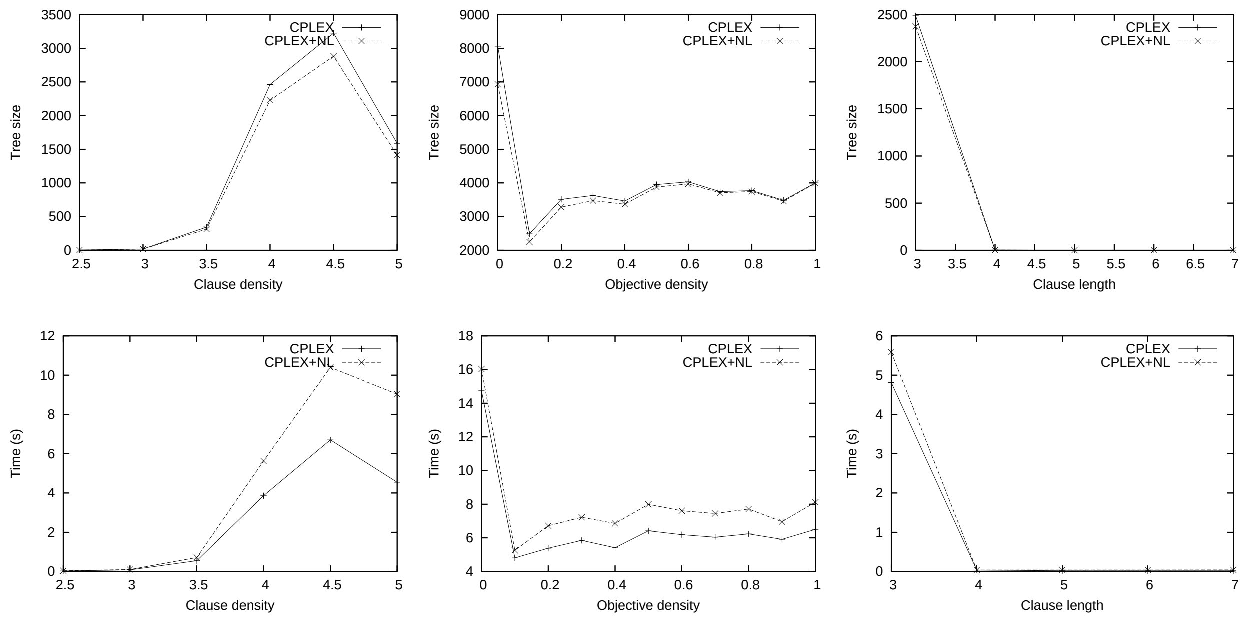


Figure 1: Average performance on SAT-based instances, with and without nogood learning (NL). Top: tree size; bottom: run time. Left: clause length 3, objective density 10%. Middle: clause length 3, clause density 4.2. Right: Clause density 4.2, objective density 10%.

We expected our technique, as described above, to perform better on instances which are hard from a satisfiability perspective (i.e., a large number of nodes are pruned due to infeasibility) and to perform worse on instances which are hard from an optimization perspective (i.e., a large number of nodes are pruned due to bounding). This is exactly what we see here:

- As clause density increases, the problem becomes over-constrained. This leads to a higher percentage of nodes being pruned due to infeasibility, which in turn allows us to generate more cutting planes. This leads to a greater relative reduction in tree size. (As usual, the overall complexity—with and without nogood learning—peaks at a clause density between four and five.)

- As objective density increases, more variable assignments actually impact the objective value. This leads to a higher percentage of nodes being pruned due to bounding; a situation in which our current technique was expected to have poor relative performance, and that was indeed observed.

  Overall, the problem is much harder for integer programming (with and without nogood learning) when there is no objective. In other words, the objective helps even in finding a feasible solution. Then, as objective density increases, the complexity increases slightly.

- As clause length increases (holding clause density constant), the problems become less constrained. This leads to fewer nodes being pruned by infeasibility, and thus our technique, as described, became relatively less effective. On the other hand, the instances overall became trivial.

4 Exploiting problem structure

We can achieve stronger constraint propagation (via tighter bounds on ϕ than those obtained from Equation 4) using knowledge of the problem structure. This leads to more implications being drawn, more frequent or earlier conflict detection, and ultimately smaller search trees.

4.1 Mutual exclusivity structure

For example, consider the often occurring case where we have a set of variables J defined to be mutually exclusive in the problem:[10]

$$\sum_{j \in J} x_j \leq 1 \tag{13}$$

Then, for *any* constraint i, we can redefine the helper function $\overline{s}_i$ of Equation 7 to get a lower upper bound U_{ij} on ϕ as follows:

$$\overline{s}_i(S) = \sum_{j \in S \setminus J} |a_{ij}| u_j + \max_{j \in S \cap J} \{|a_{ij}| u_j\} \tag{14}$$

This tighter definition can easily be generalized to settings with multiple (potentially overlapping) sets $J_1, \ldots, J_r$ where at most one variable in each set can take value 1. An extremely simple method of doing this would be the following:

$$\overline{s}_i(S) = \sum_{j \in S \setminus \bigcup J} |a_{ij}| u_j + \sum_{k=1}^{r} \max_{j \in S \cap J_k} \{|a_{ij}| u_j\} \tag{15}$$

We can improve this further with a simple greedy algorithm for constructing a set J_i^*, which will then be used in the calculation of $\overline{s}_i(S)$.

```
$J_i^* = \emptyset$
**for all** $J_k$ **do**
    $J_i^* = J_i^* \cup \max_{j \in J_k \setminus J_i^*} \{|a_{ij}| u_j\}$
```

This effectively builds up J_i^* by taking the largest element from J_1, then the largest element of J_2 that has not yet been taken, then the largest element of J_3 that has not yet been taken, and so on. The new expression for $\overline{s}_i(S)$ is now

$$\overline{s}_i(S) = \sum_{j \in S \setminus \bigcup J} |a_{ij}| u_j + \sum_{j \in J_i^*} |a_{ij}| u_j \tag{16}$$

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

The sets $J_1, \ldots, J_r$ are determined statically as a preprocessing step. Then at each node of the search tree as we inspect each constraint i in turn for possible implications, we dynamically run the greedy algorithm above for constructing J_i^* and use the tighter calculation of $\overline{s}_i(S)$ above. This is the approach we use in the experiments below.

The tightness of the bound could be further improved by solving a weighted vertex-packing problem as each constraint i is inspected for possible implications at each search node. The vertices in that problem correspond to the variables in the set S. An edge exists between two vertices if the corresponding variables both occur in some set J_k. A maximum-weight vertex-packing gives us a better J_i^*. This is in fact the best possible bound $\overline{s}_i(S)$ that can be derived solely from this one constraint and the mutual exclusion knowledge. We do not use this approach because the greedy algorithm for constructing J_i^* is faster.

4.2 Structure where at least one indicator has to be "on"

As another example of a commonly-occurring special structure, consider the case where we have a set of variables K defined such that at least one of them has to be "on":[11]

$$\sum_{j \in K} x_j \geq 1 \tag{17}$$

Then, for any constraint i, redefine the helper function $\underline{s}_i$ of Equation 8 to get a lower upper bound U_{ij} on ϕ as follows:[12]

$$\underline{s}_i(S) = \begin{cases} \sum_{j \in S} |a_{ij}| l_j, & \text{if } K \not\subseteq S \\ \sum_{j \in S \setminus K} |a_{ij}| l_j + \min_{j \in K} \{|a_{ij}|\}, & \text{otherwise} \end{cases} \tag{18}$$

This tighter definition can easily be generalized to settings with multiple (potentially overlapping) sets $K_1, \ldots, K_r$ where at least one variable in each set has to take value 1. As in the previous example, we start with a simple method:

$$\underline{s}_i(S) = \begin{cases} \sum_{j \in S} |a_{ij}| l_j, & \text{if } K \not\subseteq S \\ \sum_{j \in S \setminus K} |a_{ij}| l_j + \min_{k \in \{1,\ldots,r\}} \{\min_{j \in K_k} \{|a_{ij}|\}\}, & \text{otherwise} \end{cases} \tag{19}$$

Again, we use a more sophisticated version of the approach. We build a new set K_i^* dynamically as follows. We create a graph with a vertex for each variable. An edge between two vertices exists if the corresponding variables both appear in any one set K_k. We run depth-first search in this

[11] This structure exists, for example, in the set covering problem.

[12] This 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.

graph to identify all maximal connected components in linear time. From each such component we include the minimum-weight vertex in the set K_i^*. We now have

$$\underline{s}_i(S) = \begin{cases} \sum_{j \in S} |a_{ij}| l_j, & \text{if } K \not\subseteq S \\ \sum_{j \in S \setminus K} |a_{ij}| l_j + \sum_{j \in K_i^*} \{|a_{ij}|\}, & \text{otherwise} \end{cases} \tag{20}$$

The sets $K_1, \ldots, K_r$ are determined statically as a preprocessing step. Then at each node of the search tree as we inspect each constraint i in turn for possible implications, we dynamically compute K_i^* and use the tighter calculation of $\underline{s}_i(S)$ above. This is the approach we use in the experiments below.

The tightness of the bound could be further improved by solving a weighted set-covering problem at each step; each of the sets $K_1, \ldots, K_r$ corresponds to a set in the problem, each variable corresponds to a vertex, and the weights of the variables are the coefficients a_{ij}. This is the tightest bound that can be obtained by using only this one constraint i and the "at least one variable on in each set K_k"-knowledge. We do not use this approach because it takes longer than the one above.

4.3 Co-existing structures

If the problem exhibits the structures of both of the examples above, then both functions $\overline{s}_i$ and $\underline{s}_i$, in the revised forms above, should be used.[13] An important special case of this is the case where exactly one of a set of variables has to be "on", which corresponds to $J = K$.

We conducted an experiment to determine the effectiveness of these techniques that exploit problem structure. This experiment used exactly the same data set as the first experiment. (Better results would naturally we achieved on problems that exhibit these special structures more predominantly.) The results are presented in Table 2.

	Tree	Path pruned due to			Leaf yields nogood		Relevancy
	reduction	infeasibility	bound	integrality	infeasibility	bound	rate
Exchange	0.024%	0.001%	99.94%	0.060%	45.29%	0.000%	0.068%
Transport	0.016%	0.000%	100.0%	0.000%	42.80%	0.056%	0.054%
3SAT	2.732%	10.02%	89.98%	0.001%	75.83%	0.045%	5.372%
MIPLIB	0.018%	0.008%	99.99%	0.000%	37.66%	0.001%	0.947%

Table 2: Experiments exploiting problem structure.

While these experiments show an improvement over the base algorithm, it is very modest. Thus the overall result is still disappointing.

The Exchange and MIPLIB instance sets show the least amount of change. The special structures we are looking for occur very rarely in these instances. 9.73% of those instances had at least

[13]Each of the two example structures above can be viewed as special cases of using one knapsack constraint to deduce bounds on another, as presented in [33]. However, the actual method used to determine the bounds is completely different.

one occurrence of these special structures. Furthermore, on only 10.26% of these instances did a special structure actually lead to an otherwise undetectable implication.

The transportation instances show the most improvement. Comparing Tables 1 and 2, one can see that the exploitation of special structures in the algorithm causes some cutting planes to be generated even at nodes pruned by bounding.

The SAT-based instances also show some improvement, but it is very slight. This can be explained by noting that only the second type of exploitable structure discussed above occurs in these instances. This type of structure requires that the entirety of a set K_k be contained in a constraint set S; in the SAT-based problems this cannot occur with any other constraint except the objective bounding constraint. Accordingly, the higher the objective function density of the instance, the greater was the gain from exploiting the special structure.

5 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.

5.1 Identifying generalized nogoods (GNs)

The techniques presented so far in this paper for propagating and inferring binary variable assignments are still valid in the presence of integer or real variables. However, propagating and inferring integer or real variable assignments and bound changes requires additional consideration, as the rest of this subsection will discuss.

We first show how to handle branching on 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 [4]), then we can simply treat each of them separately using the procedure above.

5.2 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* [3] 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 constrains 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 the also the same cutting plane that our original method for 0-1 IPs would generate from the nogood $\langle x_1 = 0, x_2 = 0, x_3 = 0\rangle$.

6 Related research

No-good recording was first mentioned in the mid-1970s by Stallman and Sussman [32]. Their idea gave rise to algorithms in 1990 that include backjumping and nogood recording [23, 11]. The early work on nogood learning focused on constraint satisfaction problems (CSPs). It tended to not improve speed much in the propagator-heavy state-of-the-art CSP solvers, and was by and large not incorporated into the commercial CSP solvers. There was also some early exploration applying this technique to constraint optimization problems [22]. Nogood learning has been incorporated in some current constraint optimization solvers, such as PaLM [18].

In the mid-1990s, nogood learning was extremely successfully applied to SAT [21, 24]. It is now a part of all the competitive complete SAT solvers. We recently learned that in 2005, success with nogood learning has been reported on CSPs more generally by allowing a somewhat richer notion of nogood, where the nogood not only contains variable assignments but also nonassignments (excluded values of a variable) [19]. That notion of a generalized nogood is a special case of ours, presented in Section 5.

Since 2003, nogood learning has also been applied to restricted forms of optimization. Chai and Kuehlmann studied this in CSPs with pseudoboolean constraints (i.e., linear constraints over binary variables, but potentially with coefficient other than 1, 0, or -1) [8]. Their system was able to outperform IBM's OSLv3 (a slower MIP solver, similar in function to CPLEX) on constraint satisfaction problems, but performed far worse than OSL on optimization problems. It is not clear whether the performance difference on optimization is due to the generated cuts being weak or due to the manner in which their system performs optimization (a linear search over values of the objective, and running constraint satisfaction at each such value). Additionally, they found that OSL with an artificial objective function actually outperformed OSL without an objective on the same instances. Nogood learning has recently also been applied to problems where one is trying to maximize the number of constraints satisfied [34, 29, 2], with great success. A similar technique has been described for use in disjunctive programming for planning in an autonomous space probe [20], with similar results.

The similarity between the concept of a noogod for SAT and a cutting plane for MIP is mentioned in [12]. However, the discussion is at a high level and no method for actually generating such cutting planes is given.

The most closely related work to ours is that of Achterberg [1]. Independently and in parallel with our work, he came up with the same idea of applying nogood learning to MIP. In the rest of this section, we discuss similarities and differences between our approach and his.

The method presented by Achterberg for construction and maintenance of the implication graph is nearly identical to ours. Both his approach and ours draw nogoods from bounds propagation. His approach differs from ours in that our implementation does not draw nogoods from LP infeasibility analysis while his does. We proposed using an IIS-based method (such as Davey's [9]) for drawing nogoods from LP infeasibility, but we did not implement that because we thought it would increase solve time at each search node too much to pay off overall. Achterberg presents a novel relatively fast way of combining LP infeasibility analysis with the implication graph obtained through bounds propagation. If an LP infeasibility is reached (or the objective constraint is violated), he takes the path decisions and greedily tries to remove them one at a time while maintaining the property that the remaining decisions are in conflict. Then he inserts a conflict node into the implication graph and connects it to those remaining decisions.

Another difference is that Achterberg's work does not take advantage of special structure, such as mutual exclusivity of indicators, or the structure where at least one of a set of indicators has to be "on".

Experimentally, Achterberg's results differ from ours regarding the effectiveness of nogood learning as applied to 0-1 IP. He runs on two data sets: a selection from MIPLIB and a selection of instances from a chip design verification problem that has a convenient IP formulation. All instances are run with CPLEX 9, SCIP (his own MIP solver), and SCIP+conflict analysis. In the experiments, it appears that SCIP tends to have longer solve times than CPLEX, but much smaller search trees. His experiments show significant reduction in tree size from SCIP to SCIP+CA. However, on MIPLIB instances the smaller trees did not make up for the overhead of generating the cutting planes. On the other hand, on the chip verification instances, SCIP+CA clearly outperformed SCIP in both tree size and run time. (The use of a chip verification modeling problem is perhaps questionable for measuring the effectiveness of an optimization technique. It is a pure constraint satisfaction problem; integer programming might thus not be a good choice in the first place. However, as would be expected, Achterberg's nogood learning technique performed the best on exactly those instances.) There are several hypotheses for explaining the difference in performance between our approach and his:

- His approach draws nogoods also from LP infeasibility while our implementation does not. The significantly better reduction in tree size suggests that the nogoods drawn from LP infeasibility play a very important role.
- His implementation features different cut management. For example, we always generate at most one cutting plane per search node.
- He shows a tree size reduction when compared to SCIP. We show that there is only a tiny tree size reduction when compared to CPLEX. These may not be comparable. Similarly, in terms of run time, CPLEX and SCIP may not be comparable benchmarks.
- Different instance sets were used in the experiments.

Achterberg's generalization from 0-1 IP to MIP is somewhat different from ours. For nogoods containing bound changes of integer variables, he presents a linearization of the resulting constraints that, while correct, requires the addition of auxiliary variables. He points out that the resulting LP relaxation is weak, and considers the approach not worthwhile. Instead, he proposes restricting the algorithm to nogoods that contain only binary variables. His linearization also does not handle nogoods containing conditions other than bound changes. Our generalized nogood concept allows for arbitrary conditions and integer variables, and we propose disjunctive programming as a direction for finding linearization. While our generalized nogood concept in principle allows for continuous variables as well, Achterberg shows that if the generalized nogood (he does not actually use that concept, but we phrase his work using this concept) contains a bound change on a continuous variable, then expressing that nogood using cutting planes would require strict inequalities. Strict inequalities cannot be expressed directly in LP-based MIP, so those cases have to go unaddressed or be approximated. A potentially interesting direction for future research would be to apply (or extend) some of the special techniques for handling strict inequalities in MIP (e.g., [10]) to this context.

7 Conclusions and future research

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 (although tree size was slightly reduced). This is due to few cutting planes being generated and the cutting planes being weak. We explained why.

We also conducted experiments while controlling the ratio of constraints to variables, the number of variables per constraint, and the density of the objective. Hardness peaks at an interior value of the constraints to variables ratio, and the relative tree size reduction from our nogood learning technique improves as that ratio increases. Instances with no objective were hardest and instances with about 10% of the variables in the objective were easiest. As objective density increased further, the instances became slightly harder. The relative performance of our nogood learning technique was best when the objective was nonexistent or sparse. As clause length increased, the instances became trivial and clause learning ceased to decrease tree size.

We showed how our technique can be enhanced by exploiting special structure such as mutual exclusivity of indicator variables or at least one of a set of indicator variables having to be "on". We also showed how to capitalize on instances where multiple occurrences of each of these two structures are present.

There are a host of potentially fruitful directions for future research.

First, the difference in performance between our implementation and Achterberg's suggests that it is important to draw nogoods not only from propagation but also from LP infeasibility. As we presented (but did not yet implement), at any node that is pruned during search due to LP infeasibility or objective constraint violation, 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. produce an IIS with the smallest overlap with variable assignments on the search path, and use that as the cutting plane [9]. (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.) Further research on IIS-based techniques, on Achterberg's novel technique, and additional techniques for generating cutting planes from LP infeasibility analysis would likely pay off.

For MIPs, we can use the same IIS-based 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 of this paper to generate cutting planes from that GN. Future research includes studying disjunctive programming techniques in detail for generating linear constraints that correspond to the GN. Furthermore, for the case of continuous variables in the GN, it would be potentially interesting to apply (or extend) some of the special techniques for handling strict inequalities in MIP (e.g., [10]).

Second, one should explore additional ways of generating nogoods from the implication graph, as well as generating more than one nogood per search node. In our current implementation, we only generated one particular nogood (1-UIP).

Third, 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 proposed similar techniques for bound consistency, without experiments [15].

Fourth, 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* [25], for which lifting is relatively straightforward. Experiments should be conducted to determine whether nogood learning enhanced by lifting would improve speed.

Fifth, 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, when there are lots of constraints compared to the number of variables, and when the constraints contain few variables each.

References

[1] Tobias Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 2006. Special issue on Integer and Mixed-Integer Programming, to appear.

[2] J. Argelich and F. Many. Learning hard constraints in Max-SAT. In *In 11th Annual ERCIM Workshop on Constrant Solving and Constraint Programming (CSCLP-2006), Caparica, Portugal*, pages 5–12, 2006.

[3] Egon Balas. Disjunctive programming. *Annals of Discrete Mathematics*, 5:3–51, 1979.

[4] Evelyn Martin Lansdowne Beale and John A. Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In J. Lawrence, editor, *Proceedings of the Fifth International Conference on Operational Research*, pages 447–454. Tavistock Publications, London, 1970.

[5] Robert Bixby, Sebastian Ceria, Cassandra McZeal, and Martin Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 54:12–15, 1998.

[6] A. Bockmayr and F. Eisenbrand. Combining logic and optimization in cutting plane theory. In *3rd International Workshop on Frontiers of Combining Systems*, volume 1794 of *LNCS*, pages 1–17. Springer, March 2000.

[7] A. Bockmayr and T. Kasper. Branch and infer: A unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10(3):287–300, 1998.

[8] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 24(3):305–317, 2005.

[9] Bruce Davey, Natashia Boland, and Peter J. Stuckey. Efficient intelligent backtracking using linear programming. *INFORMS Journal on Computing*, 14(3):373–386, 2002.

[10] Ismail de Farias, M Zhao, and H Zhao. A special ordered set approach to discontinuous piecewise linear optimization, 2005. Draft.

[11] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[12] Heidi E. Dixon and Matthew L. Ginsberg. Combining satisfiability techniques from AI and OR. *The Knowledge Engineering Review*, 15:31–45, 2002.

[13] F Focacci, A Lodi, and M Milano. Cost-based domain filtering. In *International Conference on Principles and Practice of Constraint Programming (CP)*, 1999. Springer LNCS 1713, 189-203.

[14] Daniel Frost and Rina Dechter. Dead-end driven learning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 294–300, Seattle, WA, 1994.

[15] Warwick Harvey and Joachim Schimpf. Bounds consistency techniques for long linear constraints. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a workshop of CP 2002*, pages 39–46, 2002.

[16] John N. Hooker. Logic, optimization and constraint programming. *INFORMS Journal of Computing*, 14:295–321, 2002.

[17] John N. Hooker, Greger Ottosson, Erlendur S. Thorsteinsson, and Hak-Jin Kim. On integrating constraint propagation and linear programming for combinatorial optimization. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 136–141, Orlando, FL, July 1999.

[18] Narendra Jussien and Vincent Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, 2000.

[19] George Katsirelos and Fahiem Bacchus. Generalized NoGoods in CSPs. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Pittsburgh, PA, 2005.

[20] Hui X. Li and Brian Williams. Generalized conflict learning for hybrid discrete linear optimization. In *Proceedings of the Eleventh International Conference on the Principles and Practice of Constraint Programming (CP)*, 2005.

[21] J P Marques-Silva and K A Sakallah. GRASP-A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[22] F. Maruyama, Y. Minoda, Sawada S., and Y. Takizawa. Constraint satisfaction and optimisation using nogood justifications. In *Proceedings of the Second Pacific Rim Conference on Artificial Intelligence*, 1992.

[23] David McAllester. Truth maintenance. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Boston, MA, 1990.

[24] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *38th Design Automation Conference (DAC)*, 2001.

[25] George Nemhauser and Laurence Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1999.

[26] Manfred Padberg and Giovanni Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6:1–7, 1987.

[27] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[28] E Thomas Richards and Barry Richards. Nogood learning for constraint satisfaction. In *International Conference on Principles and Practice of Constraint Programming (CP)*, 1996.

[29] Martin Sachenbacher and Brian C. Williams. Solving soft constraints by separating optimization and satisfiability. In *Proceedings of the Seventh International Workshop on Preferences and Soft Constraints*, 2005.

[30] Tuomas Sandholm. Winner determination in combinatorial exchanges, 2003. Slides from the FCC Combinatorial Bidding Conference, Queenstown, MD, Nov. 21–23. URL: http://wireless.fcc.gov/ auctions/ conferences/ combin2003/ presentations/ Sandholm.ppt.

[31] Tuomas Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. Winner determination in combinatorial auction generalizations. In *International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 69–76, Bologna, Italy, July 2002.

[32] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency directed backtracking in a system for computer aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[33] Michael A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118:73–84, 2003.

[34] Roie Zivan and Amnon Meisels. Conflict based backjumping for constraints optimization problems. In *Proceedings of the Seventh International Workshop on Preferences and Soft Constraints*, 2005.