EPiC Series in Computer Science

Volume 36, 2015, Pages 1–13

GCAI 2015. Global Conference on Artificial Intelligence

SAT-Based Techniques for Integer Linear Constraints

Robert Nieuwenhuis

Technical University Catalonia (UPC), Barcelona, Spain
and
Barcelogic.com, Barcelona, Spain
roberto@cs.upc.edu

Abstract

Conflict-Driven Clause-Learning (CDCL) SAT and SAT Modulo Theories (SMT) solvers are well known as workhorses for, e.g., formal verification applications. Here we discuss ways to go beyond by learning not only clauses, but also much more expressive constraints. We outline techniques for Integer Linear Programming (ILP), going first from SAT to SMT for ILP and then to SMT with *on-the-fly bottleneck constraint encoding*. Then we illustrate the power of learning full constraints, and the resulting methods for 0-1 ILP (Pseudo-Boolean solvers) and full ILP (Cutsat and IntSat), outlining difficulties and their solutions, giving examples and some intuition on why these techniques work so well.

1 Introduction

One of the most remarkable successes of AI technology is probably the combination of heuristics, learning and novel data structures that allow Conflict-Driven Clause Learning (CDCL) propositional SAT solvers to fully automatically handle very large, hard, real-world industrial and scientific problem instances. It is therefore not surprising that for many years researchers have tried to produce effective extensions of CDCL able to handle richer constraint languages. In fact, SAT can be seen as the particular case of Integer Linear Programming (ILP) where all variables are binary (0-1) and constraints are *clauses*, sets (disjunctions) of *literals* $x_1 \vee \ldots \vee x_m \vee \overline{y_1} \vee \ldots \vee \overline{y_n}$, i.e., constraints $x_1 + \ldots + x_m + (1 - y_1) + \ldots + (1 - y_n) \geq 1$. Of course arbitrary ILP constraints can be encoded into clauses, but, as we will see here, handling them natively opens the door to far more compact and efficient reasoning, learning, and optimization.

Indeed, an extensive amount of work exists on CDCL-like techniques for Pseudo-Boolean solving (aka. 0-1 ILP), which considers Boolean variables, arbitrary linear constraints of the form $a_1 x_1 + \cdots + a_n x_n \leq a_0$ and optimizing an objective function $b_1 y_1 + \cdots + b_m x_m$ where the coefficients a_i, b_j are integers (see [20] for all background and references).

The Cutsat [12] and IntSat [17] ILP solvers go another step beyond, introducing CDCL techniques for full ILP (arbitrary integer variables, linear constraints and objective). As reported in [17], IntSat appears to be the first ILP method that is competitive with state-of-the-art commercial tools such as CPLEX and Gurobi (which solve ILP by combining LP relaxations, simplex, branch-and-cut and many other methods, see [11]). This is the case already for the first IntSat

prototype implementations (single-core, unlike its competitors), performing especially well on problems having a more combinatorial (as opposed to numerical) nature.

In this paper we outline these new techniques, along with other solutions based on Sat Modulo Theories with or without *on-the-fly bottleneck constraint encoding*. The contents of this paper is as follows:

- In Section 2 we describe CDCL SAT solving and give our intuition about the three key ingredients that make it behave so well and which we try to carry over to richer constraint languages.
- Section 3 briefly outlines SAT Modulo Theories (SMT) and our DPLL(T) approach to SMT.
- We discuss an SMT-based method for ILP (Section 4).
- We give some intuition about why SMT works so well if there are few or no *bottleneck* constraints, and explain how the SMT-based method can be improved by the selective on-the-fly SAT encoding of such bottlenecks. (Section 5).
- We consider methods that learn new linear constraints, and give intuition on why this can be much more powerful than just learning new clauses (Section 6).
- Section 7 describes some methods for 0-1 ILP, from the pure SMT solution to constraint-learning solvers.
- Section 8 is on CDCL-like constraint-learning solvers for full ILP (i.e., with optimization and solutions over $\mathbb{Z}$). In particular we discuss Cutsat [12] and IntSat [17].

2 CDCL SAT solving

SAT is the particular case of ILP where all variables are binary (0/1) and constraints have the form $x_1 + \ldots + x_m - y_1 \ldots - y_n > -n$, written as *clauses* $x_1 \vee \ldots \vee x_m \vee \overline{y_1} \vee \ldots \vee \overline{y_n}$, i.e., sets (disjunctions) of *literals*. Given a *partial assignment* A, seen as a set of (non-contradictory) literals, a clause C is *true in* A if $A \cap C \neq \emptyset$, it is *false* or a *conflict* if $\overline{l} \in A$ for every literal l in C, and it is *undefined* otherwise. All essential CDCL features are described in the following 14-line algorithm, where A is seen as an (initially empty) stack:

The basic CDCL algorithm:

1. *Propagate*: while possible and no conflict appears, if, for some clause $l \vee C$, C is false in A and l is undefined, push l onto A, associating to l the *reason clause* C.
2. **if** there is no conflict
   if all variables are defined in A, output "solution A" and halt.
   else *Decide*: push some undefined literal l, marked as a *decision*, and go to 1.
3. If A contains no decisions, output "unsatisfiable" and halt.
4. Use a clause data structure C. Initially, let C be any conflict.
   - *Conflict analysis*: Invariant: C is false in A, that is, if $l \in C$ then $\bar{l} \in A$.
     If l is the literal of C whose negation is topmost in A, and D is the reason clause of $\bar{l}$, then replace C by $(C \setminus \{l\}) \cup D$. Repeat this until there is only one literal l_{top} in C such that $\overline{l_{top}}$ is, or is above, A's topmost decision.
   - *Backjump*: pop literals from A until either there are no decisions in A or, for some l in C with $l \neq l_{top}$, there are no decisions above $\bar{l}$ in A.
   - *Learn*: add the final C as a new clause, and go to 1 (where C propagates l_{top}).

Note that replacing C by $(C \setminus \{l\}) \cup D$ is in fact an inference by *resolution* between C and $D \vee \bar{l}$. Essentially all state-of-the-art CDCL SAT solvers use this (so called *1UIP* conflict analysis-based) algorithm, with efficient data structures for propagation, heuristics that select *recently active* literals as decisions, periodically *restarting* as well as *forgetting* (removing) the least useful learned clauses, and using clause simplification and other *inprocessing* methods (see, e.g., [14, 6] for more details and further references).

In our opinion, there are three *key* ingredients that make CDCL especially effective:

- Learning at each conflict the backjump clause as a *lemma* makes propagation progressively more powerful, preventing a potentially exponential amount of repeated work in future *similar* conflicts. Note that such similar conflicts are frequent in structured industrial problems, but not in random ones, where indeed learning is not very useful.
- The decision heuristics that decide on variables with *many occurrences in recent conflicts*. Our intuition behind this dynamic activity-based heuristics (introduced in the Chaff solver [16] using the VSIDS implementation) is that it works off, one by one, clusters of tightly related variables. Indeed, if one tries a CDCL solver with this heuristic on two independent SAT instances sharing no variables. it will finish solving one instance before handling the other one.
- Forget from time to time the less useful (e.g., low-activity) lemmas. This is crucial to keep propagation fast and memory usage limited, and it may work well because the lemmas from the already worked-off clusters are no longer needed.

After decades of academic and industrial efforts, with a lot of resources from, e.g., EDA (Electronic Design Automation), altogether this technology provides complete high-performance solvers, handling real-world problems from *many* sources, with a single, fully automatic, push-button, variable selection heuristic. Hence modeling for such solvers is essentially only a declarative task.

But CDCL SAT solvers also have severe limitations. The language of clauses is low-level, so tools encoding typical constraints into SAT are needed, and sometimes no adequate or sufficiently compact encodings are available, typically for arithmetic constraints. Also, traditionally such solvers are only capable of reporting unsatisfiability or satisfiability returning a model, and optimization received less attention. Therefore more expressive frameworks appeared, such as the one we address in the next section.

3 SAT Modulo Theories (SMT)

The origin of SMT was essentially the need for reasoning about equality, arithmetic, data structures such as arrays, etc., in software and hardware verification. It amounts to deciding satisfiability of an (existential) SAT formula with atoms over a background theory T. For example, if T is *Equality with Uninterpreted Functions (EUF)* (i.e., a congruence), a (three-clause) SMT formula is $f(g(a)) \neq f(c) \ \vee \ g(a) = d, \quad g(a) = c, \quad c \neq d$. It is unsatisfiable, because in a congruence $c \neq d$ and $g(a) = c$ imply that $g(a) \neq d$ and that $f(g(a)) = f(c)$.

Another (two-clause) example combining arithmetic, the theory of arrays and EUF is:
$A = write(B, i+1, x), \qquad read(A, j+3) = y \ \vee \ f(i-1) \neq f(j+1)$.

Let us solve this example using the so-called Lazy approach to SMT, aka *Lemmas on demand* [9]. First one forgets about the meaning of each literal, and handles it as a propositional one (here named by numbers):

$$\underbrace{f(g(a)) \neq f(c)}_{\overline{1}} \ \vee \ \underbrace{g(a) = d}_{2}, \qquad \underbrace{g(a) = c}_{3}, \qquad \underbrace{c \neq d}_{\overline{4}}$$

If $\{ \ \overline{1} \vee 2, \quad 3, \quad \overline{4} \ \}$ is given to a SAT solver, a model such as $[\ \overline{1}, \ 3, \ \overline{4} \ ]$ is returned. This model is then given to a *Theory solver*, that does know the meaning of these propositional symbols as theory literals and can handle *conjunctions* of these, and it detects that this model is T-inconsistent. Then one forbids this model by adding the *blocking clause* $1 \vee \overline{3} \vee 4$ to the clause set, which is sent again to the sat solver. After repeating this, blocking also the model $[\ 1, \ 2, \ 3, \ \overline{4} \ ]$, the SAT solver detects unsatisfiability. This process always terminates with unsatisfiability or finding T-consistent model.

If a CDCL SAT solver is used, this lazy approach can be considerably improved:

1. Instead of checking the T-consistency only of full propositional models, one can incrementally check the *partial* assignment while it is being built.
2. Given a T-inconsistent assignment M, instead of adding $\neg M$ as a clause, one can ask the theory solver for an *explanation* or a *reason* of T-inconsistency, a small T-inconsistent subset of M, and add this reason as a clause.
3. Upon a T-inconsistency, instead of adding a blocking clause and restarting the SAT solver from scratch, the solver can do conflict analysis of the explanation and backjump.

All this is covered in our (nowadays standard, cf. Google) DPLL(T) approach to SMT [18]:

$$\text{DPLL(T)} = \ \text{DPLL(X) engine} \ + \ T\text{-Solvers}$$

Its main aims are modularity and flexibility: one can plug in (combinations of) T-Solvers into the DPLL(X) engine. In DPLL(T) the T-Solvers also do *Theory Propagation*, i.e., propagating literals that are theory consequences. This gives more pruning and the T-Solver also guides the search, instead of only validating it.

4 Integer Linear Programming through SMT

Assume given a set S of linear constraints of the form $c_1\ x_1 + \ldots + c_n\ x_n \leq c_0$ where $\{x_1, \ldots, x_n\}$ is a set of variables and (we assume) the *coefficients* c_i are integers. The aim is to decide the satisfiability of S over the integers, that is, the existence of a solution $Sol\colon \{x_1, \ldots, x_n\} \to \mathbb{Z}$ satisfying all such constraints, that is, $c_1 \cdot Sol(x_1) + \ldots + c_n \cdot Sol(x_n) \leq_{\mathbb{Z}} c_0$. For simplicity, here we assume that for all variables integer upper and lower bounds are known; this aspect, as well as optimization, will be addressed later on in this paper.

The SMT Theory T is then the set S, and the SMT solver will decide and propagate *bounds* of the form $lb \leq x$ and $x \leq ub$. Theory propagation of bounds simply amounts to *bound propagation* with S. For example, the bounds $\{\ 0 \leq x,\quad 1 \leq y\ \}$ and the constraint $\{\ x + y + 2z \leq 2\ \}$ entail $z \leq 0$, since $\quad 0 + 1 + 2z \leq 2 \implies 2z \leq 1 \implies z \leq \lfloor 1/2 \rfloor$. The explanation clause (disjunction of bounds) for this propagation is simply $0 \not\leq x \vee 1 \not\leq y \vee z \leq 0$. In case of a conflict, one can generate explanation clauses on demand for conflict analysis to compute the backjump clause, which is learned. Backjumping is as in standard CDCL SAT (as well as lemma forgetting, restarts, etc.). Termination and completeness of this follow from standard SMT [18]. An important aspect of this technique is that only new *clauses* are learned. The theory S does not change.

This method was later on introduced in the Constraint Programming community under the name *Lazy Clause Generation* (LCG) [19], and shown to work very well on problems such as scheduling and timetabling.

5 On-the-fly SAT encoding of bottleneck constraints

One may wonder why SMT works so well on so many practical problems. In our opinion, the main reason is that most constraints are not *bottlenecks*: they only generate few (different) explanation clauses, and SMT generates *exactly these few clauses, on demand*, thus avoiding the need of fully encoding all constraints into clauses beforehand.

However, sometimes bottleneck constraints C do occur. During the SMT solving process such a constraint C can generate very large (usually exponential) numbers of explanation clauses, and in fact all of them together can amount to an almost full, and moreover very naive, SAT encoding of C.

For example, consider a a timetabling problem with a (cardinality) constraint C of the form $x_1 + \ldots + x_n \leq 40$, stating, e.g., that only 40 time slots are available. In addition, assume that the instance is unsatisfiable because (for some possibly complicated reason) the other constraints altogether imply that strictly more than 40 slots are needed. Then the SMT solver may end up enumerating exponentially many explanation clauses for C, of the form $\neg y_1 \vee \ldots \vee \neg y_{41}$, for many of the $\binom{n}{41}$ subsets $\{y_1 \ldots y_{41}\}$ of size 41 of the n variables. This may end up being an (almost) full encoding of C, and a very naive one.

Obviously, for such C one needs a more compact encoding, using auxiliary variables (see, e.g., [1, 4] and references on such encodings). The idea of [3, 2] is to detect and encode such bottleneck constraints C on the fly, during the SMT solving process. This is a significant additional improvement.

6 Learning constraints can beat learning clauses

Given CDCL's enormous success for SAT, for decades researchers (e.g., from the SAT community, but not only) have tried to produce an effective CDCL-like method for ILP to reason at the constraint level [1]. However, due to a number of obstacles (see, e.g., Example 4 below), the results of such attempts were always orders of magnitude slower than the state-of-the-art commercial MIP/ILP solvers such as CPLEX or Gurobi, based on LP relaxations, simplex, and branch-and-cut (see, e.g., the "mip basics" at `www.gurobi.com`).

Typical attempts to generalize CDCL from SAT to ILP are in the following sense, where (possibly sub-indexed) letters a denote integer coefficients:

SAT		**ILP**	
clause	$l_1 \vee \ldots \vee l_n$	*linear constraint*	$a_1x_1 + \cdots + a_nx_n \leq a_0$
0/1 variable	x	*integer variable*	x
positive literal	x	*lower bound*	$a \leq x$
negative literal	$\overline{x}$	*upper bound*	$x \leq a$
propagation		*bound propagation*	
resolution inference		*cut* inference	

Example 1: By *bound* propagation, from the lower bound $1 \leq x$, the upper bound $y \leq 2$, and the constraint $x - 2y + 5z \leq 5$, we infer that $1 - 4 + 5z \leq 5$, so $5z \leq 8$, and hence $z \leq 8/5$, which is rounded, propagating a new bound $z \leq 1$. Note that any 1-variable constraint propagates a bound by itself, e.g., from $-7x \leq 3$ we have $-3 \leq 7x$, and hence $-3/7 \leq x$, which after rounding propagates the lower bound $0 \leq x$. □

Example 2: From $4x + 4y + 2z \leq 3$ and $-10x + y - z \leq 0$, by multiplying the former by 5 and the latter by 2 and adding them up, we obtain the *cut* $22y + 8z \leq 15$. Here the variable x is *eliminated*, which can always be achieved if in the two premises x has coefficients a and b such that $a \cdot b < 0$. The result $22y + 8z \leq 15$ can be *normalized* dividing by $gcd(22, 8) = 2$, giving $11y + 4z \leq 15/2$ and, by rounding, $11y + 4z \leq 7$. □

Example 3: Indeed, learning new constraints obtained by means of cuts can be advantageous over just learning clauses as in pure SMT. Consider the following set of constraints, where variables x, y, z, u are Boolean (0-1):

$$\begin{array}{lrrrrcl} C_1\colon & x & +y & -z & & \leq & 1 \\ C_2\colon & -2x & +3y & +z & -u & \leq & 1 \\ C_3\colon & 2x & -3y & +z & +u & \leq & 0. \end{array}$$

Below we depict the stack as it would grow upwards, in a CDCL algorithm: the first decision is the literal x (i.e., it is decided that x is true), then y, also by a decision, then z becomes true due

[1] Other SAT/SMT-related LP methods exist, but for rational arithmetic, such as [15, 13, 8], but in this paper we have decided to restrict ourselves to NP-complete problems.

to propagation with the constraint C_1, and then u becomes true due to propagation with the constraint C_2. After this, C_3 is a conflict (it becomes false in the current partial assignment):

C_3	conflict!
u	C_2
z	C_1
y	*decision*
x	*decision*
bound	reason

Stack ↑

Conflict analysis in SMT style would be as follows. The reason C_3 is conflicting due to the fact that it implies that y, z and u cannot be true all three at the same time, i.e., it implies the clause $\neg y \vee \neg z \vee \neg u$. The reason why C_2 propagated u is the clause $\neg y \vee \neg z \vee u$ implied by C_2. Resolution between both clauses yields the clause $\neg x \vee \neg y \vee \neg z$. Since it still has two literals of the current decision level, namely y and z, another iteration of conflict analysis is needed before a backjump becomes possible. However, a cut between C_2 and C_3 yields:

$$\begin{array}{rrrrcr} -2x & +3y & +z & -u & \leq & 1 \\ 2x & -3y & +z & +u & \leq & 0 \\ \hline & & 2z & & \leq & 1 \end{array}$$

which is equivalent to the clause $\neg z$, which is much stronger than $\neg x \vee \neg y \vee \neg z$. and directly allows one to backjump to decision level zero asserting $\neg z$. □

7 Constraint-learning 0-1 solvers

An extensive amount of work exists on Pseudo-Boolean solving (aka. 0-1 ILP), which considers Boolean variables, arbitrary linear constraints of the form $a_1x_1+\cdots+a_nx_n \leq a_0$ and optimizing an objective function $b_1y_1 + \cdots + b_mx_m$ where the coefficients a_i, b_j are integers (see [20] for all background and references). An important problem for extending CDCL to ILP is the following, which we will call the *rounding problem*. It already occurs in the 0-1 case, although it is much easier to solve here than for full ILP.

Example 4: Assume we have two constraints over 0-1 variables: $x + y + 2z \leq 2$ and $x + y - 2z \leq 0$. Now we take the decision that y is true, which due to $x + y + 2z \leq 2$ propagates $\neg z$ (by rounding $z \leq 1/2$). Then $x + y - 2z \leq 0$ becomes a *conflict*: it is false in the current partial assignment $A = \{ y, \neg z \}$, since x is at least 0.

Now let us attempt a straightforward generalization of the CDCL algorithm: since $\neg z$ is the topmost (last propagated) bound, a cut eliminating z between both constraints would be needed, generating the new constraint C which is $2x + 2y \leq 2$, or equivalently, $x + y \leq 1$. Then conflict analysis is over because there is only one bound in A at, or above, the last decision relevant for C, namely y. But unfortunately the new constraint $x + y \leq 1$ is not false in A, breaking (what should be) the invariant. Hence it does not propagate $\neg y$ and is *too weak to force a backjump*. This problem is due to the rounding that takes place when propagating z. □

One can always go the pure SMT way to solve the rounding problem whenever it appears. In the previous example, $x + y + 2z \leq 2$ entails the reason $\neg y \vee \neg z$ (propagating $\neg z$) and $x + y - 2z \leq 0$ is conflicting due to the reason $\neg y \vee z$. Resolution between both reasons yields $\neg y$, which can be used to backjump.

In fact some Pseudo-Boolean solvers *always* handle conflicts like this and only learn new clauses, even if for conflicts where no rounding problem appears. These are in fact SMT solvers. Other solvers such as Sat4J [5] have such an SMT setting but also other ones.

But one can be smarter. it is not difficult to see that the rounding problem does not appear if the variable that is eliminated in the cut inference has coefficient 1 or -1 in at least one of the two premises. Moreover, in the 0-1 case, for any propagation by a constraint C from a set of literals S, there exists a clause entailed by C that can also do this propagation from S. In fact, one can as well weaken C into an entailed constraint D that also does the same propagation and that is logically stronger than any clause, and such that the propagated variable also has coefficient 1 or -1 in D [10].

8 Constraint-learning solvers for full ILP

Now we consider the full ILP case, i.e., variables in $\mathbb{Z}$, and decisions and propagations in the stack are upper and lower bounds. For this situation, the rounding problem illustrated in Example 4 was solved in a very ingenious way by Jovanovic and de Moura in their *Cutsat* procedure [12]. In Cutsat, a decision can only make a variable *equal* to its current upper or lower bound, which permits, at each conflict caused by bound propagations with rounding, to compute *tightly propagating* constraints (again: where the propagated variable has coefficient 1 or -1). These tightly propagating constraints justify the same propagations *without* rounding, and allow one to do conflict analysis using tightly propagating constraints only. This on-the-fly computation of tightly propagating constraints during conflict analysis makes the learning scheme of [12] similar to the *all-decisions* SAT learning scheme. In this scheme, one does resolutions until reaching a clause built from decisions only. This is well known to perform significantly worse than 1UIP, search-wise. Experiments with the Cutsat implementation of [12] show that, not very unexpectedly, this finding unfortunately carries over to Cutsat (see [17]).

IntSat [17] is a different method for ILP. Unlike Cutsat, it admits arbitrary new bounds as decisions and guides the search exactly as with the 1UIP approach in CDCL-based SAT solving, while still overcoming the rounding problem. The key ideas behind IntSat are as follows.

Given a partial assignment A, a set (stack) of bounds, each time a constraint C and a set of bounds R with $R \subseteq A$ propagate a new bound B, this bound is pushed onto A, associating to B not only its *reason constraint* C but also its *reason set* R. Conflict analysis and cuts performed are both guided by successive refinements of a so-called *Conflicting Set* of bounds $CS \subseteq A$ that is infeasible along with the current set of constraints. After each conflict, always a backjump takes place and a new constraint is learned. In the following IntSat algorithm, A is seen as an (initially empty) stack of bounds:

The basic IntSat algorithm:

1. *Propagate*: while possible and no conflict appears, if C and R propagate some fresh bound B, for some constraint C and set of bounds R with $R \subseteq A$, then push B onto A, associating to B the *reason constraint* C and the *reason set* R.

2. **if** there is no conflict
   if all variables are defined in A, output "solution A" and halt.
   else *Decide*: push some fresh bound B onto A, marked as a *decision*, and go to 1.

3. If A contains no decisions, output "infeasible" and halt.

4. Use data structures C, a constraint, and CS, the *Conflicting Set* of bounds. Initially, C is any conflict and CS is the subset of bounds of A causing the falsehood of C.

   - *Conflict analysis*: Invariants: $CS \subseteq A$ and if S is the current set of constraints, then $S \cup CS$ is infeasible (has no solution).

     Repeat the following three steps:

     - If B is the bound in CS that is topmost in A, and R is the reason set of B, then let CS be $(CS \setminus \{B\}) \cup R$.
     - If a cut eliminating B's variable exists between C and B's reason constraint then replace C by that cut.
     - *Early Backjump*: If for some maximal $k \in \mathbb{N}$, after popping k bounds the last one being a decision, C propagates some new bound in the resulting A, then pop k bounds, learn C as a new constraint, and go to 1.

     until CS contains a single bound B_{top} that is, or is above, A's topmost decision.

   - *Backjump*: Pop bounds from A until either there are no decisions in A or, for some B in CS with $B \neq B_{top}$, there are no decisions above B in A. Then push $\overline{B_{top}}$ with associated reason constraint C and reason set $CS \setminus \{B_{top}\}$.
   - *Learn*: add the final C as a new constraint, and go to 1.

Note that in the second step of conflict analysis indeed sometimes no cut eliminating B's variable exists between C and B's reason constraint; this can be because that variable does not occur in C, or it occurs with the same sign.

Example 5: Assume there are two constraints $x + y + 2z \leq 2$ and $x + y - 2z \leq 0$ and we take the decision $0 \leq x$, which propagates nothing, and later on another decision $1 \leq y$, which due to $x + y + 2z \leq 2$ propagates $z \leq 0$, which is pushed with associated reason constraint $x + y + 2z \leq 2$ and reason set $\{\ 0 \leq x,\ 1 \leq y\ \}$. *Conflict analysis*: Initially, $x + y - 2z \leq 0$ is the conflict C and CS is the set of bounds $\{\ 0 \leq x,\ 1 \leq y,\ z \leq 0\ \} \subseteq A$ causing the falsehood C. In the first iteration, in CS we replace $z \leq 0$ by its reason set $\{\ 0 \leq x,\ 1 \leq y\ \}$. The resulting CS is $\{\ 0 \leq x,\ 1 \leq y\ \}$. A cut eliminating z exists (see Example 7) and C becomes $x + y \leq 1$. Then conflict analysis is over because the CS contains exactly one bound B_{top}, which is $1 \leq y$, at or above A's topmost decision. *Backjump*: We pop bounds until for some B in CS with $B \neq B_{top}$, there are no decisions above B in A, in this case, until there are no decisions above $0 \leq x$ in A, and then push $\overline{B_{top}}$, which is $y \leq 0$, with reason set $\{\ 0 \leq x\ \}$, and with reason constraint $x + y \leq 1$. Note that this reason constraint is not a "good" reason, i.e., it does not propagate

$y \leq 0$, but still $y \leq 0$ is a valid consequence of the set of constraints together with its reason set $\{\ 0 \leq x\ \}$. *Learn*: The final C, which is $x + y \leq 1$, is learned. □

Example 6: Assume there are three constraints

$$\begin{array}{lrrrrr} C_0: & x & -3y & -3z & \leq & 1 \\ C_1: & -2x & +3y & +2z & \leq & -2 \\ C_2: & 3x & -3y & +2z & \leq & -1 \end{array} \qquad \text{and initial bounds:} \qquad \begin{array}{ll} -2 \leq z & z \leq 2 \\ 1 \leq y & y \leq 4 \\ -2 \leq x & x \leq 3. \end{array}$$

Then the following sequence of decisions and propagations can take place on the stack:

$2 \leq y$	$\{\ 1 \leq x,\ z \leq -2\ \}$	C_0: $x - 3y - 3z \leq 1$
$x \leq 1$	$\{\ y \leq 2,\ z \leq -2\ \}$	C_0: $x - 3y - 3z \leq 1$
$z \leq -2$	*decision*	
$z \leq -1$	$\{\ x \leq 2,\ 1 \leq y\ \}$	C_1: $-2x + 3y + 2z \leq -2$
$x \leq 2$	*decision*	
$z \leq 0$	$\{\ x \leq 3,\ 1 \leq y\ \}$	C_1: $-2x + 3y + 2z \leq -2$
$y \leq 2$	$\{\ x \leq 3,\ -2 \leq z\ \}$	C_1: $-2x + 3y + 2z \leq -2$
$1 \leq x$	$\{\ 1 \leq y,\ -2 \leq z\ \}$	C_1: $-2x + 3y + 2z \leq -2$
$-2 \leq z$	*initial*	
...	...	
bound	reason set	reason constraint

In this state, C_1 is a conflict. with initial $CS = \{\ -2 \leq z,\quad x \leq 1,\quad 2 \leq y\ \}$, and conflict analysis proceeds as follows. Replacing $2 \leq y$ by its reason set, we get a new CS which is $\{\ -2 \leq z,\ 1 \leq x,\ z \leq -2,\ x \leq 1\ \}$. A cut eliminating y between C_1 and C_0 exists, and gives C_3: $-x - z \leq -1$. At his point, already an Early backjump exists, due to $z \leq -1$. So one can backjump to decision level 1, push there $2 \leq x$ and learn C_3. The new bound triggers two more propagations and we get:

$2 \leq y$	$\{\ 2 \leq x,\ z \leq -1\ \}$	C_0: $x - 3y - 3z \leq 1$
$-1 \leq z$	$\{\ x \leq 2\ \}$	C_3: $-x - z \leq -1$
$2 \leq x$	$\{\ z \leq -1\ \}$	C_3: $-x - z \leq -1$
$z \leq -1$	$\{\ x \leq 2,\ 1 \leq y\ \}$	C_1: $-2x + 3y + 2z \leq -2$
$x \leq 2$	*decision*	
$z \leq 0$	$\{\ x \leq 3,\ 1 \leq y\ \}$	C_1: $-2x + 3y + 2z \leq -2$
$y \leq 2$	$\{\ x \leq 3,\ -2 \leq z\ \}$	C_1: $-2x + 3y + 2z \leq -2$
$1 \leq x$	$\{\ 1 \leq y,\ -2 \leq z\ \}$	C_1: $-2x + 3y + 2z \leq -2$
$-2 \leq z$	*initial*	

Now again C_1 is conflicting, with initial $CS = \{\ x \leq 2,\ -1 \leq z,\ 2 \leq y\ \}$, and conflict analysis needs four iterations:

1. Replacing $2 \leq y$ by its reason set we get $CS = \{\ x \leq 2,\ \ z \leq -1,\ \ 2 \leq x,\ \ -1 \leq z\ \}$. The cut$(C_0, C_1)$ gives C: $-x - z \leq -1$ as before.
2. Replace $-1 \leq z$ by its reason set. We get $CS = \{\ x \leq 2,\ \ z \leq -1,\ \ 2 \leq x\ \}$. No cut is made (since z is negative in both C and C_3).
3. Replace $2 \leq x$. We get $CS = \{\ x \leq 2,\ \ z \leq -1\ \}$; no cut exists.
4. Replacing $z \leq -1$, we obtain $CS = \{\ 1 \leq y,\ \ x \leq 2\ \}$. Now a cut exists and gives $-4x + 3y \leq -4$. At this point one can do an early backjump adding $2 \leq x$ at decision level zero. But Conflict analysis is also finished (only one bound of current decision level in CS), and we can backjump to decision level 0 adding $x \not\leq 2$, i.e., $3 \leq x$. This latter option is stronger, and hence probably preferrable.

After one further propagation ($-1 \leq z$), the procedure returns "infeasible" since the conflict C_2 appears at decision level 0. □

The following theorem holds even if no cuts are performed and no new constraints are learned (although practical performance depends crucially on these). Its proof follows essentially the same scheme as our termination, soundness and completeness results for SAT and SAT Modulo Theories (SMT) [18].

Theorem 8.1. *The basic IntSat algorithm, when given as input a finite set of constraints S including for each variable x_i a lower bound $lb_i \leq x_i$ and an upper bound $x_i \leq ub_i$, always terminates, finding a solution if, and only if, there exists one, and returning "infeasible" if, and only if, S is infeasible.*

Optimization. In this framework, optimization is incorporated easily, since, unlike what happens in SAT, linear constraints are first-class citizens (i.e., belong to the core language). For finding a solution that minimizes a linear expression $a_1x_1 + \ldots + a_nx_n$ (or maximizes it with opposite signs), first an arbitrary solution A is found and then, at each new solution A one adds a stronger new constraint $a_1x_1 + \ldots + a_nx_n \leq a_1 \cdot A(x_1) + \ldots + a_n \cdot A(x_n) - 1$, which is conflicting, and does conflict analysis on it, etc., until infeasible. These successively stronger constraints are indeed very effective for pruning.

Initial upper and lower bounds. The assumption that each variable has initial lower and upper bounds, (or, equivalently, initial constraints propagating such bounds) is common in practical applications. However, some problems do have unbounded variables. In theory, any ILP can be converted into an equivalent fully bounded one [21], but these bounds are too large to be useful in practice. One solution is to have a *z-bound*: to introduce a single fresh auxiliary variable z, with lower bound $0 \leq z$, and for each variable x without lower bound add the constraint $-z \leq x$, and similarly, if it has no upper bound, add $x \leq z$. Then one can re-run the IntSat procedure with successively larger upper bounds $z \leq ub$ for z, thus guaranteeing completeness for finding (optimal) solutions. Further practical solutions are subject of current work, also for handling the well-known fact that, with unbounded variables and without z-bound, bound propagation may not terminate in unfeasible problems: consider, e.g., C_1: $x - y \leq 0$ and C_2: $-x + y + 1 \leq 0$ and the bound $0 \leq x$, which makes C_1 propagate $0 \leq y$; then C_2 propagates $1 \leq x$, and so on.

Implementation ideas. Our prototype IntSat implementation currently consists of 1400 lines of simple C++ code with standard STL data structures. For instance, a constraint is an STL vector of monomials (pairs of two `int`s), sorted by variable number, plus some additional information (independent term, activity). Coefficients are never larger than 2^{30}, and cuts producing any coefficient larger than 2^{30} are simply not performed, which is a straightforward way of guaranteeing that no overflow occurs if bound propagation, cuts, normalization, etc., are done storing intermediate results in 64-bit integers. Two arrays store for each variable x_i the positions in the stack of its current (strongest) upper bound and lower bound, respectively. In the stack itself bounds are linked, pointing to the position in the stack of the previous bound of the same type (lower or upper) for that variable. When pushing or popping bounds, these properties are maintained in constant time. Affordably efficient bound propagation is crucial for performance. In our current implementation, this done with occurs lists and counter-based filters preventing many cache misses due to useless visits to the actual constraints.

Performance. In [17] several experiments are described with our prototype IntSat implementation running on one core, compared with the latest 4-core versions of the commercial solvers CPLEX and Gurobi. The technology behind the latter two solvers is extremely mature, after decades of improvements. According to [7], between 1991 and 2012 they have seen a 475000x speedup from algorithmic improvements (i.e., not counting the 2000x from hardware), from a combination of techniques, such as specialized cuts, heuristics and *presolve* methods. In [17], random optimization instances are considered as well as the well-known Mixed Integer Problem Library (MIPLIB) ones. In many, but of course not all cases, IntSat is fastest in finding the first solution, i.e., in proving feasibility, as well as in finding good solutions and (near) optimal ones.

9 Conclusions and further work

A large amount of further theoretical and practical ideas around 0-1 ILP and IntSat arise to be explored. From the implementation point of view, this includes, e.g., special treatments for binary variables and for specific types of constraints, decision heuristics (tailored towards solutions or to infeasibility), restarts and cleanups, early backjump and conflict analysis strategies, reason set strengthening, or pre– and in-processing. It also seems that periodically optimizing the LP relaxation during the IntSat process can be useful at least for lower bounding, but possibly also for guiding the decision heuristics. It could be used for handling MIPs, i.e., where some variables may take non-integer solutions, by deciding on the integer variables as it is done now, and having a simplex optimize the remaining ones. Combining ideas from both worlds may worth exploring, due to our feeling that CDCL-like techniques may be superior on problem aspects with a combinatorial flavor, whereas traditional ones could be better for the essentially numerical ones. In any case, it seems unlikely that for ILP or MIP solving one single technique can dominate the others; the best solvers will probably continue combining different methods from a large toolbox.

Acknowledgments

Supported by the Spanish grants TIN2013-45732-C4-3-P (MINECO) and TIN2010-21062-C02-01 (MEC/MICINN).

References

[1] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A New Look at BDDs for Pseudo-Boolean Constraints. *J. Artif. Intell. Res. (JAIR)*, 45:443–480, 2012.

[2] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J. Stuckey. To Encode or to Propagate? The Best Choice for Each Constraint in SAT. In *19th International Conference on Principles and Practice of Constraint Programming*, CP'13, pages 97–106. Springer Berlin Heidelberg, 2013.

[3] Ignasi Abío and Peter J. Stuckey. Conflict-Directed Lazy Decomposition. In *18th International Conference on Principles and Practice of Constraint Programming*, CP'12, pages 70–85, 2012.

[4] Ignasi Abío and Peter J. Stuckey. Encoding linear constraints into SAT. In *Principles and Practice of Constraint Programming, CP, LNCS 8656*, pages 75–91, 2014.

[5] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 7(2-3):59–6, 2010.

[6] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[7] Bob Bixby. Presentation: 1000X MIP Tricks, 12 June 2012, Bill Cunninghams 65th, 2012.

[8] Scott Cotton. Natural domain smt: A preliminary assessment. In *Formal Modeling and Analysis of Timed Systems - 8th International Conference, FORMATS 2010*, pages 77–91. Springer LNCS 6246, 2010.

[9] L. de Moura and H. Rueß. Lemmas on Demand for Satisfiability Solvers. In *5th International Conference on Theory and Applications of Satisfiability Testing, SAT '02*, pages 244–251, 2002.

[10] Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Procs 18th National Conf on Artificial Intelligence*, pages 635–640, 2002.

[11] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research*. McGraw-Hill, 9th edition, 2010.

[12] Dejan Jovanovic and Leonardo Mendonça de Moura. Cutting to the chase - solving linear integer arithmetic. *J. Autom. Reasoning*, 51(1):79–108, 2013.

[13] Konstantin Korovin and Andrei Voronkov. Solving systems of linear inequalities by bound propagation. In *CADE-23 - 23rd Int. Conf. on Automated Deduction*, pages 369–383, 2011.

[14] J. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[15] Kenneth L. McMillan, Andreas Kuehlmann, and Mooly Sagiv. Generalizing dpll to richer logics. In *Computer Aided Verification, 21st International Conference, CAV*, pages 462–476. Springer LNCS 5643, 2009.

[16] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[17] Robert Nieuwenhuis. The IntSat method for integer linear programming. In *Principles and Practice of Constraint Programming, 20th International Conference, CP 2014, LNCS 8656*, pages 574–589.

[18] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM, JACM*, 53(6):937–977, 2006.

[19] O. Ohrimenko, P. Stuckey, and M. Codish. Propagation = lazy clause generation. In C. Bessiere, editor, *Principles and Practice of Constraint Programming, 13th International Conference, CP '07*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007.

[20] Olivier Roussel and Vasco M. Manquinho. Pseudo-boolean and cardinality constraints. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in AI and Applications*, pages 695–733. IOS Press, 2009.

[21] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Chichester, 1986.