Stronger Inference Through Implied Literals from Conflicts and Knapsack Covers

Tobias Achterberg[1], Ashish Sabharwal[2], and Horst Samulowitz[2]

[1] IBM, Germany
achterberg@de.ibm.com
[2] IBM Watson Research Center, Yorktown Heights, USA
{ashish.sabharwal,samulowitz}@us.ibm.com

Abstract. Implied literals detection has been shown to improve performance of Boolean satisfiability (SAT) solvers for certain problem classes, in particular when applied in an efficient dynamic manner on learned clauses derived from conflicts during backtracking search. We explore this technique further and extend it to mixed integer linear programs (MIPs) in the context of conflict constraints. This results in stronger inference from clique tables and implication tables already commonly maintained by MIP solvers. Further, we extend the technique to knapsack covers and propose an efficient implementation. Our experiments show that implied literals, in particular through stronger inference from knapsack covers, improve the performance of the MIP engine of IBM ILOG CPLEX Optimization Studio 12.5, especially on harder instances.

1 Introduction

Systematic solvers for combinatorial search and optimization problems such as Boolean satisfiability (SAT), constraint satisfaction (CSP), and mixed integer programming (MIP) are often based on variants of backtracking search performed on an underlying search tree. A key to their effectiveness is the ability to prune large parts of the search tree without explicit exploration. This is done through inference during search, which can take two different forms. First, "probing" techniques [8, 11] are performed in advance to explore the potential effect of making a certain choice. Second, post-conflict analysis [1, 9] is conducted after the search has run into a conflict (i.e., an infeasible or non-improving region of the search space) in an attempt to learn and generalize the "cause" of that conflict. Stronger inference captured in the form of redundant constraints or tighter variable bounds typically leads to stronger propagation of constraints and more pruning of the search space during search. The goal of this work is to develop a technique for stronger inference during search through the concept of *implied literals*, which we apply dynamically (i.e., during search) to enhance the amount of propagation achieved from both the original problem constraints as well as constraints learned from conflicts.

In the context of SAT where the inference mechanism is unit propagation of clauses, one may perform a form of probing [8] that simply applies unit propagation to all individual literals (i.e., variables or their negations) at the root

node of the search in order to detect *failed literals* [5] (i.e., literals setting which to 1 leads to a conflict by unit propagation) or to populate *implication lists* of literals containing implications of the form $x \to y$. The latter information can then, for instance, be used to shrink clauses by using *hidden literal elimination* (e.g., if $a \to b$ then $(a \vee b \vee c)$ can be reduced to $(b \vee c)$ [cf. 6]).

Most pertinent to this work, Heule et al. [6], Soos [12], and Matsliah et al. [10] have independently proposed ways to strengthen clause learning performed by SAT solvers by dynamically inferring implied literals, i.e., literals that the newly learned clause entails. A literal l is an *implied literal* for a clause C if all literals of C entail l. For instance, if $a \to d$, $\neg b \to d$, and $c \to d$, then $(a \vee \neg b \vee c)$ entails d. This observation forms the basis of several other techniques such as variations of *hyper binary resolution* and hidden literal elimination briefly mentioned above. To apply the technique during clause learning, one generates and periodically updates implication lists $L(l) = \mathit{UnitPropagation}(l)$ for each literal l. During this computation, one may also add not yet existing binary clauses corresponding to $\neg l \to \neg p$ for all $l \in L(p)$, and detect failed literals and add and propagate their negations as new unit literals. One may do the same also for all literals in the intersection of $L(p)$ and $L(\neg p)$. This strengthens the inference that unit propagation achieves. Matsliah et al. [10] have shown that by implementing the underlying data structures efficiently and using heuristics to guide choices such as how many implied literals and binary clauses to keep and how often to recompute implication lists, this technique can boost the performance of state-of-the-art SAT solvers such as Glucose 2.0 [3], especially on certain kinds of benchmarks coming from the planning domain.

We explore this concept further and extend it to mixed integer linear programs (MIPs). While implied literals can be derived using probing for failed literals or hyper binary resolution in the case of SAT [4], this turns out not to be the case for MIPs with non-binary variables. Further, rather than imposing the overhead of deriving and maintaining our own implication lists, we capitalize on the fact that MIP solvers often already internally maintain implications in the form of clique tables and implication tables. We propose algorithms to derive implied literals from conflict constraints in MIP solvers as well as from covers of knapsack constraints, while keeping the computational overhead low. Our experimental results on over 3,000 MIP instances show that the application of implied literals improves performance, especially on harder instances. Of the 1,096 benchmark instances that need at least 10 seconds to solve, implied literals detection affects 65% of the instances, speeding up the solution process on these instances by 6% and reducing the number of search tree nodes by 7%.

2 Implied Literals in Mixed Integer Programs

Let $N := \{1, \ldots, n\}$ be an index set for variables $z_j \in \mathbb{R}$, $j \in N$. Further, let $N_X \subseteq N$ be the subset of binary variables and $N_Y = N \setminus N_X$ the subset of non-binary variables. The set $N_I \subseteq N_Y$ denotes the indices of general integer

variables. We consider the mixed integer (linear) program

$$\begin{aligned}
\max \quad & \sum_{j \in N} c_j z_j \\
\text{s.t.} \quad & \sum_{j \in N} a_{ij} z_j \leq b_i \quad \text{for all } i = 1, \ldots, m \\
& z_j \in \{0,1\} \qquad \text{for all } j \in N_X \\
& z_j \in \mathbb{Z} \qquad \text{for all } j \in N_I \subseteq N_Y \\
& z_j \in \mathbb{R} \qquad \text{for all } j \in N_Y \setminus N_I
\end{aligned}$$

with objective coefficients $c \in \mathbb{R}^n$, right hand sides $b \in \mathbb{R}^m$, and coefficient matrix $A = (a_{ij})_{ij} \in \mathbb{R}^{m \times n}$. For ease of presentation we use the symbols $x_j := z_j$ for binary variables, $j \in N_X$, and $y_j := z_j$ for non-binary variables, $j \in N_Y$. Moreover, we define $X := \{x_j \mid j \in N_X\}$ and $Y := \{y_j \mid j \in N_Y\}$ to be the sets of binary and non-binary variables, respectively.

For $x \in X$, let $\bar{x} := 1 - x$ denote the "negated" binary variable, and let $\overline{X} := \{\bar{x} \mid x \in X\}$. We define $Z := X \cup \overline{X} \cup Y$. For $z \in Z$, $\mathrm{Dom}(z)$ denotes the domain of z.

Following Achterberg [1], we define *literals* in the context of MIP as bound inequalities of the form $(z \leq u)$ or $(z \geq l)$, where $z \in Z$ and $l, u \in \mathrm{Dom}(z)$. For the sake of simplicity of notation, we will work only with literals of the form $(z \leq u)$ in the rest of this paper. The arguments and constructs naturally generalize to any combination of literals of the form $(z_1 \leq u_1)$ and $(z_2 \geq l_2)$.

Definition 1. *Suppose we are given the implications $(z_i \leq u_i) \rightarrow (z \leq d_i)$ for $i \in \{1, \ldots, k\}$, where $z_i, z \in Z$, $u_i \in Dom(z_i)$, and $d_i \in Dom(z)$. Let $d_{max} := \max_{i=1}^{k} d_i$. Let $C := \bigvee_{i=1}^{k} (z_i \leq u_i)$ be a constraint. Then $(z \leq d_{max})$ is called an* implied literal *derived from C.*

We will sometimes refer to $(z \leq d_{\max})$ simply as an implied literal when C and the implications needed for its derivation are implicit in the context. Note that when $z_i = x$ for $x \in X$, the implication $(z_i \leq u_i) \rightarrow (z \leq d_i)$ in Definition 1 is equivalent to $(x = 0) \rightarrow (z \leq d_i)$. Similarly, when $z_i = \bar{x}$ for $\bar{x} \in \overline{X}$, the implication may equivalently be written as $(x = 1) \rightarrow (z \leq d_i)$.

2.1 Implied Literals from Conflict Constraints

As in the case of SAT, the idea is to efficiently store a number of implications and use them to infer implied literals when, for example, a new constraint is derived or added to the model. Fortunately, current MIP solvers typically already store certain types of implications of literals, which are used in presolving and during the branch and bound process. For example, CPLEX 12.5 internally maintains implications of the following types:

$$\begin{aligned}
(x_i = v_i) \rightarrow (x_j = v_j) \qquad & x_i, x_j \in X, v_i, v_j \in \{0,1\} \\
(x = v) \rightarrow (y \leq d) \qquad & x \in X, y \in Y, v \in \{0,1\}, d \in \mathrm{Dom}(y)
\end{aligned}$$

Implications of the first type are stored in a *clique table* $\mathcal{K}$. For example, the set packing constraint $\sum_{i=1}^{k} x_i \leq 1$ leads to a "clique" of implications $(x_i = 1) \rightarrow (x_j = 0)$ for $i \neq j$. The clique table stores cliques $K \in \mathcal{K}$, $K \subseteq X \cup \overline{X}$, in aggregated form as set packing constraints $\sum_{x \in K} x \leq 1$, rather than explicitly recording the $|K|(|K|-1)$ implications that are entailed by the clique K. Implications of the second type are stored in an *implication table* $\mathcal{I}$. For example, a "variable upper bound" constraint $-ax + y \leq b$ with $x \in X$ and $y \in Y$ yields the implication $(x = 0) \rightarrow (y \leq b)$. Cliques and implications can be directly extracted from general linear constraints and computed with techniques such as probing. Note that CPLEX does not store implications between pairs of non-binary variables such as $(y_1 \leq u) \rightarrow (y_2 \leq d)$ with $y_1, y_2 \in Y$.

Example 1. Consider the constraint $C = (x_1 \geq 1) \vee (x_2 \leq 0) \vee (y_3 \geq 4) \vee (y_4 \leq 8)$ and the following cliques (left) and implications (right):

$$\begin{aligned} \bar{x}_1 + \bar{x}_2 + \bar{x}_5 &\leq 1 \\ x_1 + x_6 &\leq 1 \\ \bar{x}_2 + x_6 &\leq 1 \end{aligned} \qquad\qquad \begin{aligned} (x_1 = 0) &\rightarrow (y_3 \leq 2) \\ (x_1 = 0) &\rightarrow (y_4 \geq 12) \end{aligned}$$

with $x_i \in X$ and $y_j \in Y$. Including the self-implicants, we get:

$$\begin{array}{llll} (x_1 \geq 1) \rightarrow (x_1 \geq 1), & (y_3 \geq 4), & & (x_6 \leq 0) \\ (x_2 \leq 0) \rightarrow (x_1 \geq 1), (x_2 \leq 0), & & & (x_5 \geq 1), (x_6 \leq 0) \\ (y_3 \geq 4) \rightarrow (x_1 \geq 1), & (y_3 \geq 4) & & \\ (y_4 \leq 8) \rightarrow (x_1 \geq 1), & & (y_4 \leq 8) & \end{array}$$

Since $x_1 \geq 1$ is implied by all literals of C, it is an implied literal and we can permanently fix $x_1 := 1$.

Since a MIP solver already maintains cliques and implications, we can exploit them "for free" in the implied literals detection. Whenever a conflict constraint

$$C = \bigvee_{i=1}^{k} (z_i \leq u_i)$$

is derived for an infeasible node in the search tree, we apply Algorithm 1 to infer implied literals. The goal is to find new bounds $z \leq d$ that are implied by all of the k literals of C. Thus, for each variable $z \in Z$ we count the number of literals $(z_i \leq u_i)$ that imply a bound $z \leq d_i$. If this count reaches $\mathit{count}[z] = k$ at the end of the algorithm, then an implied literal has been identified. In the case of a binary variable $z = x$ (line 18) we have $d_i = 0$ for all i, and the implied literal is $(x = 0)$. In the case of a non-binary variable $z = y$ (line 20) the implied literal is $(y \leq \mathit{maxd}[y])$ where maxd is an array that tracks the maximal implied bound d_i for non-binary variables $y \in Y$.

For each of the k literals of the conflict constraint C, the algorithm inspects the implications in the main loop of line 4 to update the *count* and *maxd* arrays. If the literal variable z_i is binary (line 6), then implications on other binary

Algorithm 1: Deriving Implied Literals from MIP Conflict Constraints

Input : variables $Z = X \cup \overline{X} \cup Y$, conflict constraint $C = \bigvee_{i=1}^{k}(z_i \leq u_i)$, clique table $\mathcal{K}$, implication table $\mathcal{I}$

Output: a set L of implied literals derived from C

```
1  begin
2      initialize count[z] := 0 for all z ∈ Z
3      initialize maxd[y] := −∞ for all y ∈ Y
4      forall i ∈ {1, ..., k} do
5          increment count[z_i]
6          if z_i ∈ X ∪ X̄ (and thus u_i = 0), then
7              forall K ∈ K with z̄_i ∈ K do
8                  forall x ∈ K \ {z̄_i} with count[x] = i − 1 do
9                      increment count[x]
10             forall ((z_i = 0) → (y ≤ d)) ∈ I with count[y] = i − 1 do
11                 increment count[y]
12                 maxd[y] := max{maxd[y], d}
13         else
14             maxd[z_i] := max{maxd[z_i], u_i}
15             forall ((x = 1) → (z_i ≥ u)) ∈ I with u > u_i and count[x] = i − 1 do
16                 increment count[x]
17     L := ∅
18     forall x ∈ X with count[x] = k do
19         set L := L ∪ {(x = 0)}
20     forall y ∈ Y with count[y] = k do
21         set L := L ∪ {(y ≤ maxd[y])}
22     return L
23 end
```

variables x' can be found in the clique table, namely by inspecting the cliques $K \in \mathcal{K}$ with $\bar{z}_i \in K$, see line 7. Implications involving non-binary variables y can be inferred from the implication table, as done in line 10. On the other hand, if z_i is non-binary (line 13), then the implication table yields implications on binary variables by reversing the direction of the implication. Namely, an implication $(x = 1) \rightarrow (y \geq u)$ with binary $x \in X \cup \overline{X}$ can be rewritten as $(y < u) \rightarrow (x = 0)$, resulting in the implication of $(x = 0)$, see line 15.

Note that we need to include the trivial self-implicants $(z_i \leq u_i) \rightarrow (z_i \leq u_i)$ in the counting, see lines 5 and 14, in order to not miss the cases in which one of the literals of C is an implied literal. Note also that in each iteration i we only consider variables for which the *count* value is maximal, i.e., equal to $i-1$. This makes sure that we do not count variables twice for the same conflict clause literal $(z_i \leq u_i)$ and it prevents unnecessary updates on variables that cannot be implied literals.

Performance improvements. It is easy to see that Algorithm 1 can be enhanced in order to avoid unnecessary work. If there is more than one non-binary variable in C, then we can initialize $\mathit{count}[y] = -1$ for all $y \in Y$, because there are no implications between pairs of non-binary variables and thus no implied literal can be found for non-binary variables. If there is exactly one non-binary variable y_i in C, then we can initialize $\mathit{count}[y] = -1$ for all $y \in Y \setminus \{y_i\}$.

After each iteration $i \in \{1, \ldots, k\}$ of the main loop, let $L_i = \{z \in Z \mid \mathit{count}[z] = i\}$. As soon as $L_i = \emptyset$, the loop can be aborted since no implied literals exist. Moreover, if $L_i \cap Y = \emptyset$, then we no longer need to inspect the implication table $\mathcal{I}$ for binary literal variables $z_j \in X \cup \overline{X}$, $j > i$, because there cannot be any implied literals for non-binary variables. On the other hand, if $L_i \cap (X \cup \overline{X}) = \emptyset$, then we can stop inspecting the clique table $\mathcal{K}$ for literals with binary variable $z_j \in X \cup \overline{X}$, $j > i$, and for non-binary variables $z_j \in Y$, $j > i$, we only need to consider the self-implication by incrementing $\mathit{count}[z_j]$ and updating $\mathit{maxd}[z_j]$.

2.2 Implied Literals from Knapsack Covers

We will now consider the derivation of implied literals based on knapsack constraints of the form $\sum_{i=1}^{k} a_i x_i \leq b$ with $x_i \in X \cup \overline{X}$ and $a_i \geq 0$. A *cover* R is a subset of $\{1, \ldots, k\}$ such that $\sum_{i \in R} a_i > b$. We will be interested in *minimal covers* which are covers whose proper subsets are not covers. Clearly, if $x_i = 1$ for all $i \in R$, the knapsack constraint will be violated. Hence, the cover R entails the implicit constraint $\bigvee_{i \in R}(x_i = 0)$, which we will refer to as the *cover constraint* corresponding to R. Suppose further that we have implications $(x_i = 0) \rightarrow (z \leq d_i)$ for all $i \in R$ and some $z \in Z$. Then, using the cover constraint along with these implications, we can derive the implied literal $z \leq \max\{d_i \mid i \in R\}$.

Remark 1. Suppose we are given an arbitrary linear constraint of the form $\sum_{i=1}^{p} c_i x_i + \sum_{j=1}^{q} e_j y_j \leq r$ where, as before, $x_i \in X$ are binary variables and $y_j \in Y$ are non-binary variables. We can derive implied literals from this constraint by first relaxing it into a knapsack constraint of the form considered above by (i) using global bounds $l_j \leq y_j \leq u_j$, (ii) using the implication table to derive variable bounds of the form $a_{j,l} x_{j,l} + b_{j,l} \leq y_j \leq a_{j,u} x_{j,u} + b_{j,u}$, and (iii) complementing the binary variables if necessary to make their coefficients non-negative.

Since the number of minimal covers of a knapsack constraint can be exponential in the length of the knapsack, it is prohibitive to naïvely enumerate all minimal covers and apply Algorithm 1 to the corresponding cover constraints. Instead, we propose a more efficient method, described as Algorithm 2.

The main idea of the algorithm is to consider the implications $(x_i = 0) \rightarrow (z \leq d_i)$ for a given variable $z \in Z$ in the reverse direction, namely as $(z > d_i) \rightarrow (x_i = 1)$. For $d \in \mathrm{Dom}(z)$ we define the set of *implied knapsack variables*

$$I_{(z>d)} := \Big\{ i \in \{1, \ldots, k\} \;\Big|\; \exists\big((x_i = 0) \rightarrow (z \leq d_i)\big) \in \mathcal{I} \text{ with } d_i \leq d \Big\}$$

Algorithm 2: Deriving Implied Literals from MIP Knapsack Covers

Input : variables $Z = X \cup \overline{X} \cup Y$, knapsack constraint $\sum_{i=1}^{k} a_i x_i \leq b$ with $x_i \in X \cup \overline{X}$ for $i = 1, \ldots, k$, clique table $\mathcal{K}$, implication table $\mathcal{I}$

Output: a set L of implied literals derived from the knapsack

```
1  begin
2      initialize weight[z] := 0 for all z ∈ Z
3      initialize I_y := ∅ for all y ∈ Y
4      forall i ∈ {1, ..., k} do
5          set T := {x_i}
6          set weight[x_i] := weight[x_i] + a_i
7          forall K ∈ K with x̄_i ∈ K do
8              forall x ∈ K \ {x̄_i} \ T do
9                  set T := T ∪ {x}
10                 set weight[x] := weight[x] + a_i
11         forall ((x_i = 0) → (y ≤ d_i)) ∈ I with y ∉ T do
12             set T := T ∪ {y}
13             set weight[y] := weight[y] + a_i
14             set I_y := I_y ∪ {{(x_i = 0) → (y ≤ d_i)}}
15     L := ∅
16     forall x ∈ X with weight[x] > b do
17         set L := L ∪ {(x = 0)}
18     forall y ∈ Y with weight[y] > b do
19         I_sorted := I sorted by non-decreasing d_i
20         set s := 0
21         forall ((x_i = 0) → (y ≤ d_i)) ∈ I_sorted do
22             set s := s + a_i
23             if s > b then
24                 L := L ∪ {(y ≤ d_i)}
25                 break
26     return L
27 end
```

and call

$$a_{(z>d)} := \sum_{i \in I_{(z>d)}} a_i$$

the *implied weight* of $(z > d)$. If $a_{(z>d)} > b$, then $z > d$ implies that the knapsack constraint is violated, and we can conclude $z \leq d$.

For binary variables $z = x \in X \cup \overline{X}$, all non-trivial implications $(x_i = 0) \rightarrow (x \leq d_i)$ have $d_i = 0$, and we can fix $x = 0$ if and only if $a_{(x=1)} > b$. This is done in the algorithm by adding up the implied weights in the *weight* array and updating $weight[x]$ for binary variables x by scanning the clique table $\mathcal{K}$ in line 7. Again, we need to consider the trivial self-implications $(x_i = 0) \rightarrow (x_i = 0)$, see line 6. The implied fixings of binary variables are then collected in line 16.

For non-binary variables $z = y \in Y$ we want to find the *smallest* bound d that yields an implied weight $a_{(y>d)} > b$. In order to find this smallest implied bound d we first collect the implications $(x_i = 0) \to (y \leq d_i)$ of the implication table $\mathcal{I}$ for all knapsack variables x_i in a set $\mathcal{I}_y$, see line 11. To evaluate these sets, we sort them by non-decreasing d_i in line 19. Then, in the loop of line 21, we consider the non-decreasing sequence of implied weights $a_{(y>d_i)}$ by adding up the knapsack weights a_i in this order until the capacity b is exceeded for the first time at element $i^\star$, that is, $i^\star$ is the first index in the sorted order such that $a_{(y>d_{i^\star})} > b$. If this process succeeds, then $d_{i^\star}$ is the smallest valid implied upper bound for y that can be derived from the knapsack and the implications using the reasoning of the algorithm. If even the total implied weight $\mathit{weight}[y]$ does not exceed the capacity, then we cannot tighten the upper bound of y.

Performance improvements. The additional performance improvements that we applied to Algorithm 2 are a bit more involved than the ones for Algorithm 1. Again, the goal is to avoid unnecessary work and abort as early as possible if no implied literals can be found. This is achieved by tracking the implied knapsack weight $\mathit{weight}[z]$ of the variables $z \in Z$ and the maximal remaining weight $w_i = \sum_{j=i+1}^{k} a_j$ of knapsack items that we did not yet consider at iteration i of the main loop. If $\mathit{weight}[z] + w_i \leq b$ at the end of an iteration, it is clear that we will not be able to find an implied literal for z. If this is true for all binary variables, then we can stop looking at the clique table $\mathcal{K}$. If it is the case for all non-binary variables, the implication table $\mathcal{I}$ becomes uninteresting. Finally, we can abort the loop if no variable remains with $\mathit{weight}[z] + w_i > b$.

Now it becomes important in which order we process the knapsack items. To be able to abort as soon as possible, we want to first look at items that have large weight, so that w_i decreases fast. Moreover, items that trigger only few implications should be preferred, which will lead to the $\mathit{weight}[z]$ values staying small. In our implementation, we sort the knapsack items x_i in a non-decreasing order defined by $|\{K \in \mathcal{K} \mid \bar{x}_i \in K\}| - 10\, a_i/b$.

Finally, it is useful to observe that there are no negated self-implications $(x = 1) \to (x = 0)$, $x \in X \cup \overline{X}$, in the clique table; otherwise we would have already fixed $x = 0$ during presolve. As a consequence, we know that nothing can be deduced for x_j, $j = i+1, \ldots, k$, if $\mathit{weight}[x_j] + w_i - a_j \leq b$ after iteration i. For set covering knapsacks $\sum_{i=1}^{k} x_i \leq k - 1$, this means that we can rule out a variable as soon as there is a knapsack item without an implication to this variable. Hence, Algorithm 2 coincides with Algorithm 1 if applied to set covering constraints.

3 Empirical Evaluation

We evaluated the proposed techniques on a benchmark set containing 3,189 MIP models from public and commercial sources.[3] All experiments were conducted

[3] Due to proprietary rights, the commercial benchmarks are not publicly disclosed. However, detailed anonymized data from our experiments may be found at the follow-

Table 1. Comparison of default CPLEX 12.5 vs. disabling implied literals detection

		CPLEX 12.5	no implied literals detection					affected		
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time	nodes
all	3172	97	101	249	270	**1.01**	**1.01**	1134	**1.03**	**1.03**
[0,10k]	3093	18	22	249	270	**1.01**	**1.01**	1134	**1.03**	**1.03**
[1,10k]	1861	18	22	246	268	**1.02**	**1.02**	1005	**1.04**	**1.04**
[10,10k]	1096	18	22	194	226	**1.04**	**1.04**	713	**1.06**	**1.07**
[100,10k]	579	18	22	128	147	**1.05**	**1.06**	430	**1.06**	**1.08**
[1k,10k]	232	18	22	63	73	**1.06**	**1.06**	191	**1.07**	**1.07**

on a cluster of identical 12 core Intel Xeon E5430 machines running at 2.66 GHz and equipped with 24 GB of memory. A time limit of 10,000 seconds and a tree memory limit of 6 GB was employed for all runs. When the memory limit was hit, we set the solve time to 10,000 seconds and scale the number of nodes processed for the problem instance accordingly.

Implied literals detection was implemented in CPLEX 12.5 (the "default" for the purposes of this section). Algorithm 1 is applied for each conflict constraint that is derived for infeasible search tree nodes, while Algorithm 2 is only applied during the presolving stage since knapsack constraints are not generated on the fly during the tree search of CPLEX.

Table 1 shows a summary of our computational experiments comparing default CPLEX with a modified CPLEX variant where implied literals detection was disabled. Both variants were run with the default parameter settings of CPLEX. We first note that across all models in our MIP test set, Algorithms 1 and 2 never took more than 0.08 and 0.18 seconds, respectively, for each invocation of the algorithm. The times reported for default CPLEX include the overhead of computing and reasoning with implied literals.

Column 1 of the table, "bracket", labels subsets of problem instances with different "hardness", each row representing a different such subset. Subset "all" is the set of all models used for the first row of data. The labels "[n,10k]" represent the subset of "all" models for which at least one of the solvers being compared took at least n seconds to solve, and that were solved to optimality within the time limit by at least one of the solvers.

Column 2, "models", shows the number of problem instances in each subset. Note that only 3,172 rather than 3,189 problem instances are listed in row "all", because we excluded those 17 models for which the two solvers reported different optimal objective values. These inconsistencies result from the inexact floating point calculations employed in CPLEX. For ill-posed problem instances such numerical difficulties cannot be completely ruled out by floating point based MIP solvers, and this does not point to a logical error in any of the two solvers.

ing URL: http://researcher.watson.ibm.com/researcher/files/us-ashish.sabharwal-/CPAIOR2013-impliedLitsMIP.txt.

Column 3, "tilim", gives the number of models in each subset for which default CPLEX hit the time or memory limit. It is by design that the numbers in the last five rows match, since these models are included in all five of the corresponding subsets. Column 4 gives the corresponding numbers when not using implied literals. Both variants hit the time or memory limit on the same 79 instances of our test set. Default CPLEX hit a limit on 18 additional instances on which modified CPLEX did not, and the converse is true for 22 instances. This marginal difference of four models is likely due to performance variability [cf. 7] across runs and not an indication of the strength or weakness of one variant of CPLEX over the other.

Columns 5, "faster", and 6, "slower", show the number of models in each subset for which disabling implied literals detection resulted in the instance being solved at least 10% faster or slower, respectively, than the baseline solver (which applied implied literals). As with the time limit hits, there is only a marginal difference between the two variants of CPLEX, which is again somewhat in favor of implied literals.

Column 7, "time" displays the shifted geometric mean of the ratios of solution times [2] with a shift of $s = 1$ second.[4] A value $t > 1$ in the table indicates that CPLEX without implied literals detection is a factor of t slower (in shifted geometric mean) than default CPLEX. Column 8, "nodes", is similar to the previous column but shows the shifted geometric mean of the ratios of the number of branch-and-cut nodes needed for the models by each solver, using a shift of $s = 10$ nodes. Note that when a time limit is hit, we use the number of nodes at that point. Recall that when a memory limit is hit, we scale the node count by $10000/t$ with t being the time at which the solving process was aborted.

We observe from the time and nodes ratios that implied literals detection speeds up the solving process on average and reduces the size of the search tree explored. In particular, for models that take more than 100 seconds by at least one of the two solvers, implied literals detection reduces the solving time by 5% and the number of search tree nodes by as much as 6%.

The last three columns, under the heading "affected", report the impact on the subset of models in each bracket for which the use of implied literals had an effect on the *solution path* itself. Here, we assume that the solution path is identical if both the number of nodes and the number of simplex iterations are identical for the two solvers. Column 9, "models", shows that implied literals lead to a path change for about 36% of the models, and this fraction increases as the solving difficulty increases. For instance, for models that take at least 100

[4] The use of arithmetic means also resulted in a similar overall picture as the shifted geometric means we report here. In general, the use of geometric means, as opposed to arithmetic means, prevents situations where a small relative improvement by one solver on one long run overshadows large improvements by the other solver on many shorter runs. Further, a shift of 1 second guarantees that large but practically immaterial relative improvements on extremely short runs (e.g., 0.05 sec improving to 0.01 sec) do not distort the overall geometric mean.

seconds to solve, over 74% of the instances are affected and the speed-up on those instances is 6%.

4 Conclusion

We extended the concept of implied literals from SAT literature to the context of conflict constraints and knapsack covers in MIPs. Our empirical results show that while the application of this technique does not significantly improve the number of instances solved within the time limit, it does speed up the solution process. For example, it affected a substantial number of MIP models that need over 10 seconds to solve, where it reduced the solution time by 6% on average and the number of search nodes explored by 7%. We found the technique to generally have a higher impact on the solution path of harder instances and provide larger performance improvements on them.

We close with a contrast to the SAT domain. While implied literals detection can be very beneficial on certain SAT benchmark families [10], its general application can be prohibitive due to the significant computational overhead. Consequently, implied literals detection is currently not a standard technique in state-of-the-art SAT solvers. However, as our results on over 3,000 MIP instances demonstrate, implied literals detection can be employed in a way that serves as a useful generic additional inference technique in the context of MIP solvers.

References

[1] T. Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, March 2007. Special issue: Mixed Integer Programming.

[2] T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, July 2007.

[3] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *21st IJCAI*, pp. 399–404, Pasadena, CA, July 2009.

[4] F. Bacchus, A. Biere, and M. Heule. Personal Communication, 2012.

[5] J. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[6] M. Heule, M. Järvisalo, and A. Biere. Efficient CNF simplification based on binary implication graphs. In *14th SAT*, pp. 201–215, Ann Arbor, MI, June 2011.

[7] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3:103–163, 2011.

[8] I. Lynce and J. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *15th ICTAI*, pp. 105–111, Sacramento, CA, Nov. 2003.

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

[10] A. Matsliah, A. Sabharwal, and H. Samulowitz. Augmenting clause learning with implied literals. In *15th SAT*, pp. 500–501, Trento, Italy, June 2012.

[11] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[12] M. Soos. CryptoMiniSat 2.9.x, 2011. URL http://www.msoos.org/cryptominisat2.