The IntSat method for Integer Linear Programming

Robert Nieuwenhuis

Barcelogic.com and Technical University of Catalonia (UPC), Barcelona

Abstract. Conflict-Driven Clause-Learning (CDCL) SAT solvers can automatically solve very large real-world problems. To go beyond, and in particular in order to solve *and optimize* problems involving linear arithmetic constraints, here we introduce IntSat, a generalization of CDCL to Integer Linear Programming (ILP). Our simple 1400-line C++ prototype IntSat implementation already shows some competitiveness with commercial solvers such as CPLEX or Gurobi. Here we describe this new IntSat ILP solving method, show how it can be implemented efficiently, and discuss a large list of possible enhancements and extensions.

1 Introduction

Conflict-Driven Clause-Learning (CDCL) propositional SAT solving technology can automatically solve hard real-world industrial and scientific problem instances involving large numbers of binary variables and constraints (i.e., clauses). 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:

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 goto step 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 $\overline{l} \in A$.
     If l is the literal of C whose negation is topmost in A, and D is the reason clause of $\overline{l}$, 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 $\overline{l}$ 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 $D \vee \overline{l}$. 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 *forgetting* (removing) the least useful learned clauses, and using clause simplification and other *inprocessing* methods (see, e.g., [16, 4] for more details and further references).

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. However, due to a number of obstacles (see, e.g., Example 3 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 subindexed) letters a denote integer coefficients:

SAT		**ILP**	
clause	$l_1 \vee \ldots \vee l_n$	*linear constraint*	$a_1 x_1 + \cdots + a_n x_n \preceq a_0$
0/1 variable	x	*integer variable*	x
positive literal	x	*lower bound*	$a \preceq x$
negative literal	$\overline{x}$	*upper bound*	$x \preceq a$
propagation		*bound propagation*	
resolution inference		*cut* inference	

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

Example 2. From $4x + 4y + 2z \preceq 3$ and $-10x + y - z \preceq 0$, by multiplying the former by 5 and the latter by 2 and adding them up, we obtain the *cut* $22y + 8z \preceq 15$. 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 $22y + 8z \preceq 15$ can be *normalized* dividing by $gcd(22, 8) = 2$, giving $11y + 4z \preceq 15/2$ and, by rounding, $11y + 4z \preceq 7$. □

Example 3. An important problem for extending CDCL to ILP is the following, which we will call the *rounding problem*. Assume we have the two constraints $x + y + 2z \preceq 2$ and $x + y - 2z \preceq 0$ and we take the decision $0 \preceq x$, which propagates nothing, and later on another decision $1 \preceq y$, which due to $x + y + 2z \preceq 2$ propagates $z \preceq 0$ (since $1 + 2z \preceq 2$, and hence $2z \preceq 1$ and by rounding $z \preceq 1/2$ we get $z \preceq 0$). Then $x + y - 2z \preceq 0$ becomes a *conflict*: it is false in the current partial assignment $A = \{\, 0 \preceq x,\ 1 \preceq y,\ z \preceq 0 \,\}$.

Now let us attempt a straightforward generalization of the CDCL algorithm: since $z \preceq 0$ is the topmost (last propagated) bound, a cut inference eliminating z between both constraints would be needed, generating the new constraint C which is $2x + 2y \preceq 2$, or equivalently, $x + y \preceq 1$. Then conflict analysis is over because there is only one bound in A at, or above, the last decision relevant for C, namely $1 \preceq y$. But unfortunately the new constraint $x + y \preceq 1$ is not false in A, breaking (what should be) the invariant. Hence it does not propagate the negation of $1 \preceq y$ (which is $y \preceq 0$) and is *too weak to force a backjump*. This problem is due to the rounding that takes place when propagating z. □

The rounding problem illustrated in Example 3 was solved in a very ingenious way by Jovanovic and de Moura in their *cutsat* procedure [14], where 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 that justify the same propagations *without* rounding, and to do conflict analysis using tightly propagating constraints only. This makes the learning scheme of [14] similar to the SAT one of doing resolutions until reaching a clause built from decisions only, which is well known to perform significantly worse than 1UIP.

Here we introduce IntSat, a new completely different method for ILP. It admits arbitrary new bounds as decisions and guides the search exactly as with the 1UIP approach in CDCL-based SAT solving, and still it overcomes 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.

This paper is structured as follows. The basic IntSat procedure is introduced in Section 2, its termination, correctness and completeness stated, and extensions are given for, e.g., optimization. In Section 3 we give some details about the efficient implementability of IntSat, and further work on IntSat is discussed in Section 4. Section 5 provides experimental results and in Section 6 we describe related work and conclude.

2 The basic IntSat procedure

In a first basic version of IntSat we deal with integer coefficients only and decide the existence of integer solutions, i.e., feasibility problems only, and no optimization yet (extensions are handled later on). Let X be a finite set of *variables* $\{x_1 \dots x_n\}$. An *(integer linear) constraint* over X is an expression of the form $a_1x_1 + \cdots + a_nx_n \preceq a_0$ where, for all i in $0 \dots n$, the *coefficients* a_i are integers. Below, variables are always denoted by (possibly sub-indexed or primed) lowercase x, y, z and coefficients by a, b, c, respectively. An *Integer Program (IP)* over X is a set S of integer linear constraints over X. A *solution* for an IP S over X is a function $sol\colon X \to \mathbb{Z}$ that *satisfies* every constraint $a_1x_1 + \cdots + a_nx_n \preceq a_0$ in S, that is, $a_1 \cdot sol(x_1) + \cdots + a_n \cdot sol(x_n) \preceq_{\mathbb{Z}} a_0$. A *bound* is a one-variable constraint $a_1x \preceq a_0$. Any bound can equivalently either be written as a *lower bound* $a \preceq x$ or as an *upper bound* $x \preceq a$. For a constraint $a_1x_1 + \ldots + a_nx_n \preceq a_0$, each a_ix_i is called a *monomial* of it, and the monomial is positive (negative) if a_i is.

Bound propagation: Let C be a constraint of the form $P + ax \preceq a_0$, where P is a sum of positive monomials $\{ b_1y_1, \dots, b_py_p \}$ and negative monomials $\{ c_1z_1, \dots, c_qz_q \}$. Let R be a set of bounds $\{ lb_1 \preceq y_1, \dots, lb_p \preceq y_p,\ \ z_1 \preceq ub_1, \dots, z_q \preceq ub_q \}$. Let E denote the expression $a_0 - b_1lb_1 - \ldots - b_plb_p - c_1ub_1 - \ldots - c_qub_q$. If $a < 0$ then then C and R *propagate* $\lceil E/a \rceil \preceq x$. If $a > 0$ then then C and R *propagate* $x \preceq \lfloor E/a \rfloor$.

Cuts and constraint normalization: In what follows we assume all constraints to be eagerly *normalized:* any constraint $a_1x_1 + \ldots + a_nx_n \preceq a_0$ with $d = gcd(a_1, \dots, a_n) > 1$ is eagerly replaced by $a_1/d\ \ x_1 + \ldots + a_n/d\ \ x_n \preceq \lfloor a_0/d \rfloor$. From constraints

$a_1x_1 + \cdots + a_nx_n \preceq a_0$ and $b_1x_1 + \cdots + b_nx_n \preceq b_0$, and natural numbers c and d, a new constraint $c_1x_1 + \cdots + c_nx_n \preceq c_0$ called a *cut* can be obtained where $c_i = ca_i + db_i$ for i in $0 \ldots n$. If some $c_i = 0$ then we say that x_i is *eliminated* in this cut. Note that if $a_ib_i < 0$, then one can always choose c and d such that x_i is eliminated. See [13, 7, 23] for further discussions and references about Chvátal-Gomory cuts and their applications to ILP.

Let A be a set of bounds. A variable x is *defined to a* in A if $\{a \preceq x, x \preceq a\} \subseteq A$ for some a. We call two bounds $a \preceq x$ and $x \preceq a'$ *contradictory* if $a > a'$. Note that if all variables of X are defined and there are no contradictory bounds in A then A can be seen as a total assignment $A\colon X \to \mathbb{Z}$. A bound $a \preceq x$ is *new* in A if there is no $a' \preceq x$ in A with $a' \geq a$. Similarly, $x \preceq a$ is new if there is no $x \preceq a'$ with $a' \leq a$. A bound is *fresh* in A if it is new in A and contradictory with no bound in A.

False constraint (conflict) in A: If C is a constraint $a_1x_1 + \ldots + a_nx_n \preceq a_0$ with positive monomials $\{\ b_1y_1, \ldots, b_py_p\ \}$, negative monomials $\{\ c_1z_1, \ldots, c_qz_q\ \}$ and there is a subset of bounds $\{\ lb_1 \preceq y_1, \ldots, lb_p \preceq y_p,\ \ z_1 \preceq ub_1, \ldots, z_q \preceq ub_q\ \} \subseteq A$ with $b_1lb_1 + \ldots + b_plb_p + c_1ub_1 + \ldots + c_qub_q > a_0$ then C is *false* or a *conflict* in A.

The basic IntSat algorithm: In the following IntSat algorithm, A is seen as an (initially empty) stack of bounds:

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, 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 2nd 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 4. We apply IntSat to Example 3: 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 3) 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 5. Consider the initial constraints

$$\begin{array}{lrcl} C_0: & x - 3y - 3z & \leq & 1 \\ C_1: & -2x + 3y + 2z & \leq & -2 \\ C_2: & 3x - 3y + 2z & \leq & -1 \end{array}$$

Below we depict the stack with some initial bounds after doing their propagations and taking and propagating two decisions:

$2 \leq y$	$\{\ 1 \leq x,\ z \leq -2\ \}$	C_0
$x \leq 1$	$\{\ y \leq 2,\ z \leq -2\ \}$	C_0
$z \leq -2$	*decision*	
$z \leq -1$	$\{\ x \leq 2,\ 1 \leq y\ \}$	C_1
$x \leq 2$	*decision*	
$z \leq 0$	$\{\ x \leq 3,\ 1 \leq y\ \}$	C_1
$y \leq 2$	$\{\ x \leq 3,\ -2 \leq z\ \}$	C_1
$1 \leq x$	$\{\ 1 \leq y,\ -2 \leq z\ \}$	C_1
$z \leq 2$	*initial*	
$-2 \leq z$	*initial*	
$y \leq 4$	*initial*	
$1 \leq y$	*initial*	
$x \leq 3$	*initial*	
$-2 \leq x$	*initial*	
bound	*reason set*	*reason constraint*

Now C_1 is a conflict. The initial CS is $\{\ -2 \leq z,\ x \leq 1,\ 2 \leq y\ \}$, with two bounds above the last decision. In the first conflict analysis step, we replace $2 \leq y$ by its reason set $\{\ 1 \leq x,\ z \leq -2\ \}$ obtaining the new CS $\{\ -2 \leq z,\ 1 \leq x,\ z \leq -2,\ x \leq 1\ \}$ which still has two bounds at or above the last decision. Now a cut eliminating y is attempted between

the initial C, which is C_1, and the reason constraint of $2 \leq y$, which is C_0. Here this cut exists, with $c = d = 1$, and we obtain and learn the new constraint $C_3 : \ -x - z \leq -1$. It allows us to perform an early backjump to before the second decision, since there it propagates $2 \leq x$ with reason set $\{\ z \leq -1\ \}$ and reason constraint C_3. Then, after two more propagations, we obtain

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

and again C_1 is a conflict, with the initial CS being $\{\ x \leq 2,\ -1 \leq z,\ 2 \leq y\ \}$. After the first conflict analysis step (replacing $2 \leq y$) the CS becomes $\{\ x \leq 2,\ z \leq -1,\ 2 \leq x,\ -1 \leq z\ \}$. As before, the cut eliminates y, between C_1 and C_0 (the initial C and the reason constraint of $2 \leq y$), obtaining $-x - z \leq -1$. After the following step (replacing $-1 \leq z$), the CS becomes $\{\ x \leq 2,\ z \leq -1,\ 2 \leq x\ \}$. The C does not change because no cut eliminating z exists with C_3. In the next step (replacing $2 \leq x$), the CS becomes $\{\ x \leq 2,\ z \leq -1\ \}$. Again no cut eliminating z exists with C_3. In another step (replacing $z \leq -1$), the CS becomes $\{\ 1 \leq y,\ x \leq 2\ \}$. Since there is only one bound at or after the last decision, we backjump, in this case to before the first decision, and add there the negation of $x \leq 2$, which is $3 \leq x$.

The result of the cut on C with C_1 eliminating z gives us $-4x + 3y \leq -4$. The backjump with this cut (C_4) can also take us to before the first decision, but propagating $2 \leq x$. Since this is weaker than the bound $3 \leq x$ obtained from the CS, here we choose the CS one. After one further propagation, the procedure returns "infeasible" since the conflict C_2 appears and there are no decisions in the stack:

$-1 \leq z$	$\{\ 3 \leq x,\ y \leq 2\ \}$	C_0
$3 \leq x$	$\{\ 1 \leq y\ \}$	C_4
$z \leq 0$	$\{\ x \leq 3,\ 1 \leq y\ \}$	C_1
$y \leq 2$	$\{\ x \leq 3,\ -2 \leq z\ \}$	C_1
$1 \leq x$	$\{\ 1 \leq y,\ -2 \leq z\ \}$	C_1
$z \leq 2$	*initial*	
...	...	
$-2 \leq x$	*initial*	

Theorem 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.*

The previous 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) [20]. For termination (from which soundness and completeness are not hard to establish), we define a well-founded ordering $\succ$ on the states of the stack A, as follows. For a given A, the number of possible values a variable x_i can still take is $v_i(A) = ub_i - lb_i + 1$, where $lb_i \leq x_i$ and $x_i \leq ub_i$ are its topmost lower and upper bounds in A, and the total number of values for all n variables is $v(A) = v_1(A) + \ldots v_n(A)$. Let A_i, for $i \geq 1$ denote the bottom part of A, below (and without) the i-th decision. We define a stack A to be larger (i.e., less advanced, search-wise) than a stack A', written $A \succ A'$, if $\langle v(A_1), \ldots, v(A_m)\rangle >_{lex} \langle v(A'_1), \ldots, v(A'_m)\rangle$ where m is the maximal number of decisions the stack can contain, at most $n \cdot v(A)$ for the initial A. It is easy to see that this lexicographic ordering $\succ$ is well-founded and that all steps of the algorithm either halt it or transform A into an A' with $A \succ A'$.

More general constraints: It is obvious that a constraint $a_1x_1 + \ldots + a_nx_n \geq a_0$ can be expressed as $-a_1x_1 - \ldots - a_nx_n \leq -a_0$, that $a_1x_1 + \ldots + a_nx_n = a_0$ can be replaced by the two constraints $a_1x_1 + \ldots + a_nx_n \leq a_0$ and $a_1x_1 + \ldots + a_nx_n \geq a_0$, and that rational non-integer coefficients a/b can be removed by multiplying the constraint by b.

Optimization is also possible in a standard way, 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 $-a_1x_1 - \ldots - a_nx_n$), in our current implementation this is done in a completely straightforward way: first an arbitrary solution A is found and then, each time a new solution A is found, it is attempted to improve it by re-running with the additional constraint $a_1x_1 + \ldots + a_nx_n \leq a_0$ where a_0 is $a_1A(x_1) + \ldots + a_nA(x_n) - 1$. This is done until the problem becomes infeasible. Bound propagations from these successively stronger constraints are indeed very effective for pruning (bounding) the resulting branch-and-bound search.

Handling unbounded variables. Up to now we have assumed that for each variable there is an initial lower bound and an upper bound, or, equivalently, initial constraints propagating such bounds. Although this is common in practical applications, some problems do have unbounded variables. In theory, any ILP can be converted into an equivalent fully bounded one [23], but these bounds are too large to be useful in practice. One solution is to introduce a 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 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.

3 Implementation

Here we describe some details of our current prototype IntSat implementation. It currently consists of 1400 lines of simple C++ code that make heavy use of standard STL data structures (this source code can be downloaded from [19]). For instance, a constraint is an STL vector of monomials (pairs of two `int`s: the variable number and the coefficient), 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 using 64-bit integers for intermediate results. During conflict analysis, the CS is implemented simply as an STL set of `int`s, the heights in the stack of the bounds in the CS. A very large source of inefficiency of conflict analysis is our current implementation of *Early Backjump*s, which, after each cut giving a new C, naively checks, at all (frequently thousands of) prefixes of the stack below a decision, whether C propagates any new bound at that prefix.

The current assignment There is an array, the *Bounds Array*, indexed by variable number, that can return in constant time the current upper and lower bounds for that variable. It always stores, for each variable x_i, the positions pl_i and pu_i in the stack of its current (strongest) upper bound and lower bound, respectively, with $pl_i = 0$ ($pu_i = 0$) if x_i has no current lower (upper) bound. The stack itself is another array containing at each position three data fields: a bound, a natural number *pos*, and an *info* field containing, among other information, (pointers to) the reason set and the reason constraint. The value *pos* is always the position in the stack of the previous bound of the same type (lower or upper) for this variable, with *pos* = 0 for initial bounds. When pushing or popping bounds, these properties are easy to maintain in constant time.

Example of bounds array and stack:

	Height in stack of current bound	
	lower:	**upper:**
x_1	1	2
x_2	0	0
	⋮	⋮
x_7	40	31
	⋮	⋮

	⋮		
40	$5 \le x_7$	23	*info*
	⋮		
31	$x_7 \le 6$	14	*info*
	⋮		
23	$2 \le x_7$	13	*info*
	⋮		
14	$x_7 \le 9$	0	*info*
13	$0 \le x_7$	0	*info*
	⋮		
2	$x_1 \le 8$	0	*info*
1	$0 \le x_1$	0	*info*

Bound propagation using filters: Affordably efficient bound propagation is crucial for performance. In our current implementation, for each variable x, there are two *occurs*

lists. The *positive* occurs list for x contains all pairs (I_C, a) s.t. C is a linear constraint where x occurs with positive coefficient a, and the *negative* one contains the same for occurrences with a negative coefficient a. Here I_C is an index to the *constraint header* of C in an array of constraint headers. Each constraint header contains an integer F_C called a *filter*, and (a pointer to) the constraint C itself. The filter F_C is maintained cheaply, and one can guarantee that *C does not propagate anything as long as $F_C \leq 0$*, thus avoiding many useless (cache-) expensive visits to the actual constraint C. This is done as follows.

Let C be a constraint of the form $a_1x_1 + \cdots + a_nx_n \preceq a_0$. Let $lb_i \preceq x_i$ and $x_i \preceq ub_i$ be the current lower and upper bounds (if any) for x_i. Each monomial a_ix_i in C can have a *minimal value* m_i, which is $a_i \cdot lb_i$ if $a_i \geq 0$, and $a_i \cdot ub_i$ otherwise. Here m_i is undefined if there is no such bound lb_i (or ub_i). Initially, if some m_i is undefined, then F_C is set to a special value $\perp$, and otherwise to $-a_0 + m_1 + \cdots + m_n + max_i\{\ |a_i(ub_i - lb_i)|\ \}$. In the latter case, F_C is said to be *precise*: the constraint C propagates *if, and only if*, $\perp \neq F_C > 0$. At all time points, $F_C = \perp$ or F_C is an upper approximation of the precise one, so C can only propagate (or be false) if $F_C > 0$.

To preserve this property, these filters need to be updated when new bounds are pushed onto the stack (and each update needs to be undone when popped, for which other data structures exist). Assume a new lower bound $k \preceq x$ is pushed onto the stack. Let the previous lower bound for x (if any) be $k' \preceq x$. For each pair (I_C, a) in the positive occurs list of x, using I_C we access the F_C and increase it by $|a(k - k')|$. If there was no previous lower bound, then F_C was $\perp$ and is now set to 1. If F_C becomes positive, the constraint C is visited because it may propagate some new bound. After each time a constraint C is visited, F_C is set to its precise value. If a new upper bound $x \preceq k$ is pushed on the stack, exactly the same is done, where $x \preceq k'$ is the previous upper bound for x (if any), and using the negative occurs list.

4 Further work

Both from a theoretical as a practical point of view, a large amount of further ideas around IntSat arise to be explored. From the implementation point of view, aspects such as special treatments for binary variables and for specific types of constraints should be worked out. Our current implementation in fact mimics several ideas from CDCL SAT solving without having tested them thoroughly.

Decision heuristics. For instance, our current heuristics for selecting the variable of the next decision bound are based on recent activity: the variable with the highest activity score is picked (for this there is a priority queue). The activity score of a variable x is increased each time a bound containing x appears in the CS during conflict analysis, and to reward *recent* activity the amount of increment grows in time. Once a variable is picked, one has to decide the actual decision bound: whether it is lower or upper, and how to divide the interval between the current lower and upper bounds. Another idea is to try to mimic the last-phase polarity heuristic from SAT [22], which would translate into picking some recent upper/lower bound and value for the selected variable.

Restarts and cleanups. Something similar happens with the periodic *restarts* that SAT solvers apply. We currently follow a rather conservative restart policy based on increasing intervals based on the number of conflicts. Another basically non-tested aspect is the *cleanup* policy for the constraint database; at each cleanup, we remove all non-initial constraints with more than two monomials and activity counter equal to 0. This activity counter is increased each time the constraint is a conflicting or reason constraint at conflict analysis, and is divided by 2 at each cleanup. Cleanups are done periodically, in such a way that the constraint database grows rather slowly over time.

Early backjump and conflict analysis. Performing early backjumps is in fact optional. When omitted (or not done whenever possible), the price to be paid is the loss of an invariant of the stack: it is no longer true that before each decision all bounds are exhaustively propagated. A slight modification of conflict analysis suffices to handle this: before starting conflict analysis, pop bounds from the stack until the initial CS contains at least one bound at or above the topmost decision. We have not done any thorough experiments yet evaluating this option. One could, for example, do, or attempt to do, early backjump with the intermediate conflicting constraint C if it is false in the current stack, or only if it is promising (e.g., short) according to some heuristic. In any case, further work on the implementation should probably cover a much better implementation for early backjumping, which is currently a black hole for efficiency.

Several other improvements exist for conflict analysis. For example, the quality of backjumps and the strength of the reason sets can be improved by doing some more work: the CS can be simplified by removing bounds that are subsumed by stronger ones, and also, instead of using the pre-stored reason sets R, one can re-compute them on the fly during conflict analysis with similar aims. One can also do a bit of search during conflict analysis, e.g., by trying to remove non-topmost bounds and do cuts with these, with the aim of finding good early backjump cuts.

Optimization. For optimization many further ideas exist: heuristics for finding a first solution quickly (which helps bounding the search dramatically), heuristics for choosing the decision bound in a "first-succeed" manner (i.e., steering it towards minimizing the cost function). For problems of a more numerical nature with many solutions, one could also search for the optimal solution with binary search instead of decreasing the objective one by one.

Pre– and in-processing, arithmetic. For some problems currently too many cuts are discarded because of coefficients larger than 2^{30}. One can look for solutions from the implementation point of view, e.g., by using large integer arithmetic, or using floating point arithmetic, but it might also be the case that more constraint simplification pre- and in-processing techniques can be helpful (and not only for this purpose). For propositional SAT, the so-called lemma shortening techniques introduced in MiniSAT [12] have turned out to be essential for modern SAT solvers and (in fact, several extensions of them) can be applied to IntSat as well. Modern SAT solvers such as lingeling [3] heavily apply different inprocessing techniques to keep the constraint database small but strong.

MIPs. Finally, it needs to be worked out how to apply IntSat in order to solve MIP instances, i.e., where not all variables are subject to integrality. One can decide on the

integer variables as it is done now, and at any desired point one can run an LP solver to optimize the values for the rational variables. The inclusion of lower bounding techniques, well-known from modern MIP solvers, also needs to be considered.

5 Experiments

All experiments described in this section were carried out on a standard 2.66GHz 4-core Intel i5 750 desktop. The reader can easily verify these results; in particular our prototype IntSat implementation including source code and all benchmarks can be downloaded from [19].

CPLEX and Gurobi. We compare with the newest versions of the commercial solvers CPLEX (v.12.6) and Gurobi (v.5.6.2). Both use all four processor cores (while IntSat uses only one!). Both are well-known to outperform, in general by far, the existing non-commercial solvers. The technology behind these solvers is extremely mature, after decades of improvements: according to [5], between 1991 and 2012 they have seen a 475000 times speedup from algorithmic improvements only (i.e., not counting another 2000 times from hardware improvements)!.

Rather than using a single method, these solvers apply a large variety of techniques, including, e.g., specialized cuts (Gomory, knapsack, flow and GUB covers, MIR, clique, zerohalf, mod-k, network, submip,etc.), heuristics (rounding, RINS, solution improvement, feasibility pump, diving, etc.) and variable selection techniques (pseudo costs, strong branching, reliability branching, etc.).

They also apply sophisticated *presolve* methods to reduce in advance the size of the problem and to tighten its formulation. Since we have no special-purpose presolve implementation for IntSat (yet), unfortunately here we had to use Gurobi's one, and for fairness, we used Gurobi to presolve and output all instances (which took essentially negligible time) and ran all three solvers on these Gurobi-presolved instances.

Other classes of solvers. It is well known that SAT Modulo Theories (SMT) [20] solvers such as Mathsat5 [8], Yices [11], Z3 [10] or our own Barcelogic solver [6] mostly focus on efficiently handling the arbitrary Boolean structure on top of the LIA constraints. Their *Theory Solver* component, the one that handles *conjunctions* of constraints (our aim here), is rather basic, and we do not compere here with SMT solvers since on conjunctive problems they are indeed in general orders of magnitude worse than CPLEX or Gurobi.

Concerning SAT and Lazy Clause Generation (LCG) [21], from our own work (see among many others [1]), we also know too well that solvers that (lazily) encode linear constraints into SAT can be competitive as long as problems are rather Boolean, without a heavy ILP/optimization component. Also CSP solvers such as Sugar [24], which heavily focus on their rich constraint language, are in general very far from the commercial OR solvers on the typical hard pure ILP optimization problems.

Also, most of these SAT/SMT/LCG solvers cannot optimize or are rather bad at it. Cutsat [14] cannot optimize either.

Random optimization instances. We used a random generator to create 100 optimization instances with 600 variables (about half of them non-binary) and 750 constraints (instances and generator are available at [19]). Then we discarded the 51 "too easy" instances (for which all three solvers could find an optimal solution and *prove optimality* in less than 2s).

The first columns (I, C, G) in the table below show runtimes in seconds of, respectively, IntSat, CPLEX and Gurobi to prove optimality, and no time indicates timeout after 10s, which happens 17 times for IntSat, 19 times for CPLEX and 9 times for Gurobi.

Since finding good solutions quickly is perhaps as important as proving optimality, the following columns show the cost of the optimal solution ("opt"), and the best solutions found after 10s, only when different from the optimal one. The reader can check that IntSat fails 8 times to find the optimal solution, CPLEX 13 times, and Gurobi 7 times, and that the total sum of distances to the optimal solutions in these cases are 22, 39 and 17, respectively. When given longer runtimes, the commercial solvers tend to behave better on these instances than the current version of IntSat. However, this should of course be re-evaluated after a more mature implementation, heuristics, and pre- and inprocessing, etc., become available for IntSat.

	I	C	G	opt	I	C	G
01.lp				-7	-5	.	-4
03.lp		7.76	5.41	-7	.	.	.
05.lp	1.85	1.08	4.19	-4	.	.	.
06.lp	2.50			-13	.	-7	10
07.lp	1.06	2.98	3.94	-7	.	.	.
10.lp	4.21	0.17	0.07	-11	.	.	.
12.lp			9.80	-9	.	.	.
14.lp	0.77	5.78	3.16	-14	.	.	.
15.lp	1.43	2.56	0.21	-10	.	.	.
16.lp			5.02	-9	.	-8	.
20.lp				-8	-6	-4	-6
21.lp		0.71	0.19	-7	-5	.	.
23.lp				-8	.	.	.
24.lp	2.85	0.08	0.05	-4	.	.	.
26.lp	1.04		2.93	-11	.	-7	.
27.lp		2.71	6.32	-2	.	.	.
28.lp	2.16			-9	.	-7	-8
31.lp				-6	.	-3	-3
33.lp	2.79	6.38	2.92	-6	.	.	.
34.lp	2.62	1.86	0.30	-13	.	.	.
36.lp	6.58	1.53	5.23	-9	.	.	.
40.lp	2.02	3.10	0.05	-18	.	.	.
44.lp	4.76	7.47	8.54	-10	.	.	.
49.lp	2.77	0.47	0.09	-8	.	.	.
50.lp			0.22	-12	-8	-11	.

	I	C	G	opt	I	C	G
53.lp		2.47	3.05	-13	-7	.	.
60.lp	1.69		2.19	-9	.	-8	.
61.lp	1.47	1.14		-16	.	.	-15
62.lp	2.58	0.53	0.02	-3	.	.	.
63.lp			4.55	-12	.	.	.
64.lp	1.56	6.33	2.60	-4	.	.	.
65.lp	3.26	0.80	0.81	-8	.	.	.
66.lp	1.32	9.23	4.47	-5	.	.	.
68.lp				-9	.	.	.
69.lp	5.91	1.20	0.13	-14	.	.	.
70.lp	8.33	0.24	0.09	-6	.	.	.
73.lp			4.75	-11	-9	-9	.
76.lp	0.74	2.89	0.40	-11	.	.	.
78.lp				-8	-5	-2	-4
79.lp	8.26	0.36	0.06	-4	.	.	.
80.lp	7.54	2.59	0.13	-7	.	.	.
81.lp	0.86	4.71	4.78	-9	.	.	.
84.lp	2.93		6.40	-12	.	-5	.
87.lp				-10	.	-8	.
88.lp			9.67	-5	-4	-4	.
91.lp	2.38		2.96	-9	.	.	.
93.lp	2.98	3.03	0.45	-9	.	.	.
95.lp	1.51	3.13	0.14	-10	.	.	.
99.lp	0.62	2.00	4.92	-7	.	.	.

5.1 MIPLIB instances

From the MIPLIB 2010 Mixed Integer Problem Library, a well-known "standard test set" to compare optimizer performance for the Operations Research (OR) community, cf. `miplib.zib.de`, we considered *all* 30 ILP instances (i.e., with integer and binary variables only) and discarded the 11 instances lacking initial lower and upper bounds for some variable.

For the remaining 19 ones, the next table below indicates runtimes (in s) needed to prove feasibility (as recommended for the commercial solvers, the objective function was replaced by 0). The table also includes some statistics on number of constraints, total number of variables, and among these, the number of binary variables.

We ran IntSat with no presolving, as Gurobi's presolve was harmful for it in some cases. Here we also compare with the Cutsat implementation of [14], which currently can only handle feasibility, and no optimization.

	Feasibility (s)				**Problem statistics**		
	IntSat	CPLEX	Gurobi	Cutsat	#constr.	total #vars.	#0/1-vars.
30n20b8	20.14	0.83	0.41	>300	666	18380	11036
d10200	10.00	0.20	0.22	>300	1147	2000	733
d20200	0.55	0.34	1.15	>300	1702	4000	3181
lectsched-1	1.11	7.75	39.73	64.84	51608	28718	28236
lectsched-1-obj	1.09	145.25	11.71	45.77	51608	28718	28236
lectsched-2	0.60	3.50	1.14	5.39	31775	17656	17287
lectsched-3	0.99	6.94	6.82	18.43	46615	25776	25319
lectsched-4-obj	0.25	0.15	0.38	0.86	14760	7901	7665
mzzv11	1.27	0.04	1.26	0.08	12871	10240	9989
neos-1224597	0.20	0.11	0.15	>300	3682	3605	3150
neos16	0.09	9.74	16.54	>300	1028	377	336
neos-555424	0.07	0.07	0.07	18.53	2746	3815	3800
neos-686190	1.98	0.44	0.27	>300	3785	3660	3600
ns1854840	3.21	3.31	1.57	3.75	151216	135754	135280
rococoB10-011000	0.06	0.05	0.06	0.06	3063	4456	4320
rococoC10-001000	0.05	0.04	0.05	4.45	2298	3117	2993
rococoC11-011100	0.10	0.16	0.10	0.05	4403	6491	6325
rococoC12-111000	0.16	0.15	0.27	>300	13181	8619	8432
sp98ir	0.16	0.15	0.28	0.92	1531	1680	871

5.2 Optimizing the MIPLIB instances

We also considered optimizing these same 16 MIPLIB instances (lectsched-1, -2 and -3 are feasibility ones), using all three applicable solvers (IntSat,CPLEX,Gurobi) with a timelimit of 600s.

30n20b8 and lectsched-4-obj: For these two instances, all three solvers find the optimal solution and prove optimality in less than 10 seconds. For 30n20b8, in 1.03s, 3.59s and 4.18s, respectively for IntSat, CPLEX, and Gurobi (optimal solution has cost 302) and for lectsched-4-obj, in 0.43s, 3.37s, and 1.61s respectively (optimal is 4). Note that

on both instances IntSat is fastest. IntSat is currently not able to prove optimality for any of the other 14 MIPLIB optimization instances in less than 600s.

neos16 and neos-1224597: IntSat does find optimal solutions quickly for these two problems. For neos16, none of the three solvers proves optimality in less than 600s, but IntSat finds the optimal solution (cost 446) in 13.1s, whereas CPLEX needs 39.2s and Gurobi needs 45s. For neos-1224597, all three solvers find the optimal solution (cost -448) in around 1s., but CPLEX and Gurobi moreover prove optimality.

The other 12 instances: CPLEX and Gurobi also prove optimality for five other instances, the same ones for both solvers, given in the first table below.

Results and best found solutions for the remaining seven instanes are given in the second table below. Out of these seven, two (d10200, rococoC11) are catalogized in the MIPLIB as "hard" and three further instances (d20200, lectsched-1-obj, and ns1854840) are "open", as their optimal cost is unknown. IntSat is the best solver by far on the open problem ns1854840, even though IntSat's 600s refer to runtime on one core only, whereas the other solvers run 600s on all four cores. In fact, for this instance Gurobi only finds an initial "heuristic" solution that is more than 100 times worse than the one found by IntSat; this happens because Gurobi's root simplex times out after 600s. Of the other "open" problems, IntSat is also better than CPLEX on two other problems: lectsched-1-obj and rococoC11-011100.

In some cases IntSat appears to be quite improvable still, e.g., due to its too naive handling of very large input problems. Sometimes also better optimization heuristics will to be useful on instances with a very numerical nature and slowly decreasing values of the objective function.

	Best solutions found			
	IntSat (600s)	optimal	time CPLEX	time Gurobi
mzzv11	-18368	-21718	16.64s	21.73s
neos-555424	1369300	1324300	4.72s	2.16s
neos-686190	11380	6730	28.95s	24.21s
rococoC10-001000	13402	11460	49.89s	436.31s
sp98ir	279007104	219676790	24.17s	33.07s

	Best solutions found after 600s		
	IntSat	CPLEX	Gurobi
d10200	12809	12441	12438
d20200	13619	12279	12262
lectsched-1-obj	92	93	85
ns1854840	288000	392000	4272000
rococoB10-011000	21462	19449	19810
rococoC11-011100	21427	21800	20957
rococoC12-111000	57118	36988	35845

6 Related Work and Conclusions

We already mentioned the work on Cutsat [14]. The idea of applying conflicting sets is not only reminiscent to the conflict analysis of SAT, but also of *SAT Modulo Theories* (SMT) [20, 2] for the theory of linear arithmetic, with the main difference, among others, that here new ILP constraints are obtained by cut inferences, normalized and learned, and not only new Boolean clauses that are disjunctions of literals representing bounds (usually only those that occur in the input formula). Other SAT/SMT related work, but for rational arithmetic is [17, 15, 9].

It is also worth mentioning that there may be some possible theoretical and practical consequences of the fact that IntSat's underlying cutting planes proof system is stronger than CDCL's resolution proof system: could IntSat outperform SAT solvers on certain SAT problems for which no short resolution proofs exist? E.g., pigeon-hole-like situations do occur in practical problems (think of timetabling or scheduling). A similar question applies to the current SMT solvers, which are based on resolution as well [18].

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, which perhaps will also include IntSat at some point. Still, IntSat by itself already appears to be the first alternative method for ILP that uses no LP relaxations and no simplex that is competitive on certain hard optimization problems, and moreover it still has an enormous potential for enhancement. We expect that this work will trigger quite some further activity on all the improvements mentioned in Section 4.

Acknowledgments. Thanks go to Albert Fiol for his work last year, as an undergraduate student, on a preliminary IntSat implementation, to Enric Rodríguez and Albert Oliveras for always trying to answer my questions, and to Dejan Jovanović for his help with cutsat and to my muse Mariona. Partially supported by the Spanish government under the SweetLogics project (TIN 2010-21062-C02-01).

References

1. 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.
2. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.
3. Armin Biere, 2010. Lingeling SAT Solver. Available at `http://fmv.jku.at/lingeling/`.
4. 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.
5. Bob Bixby. Presentation: 1000X MIP Tricks, 12 June 2012, Bill Cunninghams 65th, 2012.
6. Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The Barcelogic SMT Solver. In *Computer-aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 294–298, 2008.

7. Vasek Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4(4):305–337, 1973.
8. Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.
9. 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.
10. L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. Technical report, Microsoft Research, Redmond, 2007. Available at `http://research.microsoft.com/projects/z3`.
11. B. Dutertre and L. de Moura. The YICES SMT Solver. Technical report, Computer Science Laboratory, SRI International, 2006. Available at `http://yices.csl.sri.com`.
12. Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *6th International Conference on Theory and Applications of Satisfiability Testing, SAT '03*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.
13. Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1973.
14. Dejan Jovanovic and Leonardo Mendonça de Moura. Cutting to the chase - solving linear integer arithmetic. *J. Autom. Reasoning*, 51(1):79–108, 2013.
15. 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.
16. J. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
17. 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.
18. Robert Nieuwenhuis. SAT and SMT Are Still Resolution: Questions and Challenges (invited talk). In *Automated Reasoning - 6th International Joint Conference, IJCAR*, pages 10–13. Springer LNCS 7364, 2012.
19. Robert Nieuwenhuis. Intsat source code, makefile, benchmarks and benchmark generators, 2014. `www.lsi.upc.edu/~roberto/IntSatCP2014.tgz`.
20. 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.
21. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007.
22. Knot Pipatsrisawat and Adnan Darwiche. On modern clause-learning satisfiability solvers. *Journal of Automated Reasoning*, 44(3):277–301, 2010.
23. Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Chichester, 1986.
24. Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.