Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Facultat d'Informàtica de Barcelona (FIB)

Grau en Enginyeria Informàtica - Especialitat en Computació

Solving integer programming problems using DPLL-based algorithms

Supervisor:
Robert Nieuwenhuis
Department: LSI

Author:
Albert Fiol

Defense date: June 20, 2013

Contents

1 Abstracts

1.1 Català

En aquest projecte hem desenvolupat un SAT solver per a problemes de programació lineal entera, també anomenada programació entera. La programació lineal entera (ILP) és un problema NP-difícil, a diferència de la versió racional, problema que va ser provat estar a P en la dècada dels 70.

Tradicionalment, els solvers de problemes ILP utilitzen l'algorisme Simplex com a un dels elements principals. Simplex, introduït per Dantzig el 1947, resol problemes de programació lineal (racional). Segueix una interpretació geomètrica del problema: les restriccions defineixen un poliedre convex a l'espai, i la funció objectiu especifica una direcció en la qual es vol otpimitzar. Simplex recorre els vèrtexos del poliedre fins a trobar-ne un d'òptim, que pot ser no enter. Perquè Simplex pugui resoldre problemes de programació lineal entera, hem d'aplicar tècniques addicionals, com afegir restriccions noves (*cutting planes*) o utilitzar mètodes basats en *branch and bound*.

El nostre enfocament és diferent. El nostre solver, *Intsat*, és un SAT solver en el qual les variables són enteres. Per a desenvolupar-lo hem adaptat l'algorisme DPLL per a que suporti variables enteres. En aquesta memòria explicarem el procediment, la implementació i mostrarem resultats, comparant intsat amb altres programes com cutsat o cplex.

1.2 Castellano

En este proyecto hemos desarrollado un SAT solver para problemas de programación lineal entera, también llamada programación entera. La programación lineal entera (ILP) es un problema NP-difícil, a diferencia de su versión racional, cuya pertenencia a la clase P fue probada en la década de los 70.

Tradicionalmente, los solvers de problemas ILP utilizan el algoritmo Simplex como uno de sus ingredientes principales. Simplex, introducido por Dantzig en 1947, resuelve instancias de programación lineal (racional). Funciona siguiendo una interpretación geométrica del problema: las restricciones definen un poliedro convexo en el espacio, y la función objetivo especifica una dirección en la cual optimizar. El algoritmo recorre los vértices del poliedro hasta encontrar uno óptimo, que puede no ser entero. Para que Simplex pueda resolver problemas de programación lineal entera, hay que usar técnicas adicionales como añ adir nuevas restricciones (*cutting planes*) o usar métodos basados en *branch and bound*.

Nuestro enfoque es diferente. Nuestro solver, *Intsat*, es un SAT solver en el cual las variables son enteras. Para desarrollarlo hemos extendido el algoritmo DPLL para que soporte variables enteras. En esta memoria explicaremos el

procedimiento e implementación, y presentaremos resultados de la comparación entre intsat y otros programas como cutsat o cplex.

1.3 English

In this project we have developed a SAT solver for integer linear programming problems (ILP), also called integer programming problems. Although the rational version of Linear Programming belongs to P (this was proved in the 70s), the integer version is an NP-hard problem.

Traditionally, ILP solvers use the Simplex algorithm as a main ingredient. Simplex, introduced by Dantzig in 1947, solves rational linear programming problems. It works by following a geometrical interpretation of the problem: constraints define a convex polyhedron in space, and the objective function explicits a direction. The algorithm traverses vertices of the polyhedron until an optimal one (which may be non-integer) is found. For Simplex to be able to solve integer linear programming problems, one must apply some additional techniques such as adding new constraints (*cutting planes*) or *branch and bound* methods.

Our approach is different. Our solver, *Intsat*, is a SAT solver that uses integer variables. To develop our solver, we have extended the DPLL algorithm to make it support integer variables. In this document we will explain the procedure and implementation, and we will compare it against other solvers such as *cutsat* and *CPLEX*.

2 Motivation

Many problems can be expressed as the maximization or minimization of an objective function, given limited resources and constraints that relate these resources. These constraints take the form of linear equalities or inequalities. With all these elements we have a linear programming problem: which is the assignment (a feasible value for each variable) that maximizes or minimizes an objective function?

The first algorithm to solve these problems was developed during the World War II to reduce costs to the army. However, this method was kept secret and the field would have to wait until George Dantzig's simplex method appeared. In 1979, the linear programming problem was proved to be solvable in polynomial time. Not long after that, in 1984, Narenda Karmarkar developed a new method for solving linear programming problems that could work as fast, or even faster, than the simplex method. All of these advances have made possible for linear programming to be used in a vast number of applications. Many of them can be found in industry, which requires efficient planning to reduce costs or find the most suitable solution to a problem.

However, these methods work with problems in which variables can take values from the rational number set. This may work in some applications, but many real-life problems cannot have rational solutions since the variables are naturally integer. Linear programming problems were shown to be solvable in polinomial time in 1979; however, integer linear programming (that is, a linear programming problem in which some or all variables are restricted to be integers) is much harder than its rational counterpart. The integer linear programming problem is NP-hard.

Research on integer linear programming has also been done in parallel with research on linear programming. In 1958, Gomory extended the simplex method to deal with integer-restricted problems. His method consisted in deriving new constraints once a non-integer solution was found; these constraints would keep rendering non-integer solutions unfeasable until an integer based one was found. Apart from this approach, there are other forms of solving integer programming problems: for example, heuristic methods to find a "good enough" (but not the best) solution, such as hill climbing, simulated annealing or more sophisticated metaheuristics.

In our case, we are not interested in finding the most efficient solution, but in finding wether a solution exists or not. In other words, we want to know if an integer linear programming problem is satisfiable: does an assignment exist such that all the constraints hold true? This is closely related to the boolean satisfiablity problem and, not surprisingly, it is very easy to rewrite a SAT problem in linear inequality form. The next example illustrates this fact:

$$x_1 \vee \overline{x_2} \vee x_3 \quad \wedge \quad x_1 \vee x_2 \vee \overline{x_3}$$

becomes

$$x_1 + (1 - x_2) + x_3 \geq 1 \quad \wedge \quad x_1 + x_2 + (1 - x_3) \geq 1 \quad 0 \leq x_1, x_2, x_3 \leq 1$$

In other words, variables are restricted to be either 0 or 1 (false and true, respectively). A boolean clause is true if at least one of its variables is true: this is equal to say that if a true variable has a value of 1 and a false variable has a value of 0, the value in the left side of the inequality must be at least 1.

SAT has been thoroughly studied and there have been many important advances in the last years, to the point that one can solve problems with millions of variables using SAT solvers. The main reason of this success is the DPLL algorithm, the first versions of which were introduced in 1962 by Davis, Putnam, Logemann and Loveland. It is a method that runs by choosing an unassigned variable and assigning a value to it; this can lead to a simpler formula which is treated recursively. If that formula is satisfiable, then the original formula is also satisfiable. However, if the simpler formula is not satisfiable, then DPLL backtracks and continues the search. Nowadays, DPLL has been extended with additional functionalities such as *learning*, *backjumping* and *restarts*, among other enhancements.

As said, SAT solvers are commonly used to solve many real-life problems. However, there is a quite important drawback: SAT is limited to boolean constraints. This makes trying to solve some problems really complicated, because many problems are naturally non boolean; they can be encoded in boolean form, but paying a high price reflected in a growth in the numbers of variables and constraints.

Research in this field has been going on for the last decades, but there are many extensions to SAT that have not been studied that much. In our case, we will deal with generalizations of the boolean satisfiability problem. Particularly, we will focus on integer linear programming [1]: the set of feasibility problems in which all the variables are restricted to be integers and their values are restricted to fall between two specified bounds. From now on, we will refer to this set of problems as *int-SAT* problems.

Being a generalization of SAT, int-SAT problems are very useful in areas such as theorem proving, circuit design, or operations research. Furthermore, being an NP-complete problem, finding an algorithm that solves it efficiently (to a certain degree) can be useful to solve other problems and, of course, boost the level of research in this field.

2.1 Context

We will now talk about a programming paradigm related to our project, the kind of problems we want to solve and our project's stakeholders.

2.1.1 Constraint programming

Many different programming paradigms exist: in our case, a quite interesting one is called Constraint Programming. As Eugene C. Freuder once stated, "Constraint Programming represents one of the closest aproaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it" [2]. This is a basic characteristic shared with other languages (this is a form of *declarative programming*). In these cases, instead of telling the computer what to do (that would be an imperative approach), the user only *declares* the properties of a desired solution. From this, the computer has the job to find such a solution, if it exists.

Constraint Programming presents a series of advantages with respect to other paradigms when it comes to solving certain kind of problems. Firstly, this ease of use is quite attractive, since once a good algorithm that solves constraint programming problems is made, all the work is shifted from the user to the computer. Thus, users only need to know how to express constraints to be able to solve problems. Secondly, a huge number of interesting problems can be expressed with a series of constraints. This universality means that having a good algorithm can make the problem solving process much easier.

A typical constraint satisfaction problem consists of a group of variables, each one with a *domain* (the set of possible values that variable can take) and a set of constraints that restrict these domains. For example, consider the *Nurse Scheduling Problem*: in this problem, a hospital wants to assign shifts and holidays to a group of nurses, trying to get a fair schedule for everybody. The constraints may take different forms:

- A nurse can not work three consecutive shifts.
- On certain days, the number of working nurses must be at least 10.
- There are time periods in which certain nurses will not come to work because of holidays. These days are specified by the nurses as a list of preferences.

One can see that there are many possible types of constraints, and that the complexity may escalate very quickly as the number of nurses and constraints grows.

The nurse scheduling problem is an example of classic constraint satisfaction. However, problems can come in many ways depending on the domain specified for each variable. For example, there may be boolean variables (the classic SAT problem), but there also may be numerical variables [3].

2.1.2 Problems we want to solve

In our project we are dealing with integer domains and constraints expressed as linear inequalities of polynomials. In this case, problems may be stated with another added feature: an objective function the user wants to optimize. For example, in the nurse scheduling problem, a possible objective function would count the number of constraints that can not be satisfied. In that situation, the solver would try to find a solution that minimizes that number. This means that if there is no solution that satisfies everybody, at least the most "fair" one will be found.

This optimization process belongs to linear programming, but we are not going to deal with it in this project. The objective of our solver is to find a solution that satisfies the whole set of constraints, assigning each variable a value that belongs to its domain.

2.1.3 Stakeholders

As we have stated, our project is research oriented. It has no immediate objective of being commercialized or used in real problems yet, since it still needs polishing and work. However, once the solver is completed and ready for deployment, it can solve many problems from many different fields such as operations research, scheduling, or even life sciences such as medicine.

Any constraint programming problem that can be expressed with linear constraints and integer domains for the variables can, thus, be solved by our solver. Classic SAT problems can be solved too, since boolean constraints can be expressed in form of equations.

3 State of the art

Today, different strategies are used to solve this problems. The most commonly used approach are SAT Solvers for boolean domains. However, for other types of domains there are other strategies such as Satisfiability Modulo Theories solvers (SMT) or completely different algorithms such as the Simplex method.

3.1 SAT Solvers

SAT Solvers work on the boolean satisfiability problem. This was the first problem proved to be NP-complete[4] and thus is a very important challenge to solve. The input of the problem is a logical formula, usually in conjunctive normal form (CNF), and a set of variables. By definition, the domains of the variables are true/false, and the constraints are expressed as *clauses*.

The DPLL algorithm is a complete and correct algorithm that solves the SAT problem [5]. Many SAT solvers use it as a base to start working on developing faster algorithms to solve problems. In our case, we will have a basic DPLL structure and we will start improving it with better data structures, better heuristics and better conflict-resolution procedures.

SAT Solvers have been improved for years and can be quite efficient at solving some problems. However, boolean formulas can be very tedious to work with, especially if the problem is not "naturally boolean", i.e., it is hard to express it in form of boolean constraints. In this case, it is more adequate to use generalizations of SAT or even different algorithms.

3.2 Simplex-based methods

The Simplex method is an algorithm to solve linear problems (optimizating an objective function subject to a set of constraints). It was first published by George Dantzig in 1947 and has been key in our linear programming solving techniques.

The algorithm works by having the constraint set define a convex polyhedron in space and then travelling through its vertices until an optimal one is found. This is, thus, a more geometrical approach than the ones done by the previous methods we have stated. Simplex has proved to be a very efficient method in practice; however, it has been shown that it has an exponential cost for the worst case [6]. This can be mitigated by introducing certain variations and changes, but we can not change the fact that there will always be some problems that exponential time to solve.

3.3 Our proposal and Jovanović and De Moura's *cutsat*

Our solver is not based on Simplex. Instead, it uses a DPLL-like algorithm that works with integer variables at its core.

3.3.1 cutsat

Our work began in 2011 after reading Dejan Jovanović and Leonardo de Moura's paper in which they present an algorithm for solving linear integer programming problems [7]. In this project, we first implemented this algorithm. We then added our own methods and finally even changed some of its fundamentals, mostly in the way *cut procedures* are applied to learn new constraints during the search.

4 Scope description

In this project we will develop a solver for int-SAT problem instances from a research-oriented perspective. Having an already working int-SAT solver to begin with, we will use it to compare our results and test different strategies. All these strategies come in form of algorithms, data structures and heuristics to improve the search. By the end of the project we will have a working int-SAT solver with the best methods we have found along the way.

Our solver will be based on the DPLL algorithm. From there we will implement *lemma learning* and *cleanup*, *backjumping* after conflicts and efficient propagation algorithms. After that we will be implementing data structures to accelerate the process. Once the basic body is finished, we will implement heuristics to try and achieve better search results (following fail-first and succeed-first principles to make more powerful decisions). Once we have completed all this parts we will think of new improvements to code.

5 Objectives

The project has, as a first objective, developing an SAT solver for int-SAT problems to aid research on this field. The solver is desired to have the following characteristics:

- **A certain degree of efficiency.** Not all instances can be solved efficiently, but we expect our program to solve some instances in a reasonable time. To have a clearer purpose, we will use cutsat: our goal is to make our solver at least as good as the current implementation of cutsat.

- **Analysis and debugging tools.** Our solver has to have a way to tweak parameters quickly and enable/disable certain modules to learn about their behaviour.

The project also has, as a secondary objective, the analysis of new algorithms and implementation techniques to solve int-SAT problems. This will be accomplished testing our program under specific environments and disabling specific modules of the solver.

6 Methodology

During our project, we will use an iterative method consisting of various phases. In each phase, we will define a series of improvements or new things to add: this is the "theory" part, in which we will discuss ideas and methods. After the theory comes the "implementation" period, in which we will code the things we agreed upon previously. After this part, the current solver will be tested against a set of randomly generated instances (our benchmarks) and any bugs found will be debugged. This is the "debugging" part and marks the end of a phase.

Each phase will correspond to one or two week periods, depending on the complexity of the implementation and debugging parts. After each phase, a meeting will be held to discuss next phase's options and start again the process.

When we have a functional version we are confident with, we will test it on special problems and compare it to cutsat and other public int-SAT solvers that we can find. These tests will give us true insight about how our program works since we will be able to compare results to other algorithms.

7 Acceptance criteria

Even if we can not manage to build a solver as good as cutsat, we can still take advantage of our work. As we stated earlier, this project is research oriented and its main goal is to find new ways to develop solvers for int-SAT problems. In this case, the acceptance criteria for this project are the following:

- **Develop a functional solver.** The resulting program has to work with a wide array of examples. It is highly desirable to do it in a reasonable time, but not required.

- **Compare algorithms and state-of-the-art techniques.** Our program needs to allow us to test different algorithms to be able to compare them, running against sets of tests. If we can extract useful conclusions and gain insight into this area, this criterion will be satisfied.

8 Limitations and risks

8.1 Limitations

The most important limitations in this project are time and usefulness:

- **Time.** Once the developing stage finishes, no further improvements or additions will be done to the solver. At that point, the version that is currently working will be the final version. This means we must get the best performace possible in the time we have.

- **Usefulness.** If the solver is not good enough for big problems, or the performace is not the one we expected, we can still take advantage of it. The solver's design must be one that is simple to read and understand, and leaves future work open to do. This constraint is not as hard as the previous one, but it affects the design and developement of our program.

8.2 Risks

Many problems can arise during the developement of this project. Among them, the most important are:

- **Not general enough algorithms.** In other words, coming up with algorithms that do not work for all of the problems (due to a conceptual error or some other factor).

- **Unexpected errors.** Even if the theory is correct, there are many errors that can appear once we have started our implementation. These errors can delay the implementation and debugging phases, or even bring us back to the theory phase.

- **Benchmarks.** Our set of test problems has a limited size. This means that there is a chance of them being not significative or big enough to find errors on time, which would delay our developement.

9 Definitions

In this section we will formalize the problem and explain the notation we will be using throughout this document. Apart from this, we will include here definitions for some terms used in the following pages.

9.1 Basic notation

We will use x, y, z to refer to variables in $\mathbb{Z}$. We will also use a, b, c to denote *coefficients*, elements in $\mathbb{Z}$. A *linear polynomial* is a polynomial of the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n + k$, where the a_i and k are coefficients and x_i are variables. A *constraint* is a linear inequality of the form $p \leq 0$, where p is a linear polynomial. We will use $p, q, r...$ to denote linear polynomials, and we will use the symbols $C, D, E...$ to denote constraints.

Any linear inequality with a different form can be expressed in this form using the following rules:

- $p < 0$ can be expressed as $p + 1 \leq 0$
- $p \leq k$ becomes $p - k \leq 0$
- $p > 0$ becomes $-p < 0$
- $p = 0$ becomes $p \leq 0 \quad \wedge \quad -p \leq 0$

A *lower bound* of a variable x is an expression of the form $x \geq k$, where k is an integer. An *upper bound* of a variable x is an expression of the form $x \leq k$. We will use $lower(x)$ and $upper(x)$ to denote (respectively) the lower and upper bounds of a given variable x. Finally, to denote a generic bound (either lower or upper) for a variable x, we will use $x \bowtie k$. At some points we will specify both bounds at the same time for a variable x, writing $k \leq x \leq k'$, where $lower(x) = k$ and $upper(x) = k'$, assuming that indeed always $k \leq k'$.

Bounds can be expressed as constraints with one variable, and constraints with one variable are equivalent to bounds. The *negation* of a lower bound $x \geq k$ is an upper bound of the form $x \leq k - 1$. The *negation* of an upper bound $x \leq k$ is a lower bound of the form $x \geq k + 1$. We will write $\neg b$ to denote the negation of the bound b.

If X is a set of variables, an *assignment* is a function $\sigma : X \to \mathbb{Z}$. If p is a polynomial $a_1x_1 + \cdots + a_nx_n + k$, we write $p\sigma$ to denote $a_1\sigma(x_1) + \cdots + a_n\sigma(x_n) + k$.

9.2 Problems

In this document we define an *ILP problem* as a set of constraints S, a set of variables X, and for each variable x_i a lower bound and an upper bound, $k_i \leq x_i \leq k_i'$.

A *solution* for such an ILP problem is an assignment σ such that for every constraint $p \leq 0$, we have $p\sigma \leq 0$ and $k_i \leq \sigma(x_i) \leq k_i'$ for every variable x_i.

The aim of this work is to find solutions for ILP problems as efficiently as possible, or to prove that no such solution exists.

10 Main search procedure

In this section we will first explain the details about how the DPLL algorithm works, and after that we will introduce our extensions to make it able to support integer variables.

10.1 Basic DPLL procedure

Basic DPLL deals with propositional logic. *Atoms* are propositional symbols from a finite set P. If $p \in P$, then p is a *positive literal* and $\neg p$ is a *negative literal* . The *negation* of a literal l, written $\neg l$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* is a set of literals and a *cnf* (formula) is a set of clauses. A (partial truth) *assignment* M is a set of literals such that $\{p, \neg p\} \subseteq M$ for no p. A literal l *is true in* M if $l \in M$, *is false in* M if $\neg l \in M$, and *is undefined* otherwise. M is *total* if no literal of P is undefined in M. A clause C is *true in* M if $C \cap M \neq \emptyset$, is *false in* M, denoted $M \models \neg C$, if all its literals are false in M, and is undefined otherwise. A cnf F is true in M (or *satisfied* by M), denoted $M \models F$, if all its clauses are true in M. In that case, M is called a *model* of F. If F has no models then it is *unsatisfiable*. We write $F \models C$ ($F \models F'$) if the clause C (cnf F') is true in all models of F. If $F \models F'$ and $F' \models F$, we say that F and F' are *logically equivalent*. We denote by $C \vee l$ the clause D such that $l \in D$ and $C = D \setminus \{l\}$.

Given a formula, DPLL works by incrementally building a satisfying truth assignment for all variables in the formula. At each step, the assignment is expanded either by *deciding* the truth value of an unassigned variable, or by *propagating* truth values of variables using logical rules.

The procedure is described as a transition system between a set of states. A state is either UNSAT or a pair $M \parallel F$, where M is a sequence of literals, and F is a finite set of clauses. We will denote the empty sequence of literals by $\emptyset$, unit sequences by their own literal, and the concatenation of two sequences by simple juxtaposition. Some literals l will be annotated as being *decision literals*, this fact will be denoted by writing l_d.

The transition relation between sets in DPLL is defined by a set of transition rules:

UnitPropagate:

$$M \parallel F \cup \{C \vee l\} \;\Rightarrow\; M\ l \parallel F \cup \{C \vee l\}, \qquad \text{if } \begin{cases} M \models \neg C \\ l \text{ is undefined in } M \end{cases}$$

Decide:

$$M \parallel F \;\Rightarrow\; M\ l_d \parallel F, \qquad \text{if } \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$$

Unsat:

$$M \parallel F \cup \{C\} \;\Rightarrow\; \text{UNSAT}, \qquad \text{if } \begin{cases} M \models \neg C \\ M \text{ contains no decision literals} \end{cases}$$

Backjump:

$$M\ l_d\ N \parallel F\ \Rightarrow\ M\ l' \parallel F, \qquad \text{if } \begin{cases} \text{there is some clause } C \vee l' \text{ s.t.:} \\ F \models C \vee l' \text{ and } M \models \neg C \\ l' \text{ is undefined in } M \\ l \text{ or } l' \text{ occurs in a clause of F} \end{cases}$$

These rules can be expanded with the learning and forgetting of *lemmas*. A lemma is a clause learned after finding a false clause that, had it been present before, would have prevented this clause to be false. This extension gives us DPLL with clause learning:

Learn:

$$M \parallel F\ \Rightarrow\ M \parallel F \cup \{C\}, \qquad \text{if } \begin{cases} \text{all atoms of } C \text{ occur in F} \\ F \models C \end{cases}$$

Forget:

$$M \parallel F \cup \{C\}\ \Rightarrow\ M \parallel F, \qquad \text{if } \{\ F \models C$$

Termination and correctness has been proved for both basic DPLL and DPLL with clause learning [8]. The algorithm can finish in two diferent ways: the first one happens when a conflict arises and there are no decision literals, which is the UNSAT state. The second one happens in absence of conflicts and when both UnitPropagate and Decide can not be applied. In this case, M is a model that satisfies F and thus, the problem is SAT, or satisfiable.

10.2 Basic Intsat procedure

In what follows, a *partial assignment* B is a sequence of bounds. If two lower bounds for the same variable $b_1 = x \geq k$ and $b_2 = x \geq k'$ are contained in B, and b_2 occurs later in B than b_1, we say that b_2 is *stronger* than b_1 and it must be true that $k' > k$. Similarly, if two upper bounds for the same variable $b_1 = x \leq k$ and $b_2 = x \leq k'$ are contained in B and b_2 occurs later in B than b_1, we say that b_2 is *stronger* than b_1 and it must be true that $k' < k$. From this one can infer that B is a sequence of increasingly stronger bounds for its variables.

A partial assignment such as B *induces* a lower bound and an upper bound for each variable: the *lower bound of x in B* is written as $lower(x, B)$ and is the strongest lower bound of x that is contained in B. Similarly, the *upper bound of x in B* is written as $upper(x, B)$ and is the strongest upper bound of x contained in B. The concepts of lower bound and upper bound can also be extended to constants, monomials and polynomials:

- Constants. Given a constant k, $lower(k, B) = upper(k, B) = k$.

- Monomials. Given a monomial $m = ax$, its bounds depend on the sign of a. If $a > 0$, then $lower(m, B) = a\ lower(x, B)$ and $upper(m, B) = a\ upper(x, B)$. If $a < 0$, $lower(m, B) = a\ upper(x, B)$, and $upper(m, B) = a\ lower(x, B)$.

- Polynomials. Given a polynomial $p = m_1 + m_2 + \cdots + m_n$, $lower(p, B) = k + \sum\ lower(m_i, B)$. Likewise, $upper(p, B) = k + \sum\ upper(m_i, B)$. This is simply the sum of bounds of each element in the polynomial.

The Intsat algorithm is an extension of DPLL. It is also described as a transition system between a set of states. A state is either UNSAT or a pair $B \parallel S$, where B is a partial assignment, and S is a set of constraints. The initial state contains the initial bounds in B and the set of constraints in S.

We say a variable x is *assigned* in B if $lower(x, B) = upper(x, B)$. We say a constraint $C = p \leq 0$ is *false in* B if $lower(p, B) > 0$. An assignment B is *complete* if one such that for every variable x_i, x_i is assigned in B. We say that a complete assignment is a *model* of a set of constraints S if for every constraint $C_j \in S$, C_j is not false in B. If such assignment exists, we say S is *satisfiable*, and if it does not exist, we say S is *unsatisfiable*.

The Intsat algorithm works by trying to build a model for the set of constraints S, starting from a partial assignment B that contains the initial bounds for variables. It either ends by reaching a UNSAT state, or by finding a model.

We say a set of constraints S entails a bound b under B if $S \cup \{\neg b\}$ is unsatisfiable, and we will write this as $S \models_B b$. Extending the partial assignment B can be done via the use of various transition rules such as *deciding* a bound (arbitrarily adding a new bound to the assignment) or deriving bounds from S. We say a bound of the form $x \bowtie k$ is *relevant in* B if it is stronger than the best bound of the same type for x in B. Again, we will denote the empty sequence of literals by $\emptyset$, unit sequences by their own bound, and the concatenation of two sequences by simple juxtaposition.

The set of Intsat's transition rules is also similar to DPLL's:

Propagate:

$$B \parallel S \cup \{C\} \Rightarrow M\ x \bowtie_C k \parallel S \cup \{C\}, \quad \text{if} \begin{cases} \{C\} \models_B x \bowtie k \\ x \bowtie k \text{ is relevant in } B \parallel S \end{cases}$$

Decide:

$$B \parallel S \Rightarrow B\ x \bowtie_d k \parallel S, \quad \text{if} \begin{cases} x \text{ occurs in a constraint of } S \\ x \bowtie k \text{ is relevant in } B \parallel S \end{cases}$$

Unsat:

$$B \parallel S \cup \{C\} \Rightarrow \text{UNSAT}, \quad \text{if} \begin{cases} C \text{ is false in } B \\ B \text{ contains no decisions} \end{cases}$$

Backjump:

$$B_1\ (x \bowtie_d k)\ B_2 \parallel S \Rightarrow B_1\ x' \bowtie k' \parallel S, \quad \text{if} \begin{cases} \text{there is some constraint } C \text{ s.t.:} \\ S \cup C \models_{B_1} x' \bowtie k' \\ x' \bowtie k' \text{ is relevant in } B_1 \parallel S \\ x' \text{ occurs in a constraint of } S \end{cases}$$

Learn:

$$B \parallel S \Rightarrow B \parallel S \cup \{C\}, \quad \text{if} \begin{cases} \text{all variables of } C \text{ occur in } S \\ S \models C \end{cases}$$

Forget:

$$B \parallel S \cup \{C\} \Rightarrow B \parallel S, \quad \text{if} \begin{cases} S \models C \end{cases}$$

Restart:

$$Bb_0B' \parallel S \Rightarrow B \parallel S, \quad \text{if} \begin{cases} b_0 \text{ is the first decided bound} \\ \text{restart conditions are met} \end{cases}$$

The way that we detect how does a constraint entail a bound (i.e. $C \models_B x \bowtie k$) will be explained in the next section. In the case of Intsat, a lemma is defined similarly than the DPLL counterpart, but instead of being a boolean clause it is a constraint.

A state of the form $B_0b_1B_1 \cdots b_nB_n \parallel S$, where $b_1 \cdots b_n$ are all the decided bounds, is said to be in *decision level* n. Furthermore, the bounds b_iB_i are said to be in decision level i.

The proof of correctness and termination is similar to DPLL's. We will show that there exist no infinite sequences of the form $B \parallel S \Rightarrow B' \parallel S' \Rightarrow \cdots$. To do this, we will define a well-founded strict partial ordering $\succ$ on states, and show that each rule application $B \parallel S \Rightarrow B' \parallel S'$ is decreasing with respect to this ordering.

Given a state $B \parallel S$, let B be of the form $B_0b_1B_1b_2B_2 \cdots b_pB_p$, where $b_1 \cdots b_p$ are all the decided bounds in B. Similarly, let B' be $B'_0b'_1 \cdots b'_{p'}B'_{p'}$. Define $m(B)$ to be the sum of $upper(x_i, B) - lower(x_i, B)$, for every variable x_i. In other words, $m(B)$ is the sum of all the magnitudes of the bound intervals for all variables in $B_0 \cdots B$. Now define: $B \parallel S \succ B' \parallel S'$ if:

1. There is some i with $0 \leq i \leq p, p'$ such that $m(B_0) = m(B'_0), m(B_1) = m(B'_1) \cdots m(B_i) > m(B'_i)$, or

2. $m(B_0) = m(B'_0), \cdots m(B_p) = m(B'_{p'})$ and $m(B) > m(B')$.

In other words, a state is more advanced than another if the magnitude of the bounds of the unassigned variables is smaller, or if there is some decision level in which this magnitude is smaller. It is easy to see that every transition rule moves to a more advanced state, if we define Unsat as a minimal state:

- Propagate moves to a more advanced state because it adds a bound to the sequence B. This reduces the magnitude of the bound interval of one variable, and for that reason, $m(B)$ is smaller (case (1) of the definition).

- Decide also adds a bound to the sequence. This bound is also stronger by definition, and so the transition rule moves to a more advanced state (case (2) of the definition).

- Unsat is defined as a minimal state, and thus moving to this state is always moving to a more advanced state.

- Backjump is used after finding a false constraint and always propagates some bound at a previous decision level. Thus, the magnitude of the bound intervals in that decision level is reduced (case (1) of the definition).

In the case of Learn, Forget and Restart, the condition still holds if we apply them with increased periodicity. We will show now that if at some point a constraint becomes false, then either Unsat or Backjump can apply.

Lemma 1. If $\emptyset \parallel S \Rightarrow \cdots \Rightarrow B \parallel S$, then if B is of the form $B_0 b_1 B_1 \cdots b_n B_n$, where $b_1, \cdots, b_n$ are the decided bounds, then $S \cup \{b_1 \cdots b_n\} \models_{B'} B_i$, for all i in $0...n$, and $B' = B_0 \cup \cdots \cup B_i$.

Suppose a constraint C becomes false under a certain partial assignment B. If there are no decided bounds, then the Unsat rule is applied. If there are decided bounds, then B has the form $B_0 b_1 B_1 \cdots b_n B_n$ for some $n > 0$, where b_i are all the decided bounds. Since $S \cup B$ is unsatisfiable (remember that any bound can be expressed as a one variable constraint), then due to lemma 1 $S \cup \{b_1 \cdots b_j, b_i\}$ is unsatisfiable. Consider any i in $1 \cdots n$ such that $S \cup \{b_1 \cdots b_i\}$ is unsatisfiable, and any j in $0 \cdots i-1$ such that $S \cup \{b_1 \cdots b_j, b_i\}$ is also unsatisfiable. We will show that we can perform a backjump to decision level j:

Let K be the set $\{\neg b_1, \neg b_2, \cdots \neg b_j\}$, and note that $B = B' b_j B''$. Then we can apply backjump to $B \parallel S$ as $B' b_j B'' \parallel S \Rightarrow B' \neg b_i \parallel S$ because the set $K \cup \neg b_i$ meets the conditions for the Backjump rule:

1. Since all $b_1 \cdots b_j$ are true except for b_i, $S \cup K \cup \neg b_i \models_B b_i$.

2. Since b_i was a decision, it was a relevant bound. It is easy to see that the negation of a relevant bound is also relevant: suppose the bounds for a variable x are $a \leq x \leq b$ in a certain point. Then, the decision has the form $b_d = x \geq c$, and since the decision is a relevant bound, $c > a$. Then, $\neg b_d = x \leq c - 1$, and then since $c > a$, $\neg b_d = x \leq a$ (at most). Since $a < b$ because the variable was not yet assigned, the bound is relevant. The same reasoning is applied to prove the case of a decision on an upper bound.

3. The variable in b_i occurs in some constraint in S because it was decided, and to apply de Decide rule it must occur in a constraint of S.

10.3 Short problem example

$$B = \{x \geq 2,\ x \leq 7,\ y \geq -2,\ y \leq 3,\ z \geq 0,\ z \leq 0\}$$
$$S = \{C = x + y \leq 0,\quad D = 2x - z + 4 \leq 0,\quad E = 3x \leq 0\}$$

In this example we have 3 constraints, 3 variables and a set of initial bounds. Note how all variables are bounded.

10.4 Propagation of bound refinements

Any constraint of the form $a_1x_1+\cdots+a_nx_n+k \leq 0$ implies (entails) a bound for each one of its variables x_i. However, propagation only occurs if this bound is stronger than the best bound in B. Given a variable x_i with a coefficient a_i, we can calculate the implied bound using the following expressions. We will differentiate two cases depending on the sign of a_i. In what follows, let $q = p - (a_i\ x_i)$:

- $\boldsymbol{a_i > 0}$**.** Suppose we have $C = a_1x_1 + a_2x_2 + \cdots + a_nx_n + k \leq 0\ =\ p \leq 0$. We will find the implied bound for the variable x_i. To do this, consider the lower bound of the rest of the polynomial and the upper bound of a_ix_i:

$$lower(q) + upper(a_i\ x_i) \leq 0 \quad =$$

$$lower(q) + a_i\ upper(x_i) \leq 0 \quad =$$

$$a_i\ upper(x_i) \leq -lower(q) \quad =$$

$$upper(x_i) \leq \frac{-lower(q)}{a_i} \quad =$$

$$upper(x_i) \leq \left\lfloor \frac{-lower(q)}{a_i} \right\rfloor =$$

$$\text{Implied upper bound} = \left\lfloor \frac{-lower(q)}{a_i} \right\rfloor$$

- $\boldsymbol{a_i < 0}$**.** Same reasoning as before is applied:

$$lower(q) + upper(a_i\ x_i) \leq 0 \quad = {}^{1}$$

$$lower(q) + a_i\ lower(x_i) \leq 0 \quad =$$

$$lower(q) \leq -a_i\ lower(x_i) \quad =$$

$$\frac{lower(q)}{-a_i} \leq lower(x_i) \quad =$$

$$\frac{-lower(q)}{a_i} \leq lower(x_i) \quad =$$

$$\left\lceil \frac{-lower(q)}{a_i} \right\rceil \leq lower(x_i) \quad =$$

$$\text{Implied lower bound} = \left\lceil \frac{-lower(q)}{a_i} \right\rceil$$

[1] (because $a_i < 0$)

As we can see, both expressions are almost the same. The only difference is that we take the ceiling function for the lower bound (the next integer in the direction of negative infinity), and the floor function for the upper bound (the next integer in the direction of positive infinity). The idea behind these expressions is that when every other monomial has its minimal value, we can derive information about the maximal possible value a single variable needs to have in order to satisfy the constraint. Regardless of the sign of the coefficient, if the implied bound is stronger than the best bound in B, then this constraint can propagate a new bound refinement, advancing the search procedure.

10.4.1 When does a constraint propagate?

We have seen how to compute implied bounds for variables. However, most of the time the implied bounds will not be stronger than the ones we already have. We will now find the condition for a constraint to propagate a stronger bound on a variable. We will use our last result:

- $\boldsymbol{a_i > 0}$. In order for the implied bound $\frac{-lower(q)}{a}$ to be stronger than the current bound $upper(x)$, this must be true:

$$upper(x_i) > \frac{-lower(q)}{a_i}$$

  This means that for the implied upper bound to be stronger, it must be strictly lesser than the current upper bound. We will work from here to get a condition in terms of the polynomial p and the monomial $a_i\ x_i$:

$$upper(x) > \frac{-lower(q)}{a_i} \quad =$$

$$a_i\ upper(x_i) > -lower(q) \quad =$$

$$a_i\ upper(x_i) + lower(q) > 0 \quad =$$

$$a_i\ upper(x_i) + lower(p - a_i\ x_i) > 0 \quad =$$

$$a_i\ upper(x_i) + lower(p) - lower(a_i\ x_i) > 0 \quad =$$

$$a_i\ upper(x_i) + lower(p) - a_i\ lower(x_i) > 0 \quad =$$

$$lower(p) + a_i\ (upper(x_i) - lower(x_i)) > 0 \quad =$$

$$lower(p) + |a_i|\ (upper(x_i) - lower(x_i)) > 0$$

- $\boldsymbol{a_i < 0}$. In order for the implied bound to be stronger, this must be true:

$$lower(x_i) < \frac{-lower(q)}{a_i}$$

This means that for the implied lower bound $\frac{-lower(q)}{a_i}$ to be stronger, it must be strictly greater than the current lower bound $lower(x)$. We will again work from this point to get a condition in terms of the polynomial p:

$$lower(x) < \frac{-lower(q)}{a_i} \quad =$$

$$lower(x_i) < \frac{lower(q)}{-a_i} \quad =$$

$$-a_i \; lower(x_i) < lower(q) \quad =$$

$$0 < lower(q) + a_i \; lower(x_i) \quad =$$

$$0 < lower(p - a_i \; x_i) + a_i \; lower(x_i) \quad =$$

$$0 < lower(p) - lower(a_i \; x_i) + a_i \; lower(x_i) \quad =$$

$$0 < lower(p) - a_i \; upper(x_i) + a_i \; lower(x_i) \quad =$$

$$0 < lower(p) + a_i(lower(x_i) - upper(x_i)) \quad =$$

$$0 < lower(p) - a_i(-lower(x_i) + upper(x_i)) \quad =$$

$$0 < lower(p) + |a_i| \; (upper(x_i) + lower(x_i))$$

As we can see, both expressions are the same. Given a constraint A, if there exists a variable x such that this expression is true, then A implies a stronger bound on x, regardless of the sign of x. This expression will be useful later in order to optimize the detection of possible propagations.

10.5 A short satisfiable example

$$B = \{x \geq 3, \quad x \leq 7, \quad y \geq 2, \quad y \leq 5, \quad z \geq -10, \quad z \leq 10\}$$
$$S = \{C = 2x + y - 8 \leq 0, \quad D = y - z + 2 \leq 0\}$$

We first check if any constraint can propagate. Lets take the first constraint, $A = 2x + y - 8 \leq 0$. As we have seen, this constraint implies two upper bounds on the variables x and y:

$$upper(x) = \frac{-lower(p) + lower(2x)}{2} = \frac{-(6 + 2 - 8) + 6}{2} = 3$$
$$upper(y) = \frac{-lower(p) + lower(y)}{1} = \frac{-(6 + 2 - 8) + 2}{1} = 2$$

Since both of the implied bounds are stronger than the best ones in B, we can add them to the set:

$$B = \{x \geq 3, \quad x \leq 7, \quad y \geq 2, \quad y \leq 5, \quad z \geq -10, \quad z \leq 10, \quad x \leq_C 3, \quad y \leq_C 2\}$$

We continue checking for propagations in the rest of constraints. The constraint $D = y - z + 2 \leq 0$ implies $upper(y) \leq 8$, but this bound is weaker than the one we had, so we discard it. This constraint also implies a lower bound for z $lower(z) \geq -4$. This bound is stronger than the one we had before, advancing the search. The new set of bound refinements is the following:

$$B = \{x \geq 3, \quad x \leq 7, \quad y \geq 2, \quad y \leq 5, \quad z \geq -10, \quad z \leq 10, x \leq_C 3, \quad y \leq_C 2, \quad z \geq_D -4\}$$

Since nothing else can be propagated, we have to make a decision. Both x and y are assigned, since their upper and lower bounds are the same. The only variable we can decide on is z, and we will decide its upper bound to be the same as its lower bound:

$$B = \{x \geq 3, \quad x \leq 7, \quad y \geq 2, \quad y \leq 5, \quad z \geq -10, \quad z \leq 10, \quad x \leq_C 3, \quad y \leq_C 2, \quad z \geq_D -4, \quad z \leq_d -4\}$$

This decision does not propagate anything, nor makes any constraint false. Since every variable is assigned, we can say this problem is satisfiable and the model is the following:

$$x = 3, \quad y = 2, \quad z = -4$$

10.6 Non-termination with unbounded variables

We have imposed the restriction that all variables must be bounded, but we have not shown why. Bounds are needed to stop potentially infinite chains of propagations. For example, consider this problem:

$$S = \{C = -x + y + 1 \leq 0, \quad D = -y + x \leq 0, \quad E = -y \leq 0\}$$

Starting with the initial bound propagated by E, $y \geq 0$, C propagates $x \geq 1$, but D propagates $y \geq 1$. We can now start an infinite chain of propagations:

$$B = \{y \geq_E 0, \quad x \geq_C 1, \quad y \geq_D 1, \quad x \geq_C 2, \quad y \geq_D 2, \quad x \geq_C 3, \quad y \geq_D 3, \cdots\}$$

Informally, x must be at least one more than y, due to C. However, y must be at least as big as x. This two constraints keep propagating bigger and bigger bounds at every step. The fact that variables are unbounded makes this an infinite loop of propagations. Because of this we have imposed the restriction of having bounded variables to guarantee termination.

11 Implementation

We have seen that our method works: it is correct and the algorithm finishes if all the variables are bounded. Implementing the algorithm is pretty straightforward; in this section we will explain how we have implemented it, and additional optimizations we made to improve performance.

11.1 Constraints

Constraints are very easy to implement: using STL containers, we can implement a constraint as a vector of Monomials. A Monomial is just a pair <coefficient, variable>. Monomials are sorted in a constraint by variable IDs to make efficient many queries and operations (finding and removing monomials, or adding two constraints).

11.1.1 Potential overflows

Doing successive operations on a constraint can end up causing overflows. To prevent this we have limited the absolute value of coefficients to 31 bits. When performing operations on constraints, we do the calculations treating the numbers as 64-bit integers. After this process, if the final coefficients can fit in 31 bits we continue the process; if they can not, we have detected an overflow and we can take convenient measures.

11.2 Constraint database and occur lists

The constraint database stores all the information related to constraints and the mechanisms to detect propagations efficiently.

11.2.1 Constraint storage

We know that most of the runtime is spent during propagation and, during this period, constraints are visited sequentially. For this reason, we need to store constraints in a way that reduces caché faults as much as possible. We decided to store all constraints as a contiguous memory block: when a constraint is read, the following ones can be loaded onto the caché, thus reducing caché fault time considerably.

Constraints are stored as an array of integers like this:

$$\text{activity - lemma mark - numMonomials - } a_1 \text{ - } x_1 \text{ - } \cdots \text{ - } a_n \text{ - } x_n$$

This method reduces caché fault time as much as possible and avoids copying full constraints when we are interested in only a few fields (for example, the ith monomial of a constraint).

11.2.2 Occur lists

We can greatly improve the propagation using *occur lists*; when a bound is refined, new propagations can only occur in constraints that contain the variable its bound was refined. The occur list of a variable is a list that contains all the constraints that contain it.

In our case, occur lists contain pairs of the form $< constraintID, c >$, where $constraintID$ is the position of a small header that stores useful information about the constraint, and c is the coefficient that appears with the variable in the constraint. For example, the next constraint gives us the following occurList:

$$C = 3x - 2y + z - 4 \leq 0$$

$$occList(x) =< 1, 3 >$$

$$occList(y) =< 1, -2 >$$

$$occList(z) =< 1, 1 >$$

The c field is used to detect more efficiently if a constraint propagates.

11.3 The Model

The model is one of the most important elements in Intsat. It contains the current variable bounds at all times and the sequence of bounds (the model stack), among others.

11.3.1 Model stack

The model stack is the direct implementation of the sequence of bounds B. As the name hints, we have implemented it as a stack of StackElements. A StackElement is an abstract representation of a bound. A StackElement contains a variable and a new value for its bound (this is the information related to the bound refinement). It also contains the following fields:

- **Pointer to last refinement of the same type**. This is useful to find all bounds for a variable, or check what the best bound of a variable was in a certian decision level. It is also used to restore values during backjumps.

- **Current decision level**. Every element has an integer that holds the decision level it can be found. This information is redundant, but it simplifies some algorithms.

11.4 Cleanups

A *conflict* arises when a constraint becomes false. As we said in the main procedure, this results in a backjump. Intsat can learn lemmas after a conflict. Remember that a lemma is a constraint that results from a conflict and, had the lemma been present earlier, the conflict would have been completely avoided.

This means that a lemma is generated after every conflict: lemma learning results in a large number of stored lemmas that may not be useful during the rest of the search. Cleanups try to solve this problem: when some conditions are met, the least used lemmas are deleted to save space. To be able to do this, Intsat saves an activity factor for each lemma. During a cleanup, all lemmas that have 0 activity get deleted, and llemas that survive have their activity divided by 2. During the search, a lemma's activity is increased by 1 if it takes part in a conflict. New lemmas have their activity initialized to 7, making them "survive" 3 cleanups if they do not take part in conflicts.

There is one more thing done during cleanups: variables that are assigned can be removed from the constraints, in the spirit of classic SAT (when a variable is set to false at decision level 0, it can be removed from all the clauses since it will not affect their truth value). In our case, variables that are already assigned can be eliminated adding their value to the constant part of the constraint:

$$C = 3x + 4y - 2z + 13 \leq 0 \quad , \quad x = 2 \quad \Rightarrow$$

$$C = 4y - 2z + 19 \leq 0$$

The conditions for a cleanup are met when the number of lemmas exceeds a certain threshold (that can be set as a parameter), and when the decision level of B is 0.

11.5 Heuristics and strategy

There are certain points in the search process in which we need to take some decisions that can greatly affect the whole process. For example, how do we find which variable should we decide? How do we compute the value of the bound refinement of a decision?

11.5.1 Next decision variable

We use a simple priority queue to store variables. As their key, we use an activity value (similar to lemma activity). This activity is increased when a variable takes part in a conflict, and the activity value is increased following a geometric progression. The speed of this activity increase can be also adjusted as a parameter. When a certain activity threshold is reached, all activities are normalized to avoid overflows. This activity heuristic translates to giving variables which appear in many conflicts a lot of importance; furthermore, it also gives more importance to variables which appear in later conflicts during the search.

11.5.2 Value of decided bounds

Once we know which variable are we making a decision on, we need to decide its new bound. We have chosen to do this arbitrarily and decide the lower bound to be equal to the mid point between lower and upper bound (thus removing half of the possible values with a single decision).

11.5.3 Restarts

There are different ways to do restarts: Intsat can do restarts using a nested strategy, or a luby number based strategy. Each of them can be easily selected.

Nested restarts work using two thresholds: the innerBoundRestart and the outerBoundRestart. A restart occurs when the number of conflicts since last restart is larger than the innerBoundRestart. After this restart, the innerBoundRestart is increased following a geometric progression. This event is repeated until the innerBoundRestart gets larger than the outerBoundRestart. At this point, outerBoundRestart is increased, and innerBoundRestart is set to its initial value.

Luby number based restarts work by using a sequence defined the following way:

$$luby(x) = \begin{cases} 2^{log_2(n+1)-1} & \text{if x is a power of 2} \\ luby(n+1-2^{log_2(n)}) & \text{otherwise} \end{cases}$$

This definition translates to a sequence of the form 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8... Once we have this sequence, we can compute the luby number associated to the number of restarts we have done, and we multiply it by a certain constant (which can be tweaked) to have the number of conflicts needed for the next restart.

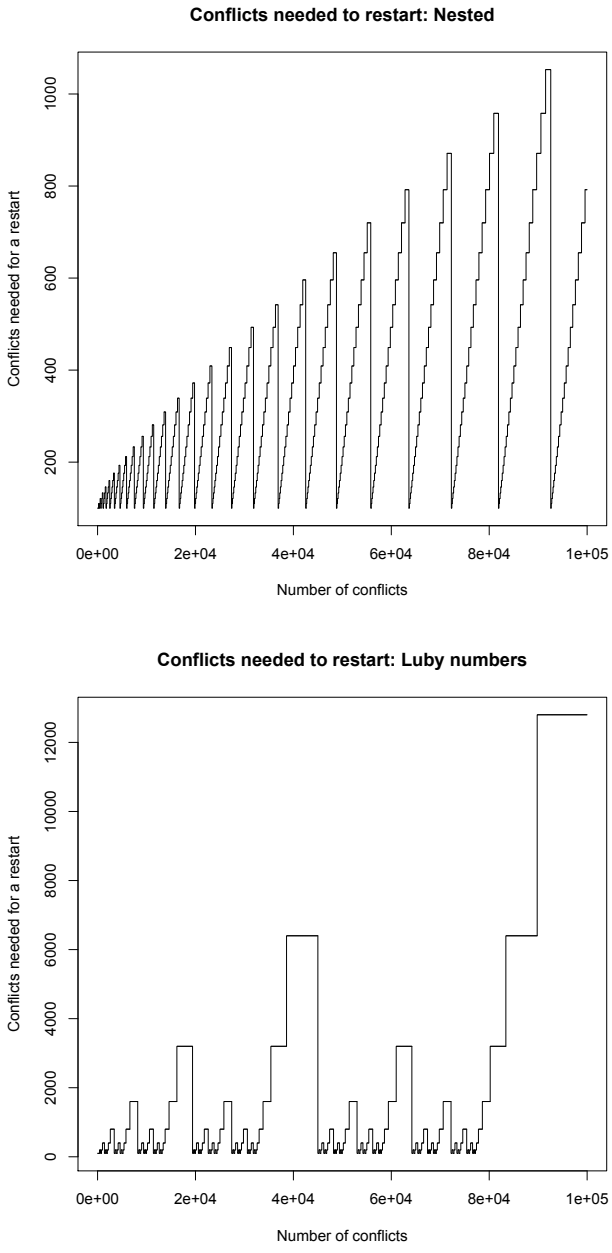

Figure 1: Conflicts needed for each method to do a restart vs the total number of conflicts.

12 Experimental results

We have tested our solver against other programs that solve integer linear arithmetic: cutsat and CPLEX.

In order to evaluate Intsat's performance, we have used random problems and real problems taken from the MIPLIB 2003 library. These benchmarks can be accessed from cutsat's website [7]. There are several sets of problems:

- **Pigeon hole principle problems**. These are a series of unsatisfiable problems that encode the pigeon hole principle.
- **Primes**. These problems encode a tight cone around an n-dimensional point of the first n prime numbers.
- **MIPLIB2003**. These problems are taken from the miplib2003 library.
- **Slacks**. These problems were used in the "Cuts from proofs" paper by Isil Dillig, Thomas Dillig and Alex Aiken[9].
- **Random problems**. This is a set of randomly generated problems.

All the problems have been tested on a Intel Core i5 (2.67GHz) with 8GB of RAM memory and executions have been limited to 600 seconds. To measure time, instead of using the time reported by each program, we have used the UNIX command **time** to do our measurements. This command gives us several times; we have used the sum of CPU time in seconds for both user and kernel mode. This is so because there may be other processes running at the same time in the machine, and having a single measurement method and read only the CPU time for our own process will give us the best results.

However, since we are invoking CPLEX using a script and some previous work, CPLEX times are expected to be a bit higher than their true value. In some cases we sill also add the time reported by CPLEX itself, but in bigger sets we will not be including it for convenience.

As a final note, we are solving instances taken from the web. Many of these instances do not meet our requirements: they have unbounded variables. To be able to compare solvers, we have set arbitrary bounds ($-1024...1024$) for any unbounded variables. This may hurt Intsat's performance a little, but it is worth it because it lets us compare our solver with the other two.

12.1 Pigeon hole problems

Our first test will deal with problems that encode the pigeon hole principle. The pigeon hole principle states that if n items are put in m holes and $n > m$, then at least one of the holes must contain more than one item. This principle is translated into a series of pseudo boolean (unsatisfiable) problems. The resulting times (in milliseconds) are the following:

Problem	Intsat	cutsat	CPLEX	CPLEX (reported)
pigeon-hole-2	10	10	30	0
pigeon-hole-3	10	10	30	0
pigeon-hole-4	10	10	20	0
pigeon-hole-5	10	20	40	0
pigeon-hole-6	10	10	40	10
pigeon-hole-7	10	10	40	10
pigeon-hole-8	20	10	40	10
pigeon-hole-9	20	10	40	10
pigeon-hole-10	20	10	40	10
pigeon-hole-11	30	10	40	10
pigeon-hole-12	30	10	40	10
pigeon-hole-13	40	10	40	10
pigeon-hole-14	40	30	20	10
pigeon-hole-15	50	20	40	10
pigeon-hole-16	80	20	40	10
pigeon-hole-17	100	20	40	10
pigeon-hole-18	110	20	40	10
pigeon-hole-19	140	20	50	10
pigeon-hole-20	180	30	40	10
total time	920	290	710	160

Table 1: Times (in ms) for the pigeon-hole set of problems.

As we can see, Intsat has the worst performance of them all in this set. Its times are similar to CPLEX's, but when we look at the real time spent by CPLEX we will see that Intsat is actually far away from the other two solvers.

12.2 Prime cones

Our second set of problems corresponds to problems that encode a tight cone around an n-dimensional point of the first n prime numbers. They consist in around 20 SAT and 20 UNSAT problems. The results have been the following:

Problem	Intsat	cutsat	CPLEX	CPLEX (reported)
prime_cone_sat_2	10	10	30	0
prime_cone_sat_3	10	10	40	10
prime_cone_sat_4	20	10	40	10
prime_cone_sat_5	10	10	40	10
prime_cone_sat_6	10	20	40	10
prime_cone_sat_7	10	20	40	10
prime_cone_sat_8	10	20	40	10
prime_cone_sat_9	10	30	50	10
prime_cone_sat_10	10	30	40	10
prime_cone_sat_11	10	40	40	10
prime_cone_sat_12	20	40	40	10
prime_cone_sat_13	20	60	50	10
prime_cone_sat_14	20	60	50	10
prime_cone_sat_15	20	80	50	10
prime_cone_sat_16	30	100	40	10
prime_cone_sat_17	30	130	40	10
prime_cone_sat_18	20	170	40	10
prime_cone_sat_19	30	210	40	10
prime_cone_sat_20	20	250	50	10
total time	320	1300	800	180

Table 2: Times in ms for the SAT prime-cone set of problems.

Problem	Intsat	cutsat	CPLEX	CPLEX (reported)
prime_cone_unsat_3	10	10	40	10
prime_cone_unsat_4	10	10	50	10
prime_cone_unsat_5	10	10	50	10
prime_cone_unsat_6	10	20	60	10
prime_cone_unsat_7	10	20	70	20
prime_cone_unsat_8	10	20	60	10
prime_cone_unsat_9	10	30	50	20
prime_cone_unsat_10	10	30	60	10
prime_cone_unsat_11	30	40	60	20
prime_cone_unsat_12	20	40	70	20
prime_cone_unsat_13	20	50	60	20
prime_cone_unsat_14	20	70	50	20
prime_cone_unsat_15	20	70	50	20
prime_cone_unsat_16	30	110	50	20
prime_cone_unsat_17	30	130	50	20
prime_cone_unsat_18	30	170	50	20
prime_cone_unsat_19	30	210	50	20
prime_cone_unsat_20	40	250	60	30
total time	350	1290	990	310

Table 3: Times in ms for the UNSAT prime-cone set of problems.

Results have been more positive in this set of problems. We can see that cutsat starts to struggle with the bigger problems, but Intsat and CPLEX increase their times at a much slower rate. Particularly, in the UNSAT problem set, Intsat's performance is quite similar to CPLEX's.

12.3 MIPLIB2003's problems

These problems have been taken from the Mixed Integer Problem Library (MIPLIB) which was last updated in 2003. The set we are using can be downloaded from cutsat's website, and has the optimization constraints removed. Here are the results:

Problem	Intsat	cutsat	CPLEX
air04.sat		0.26	6.46
air05.sat	15.51	0.19	0.30
cap6000.sat	30.71	7.18	0.06
disctom.sat	30.66		0.74
ds.sat		5.71	2.28
fast0507.sat	3.42	1.43	0.23
harp2.sat	0.08	2.67	0.05
manna81.sat	0.14	0.07	0.05
mzzv11.sat	2.22	0.29	1.34
mzzv42z.sat	5.05	0.58	1.80
nw04.sat			1.38
p2756.sat	0.10	0.22	0.12
protfold.sat	0.21	1.28	10.10
seymour.sat	0.22	0.05	0.05
sp97ar.sat	2.24	14.89	0.42
stp3d.sat			
solved	12	13	15
total time	90.56	34.82	25.38

Table 4: Times in s of the MIPLIB2003 problem set.

12.4 Slacks

These problems have been used in the "Cuts from proofs" paper, and they are available for download from cutsat's website. There are 251 problems in total, and we have tested as many as we have been able to. Both Intsat and CPLEX have been able to solve most of the problems, but cutsat has been struggling with many problems:

solver	instances solved	total time (in s)
Intsat	216	2055.07
cutsat	172	3004.00
CPLEX	250	13.19

Table 5: Time results for the "Slacks" set of problems.

The performance of CPLEX is again quite remarkable. Our solver, Intsat, has had some trouble solving a small subset of the instances. However, cutsat has been the one who has had the worst performance.

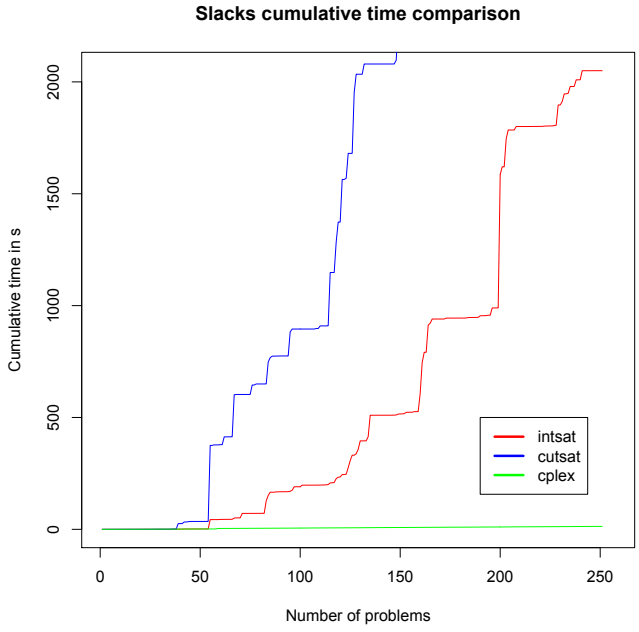

Figure 2: Performance comparison in Slacks problem set.

12.5 Random problems

We have used a random problem generator to create several random problems. As input, the generator asks for the number of variables, the number of constraints and a random seed. Parameters such as minimum and maximum bounds, or the number of variables per constraint can also be adjusted. The first set of 100 problems has 50 variables and 100 constraints:

solver	instances solved	total time (in s)
Intsat	100	2.54
cutsat	93	1435.29
CPLEX	100	11.18

Table 6: Time results for random problems.

The performance of cutsat with random problems is very poor, compared to the other solvers. The reason of this is because cutsat calculates "tight reasons" for each bound refinement. These reasons are used when a conflict arises to calculate where to backjump. However, when a solver learns a lemma from a conflict, this lemma is useful to avoid similar conflicts to the one that produced it. Random problems do not have this property: learning is almost next to useless in a random problem.

12.6 Restart method comparison

In this last section we will compare Intsat against Intsat itself. We want to see how Intsat behaves with different restart schemes. Our tests will be performed on the Slacks problem set, because it is a fairly big problem set with some hard problems. We already have an execution of insat working with Luby restarts on it and it took around 2000 seconds. We will now solve the same batch of problems with Intsat working with nested restarts:

solver	instances solved	total time (in s)	average time (in s)
luby restarts	216	2055.07	9.51
nested restarts	217	1475.03	6.80

Table 7: Time results for Intsat with luby or nested restarts

Nested restarts have proved to work better than luby restarts, at least in this problem set. With nested restarts, Intsat has been able to solve one more problem and it has been able to solve the whole batch in less total time. Here is a graphical comparison of both versions performance:

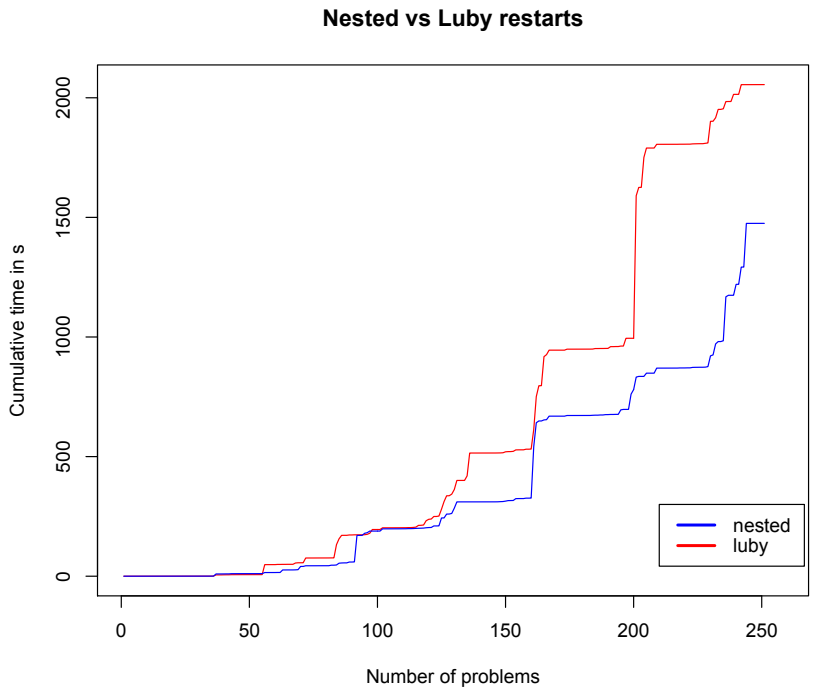

Figure 3: Performance comparison between different restart schemes.

13 Future work

Our solver, Intsat, is already working and it is giving positive results. However, there are a few things we have not done and would be interesting to investigate in a future:

- **Bug fixes**. We have found some execution errors while we were testing Intsat and comparing it to the other solvers. These errors are probably related to the problem parsing stage, but it is a problem that needs to be solved as soon as possible.

- **New heuristics**. There are many things left to try: new restart strategies, different decision procedures... We have tried a few, but there are many more things to test and evaluate.

- **Parallelization**. This is something we have completely ignored in this project. We have worked being concious of the architecture underneath, but we have not thought about parallelizing Intsat. This means examining the whole process to find suitable parallelization points, and adapting the most we can to be able to use more than one processor.

- **Unbounded variables**. Intsat only works with problems that have correctly bounded variables. This is temporarily addressed setting an arbitrary bound to variables, if they are missing it. However, there are different strategies such as adding "control variables" that refer to maximum and minimum bounds of unbounded variables, and have the process work automatically.

14 Planning and budget

Arrived to this point, we must ask ourselves: was the project successful? Did we accomplish the objectives and did we follow our planning? The answer to those questions is yes.

Our planning consisted in two week developement phases in which each one of them we would develop a core element of our solver. This has been the case: we started with the basics, and we kept adding things or improving the already existing code. We have reached the end of this project and we have a working SAT solver for integer linear programming problems. Furthermore, as we have seen in the results section, Intsat's performance has been generally better than cutsat's. There are, of course, some exceptions, but the experiments have given us some positive results. With all of this, we can safely state that we have accomplish our first objective: build a working solver at least as good as cutsat. Our second objective has been accomplished along the way: our solver is quite modifiable and easy to experiment on.

The project has also been successful regarding to the budget. The hours spent in this project have been roughly the ones we predicted, and since the planning has been met, our budget requirements have also been met.

15 Sustainability and social responsability

The developement of Intsat is not only theory oriented. Of course research is the main goal, but this software has many applications. Many critical problems are in fact integer linear programming problems which could be solvable by Intsat. We have seen that cplex is a powerful alternative, and actually works better than Intsat. However, Intsat is only the beginning. It has been a start point to explore new ways to solve integer linear programming problems. In a future, Intsat could achieve cplex's performance, or even surpass it.

With this potential, Intsat could solve many industrial, business and engineering problems. It could solve complicated logistics problems, reducing production costs, or risks for workers. In conclusion, Intsat is a small piece of software, but capable of many things yet to see.

16 Conclusion

In this project we have built a fully functional SAT solver for integer linear problems from scratch. Our idea was to start with the DPLL algorithm, which we know it works, and extend it to support integer variables.

We started with a simple planning: we would began with a basic solver and we would extend it. Every two weeks we would implement a new optimization, or a core module. This planning, as well as the budget requirements, have been met. In this sense, the project has been successful. We have also met our two main objectives: we have built a solver with a certain degree of efficiency which is at least as good as cutsat, and this solver can be tweaked and played with to experiment different behaviours and learn from them.

We have gone through all the details in how our algorithm works, and we have explained how we have implemented every single one of them. Furthermore, we have shown possible optimizations to reduce runtime and save work. These optimizations include both data structures and algorithms.

We have tested Intsat and compared it against two more solvers: cutsat and CPLEX. We have seen that Intsat's performance is not bad at all: it has surpassed cutsat's performance in some problem sets, and the overall results have been quite positive. We have also included a test comparing two different restart schemes for Intsat.

In conclusion, this project has been a success. It represents the end of many months of work, and although this project ends here, there are many things left to do. Proof of this is the full "Future work" section, which gives a list of things that I want to keep working on. I have learned a lot with this project, and I sincerely hope other people can benefit from all the work we have done here.

17 References

[1] H. Paul Williams. Springer. Logic and integer programming. springer, 2009.

[2] Eugene C. Freuder. The constraints journal. "http://www.springer.com/computer/ai/journal/10601", April 1997.

[3] Roman Barták. Foundations of constraint programming. European Joint Conferences on Theory and Practice of Software, 2003.

[4] Stephen Cook. The complexity of theorem proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[5] M. Davis, G. Logemann, D. Loveland. A machine program for theorem-proving. *Communications of the ACM, Vol 5, No. 7*, pages 394–397, 1962.

[6] Ron Shamir. The efficiency of the simplex method: a survey. *Management Science, Vol 33, No. 3*, pages 301–304, 1987.

[7] Dejan Jovanović. cutsat. http://www.cs.nyu.edu/ dejan/cutsat/.

[8] Robert Nieuwenhuis, Albert Oliveras and Cesare Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories.

[9] Isil Dillig, Thomas Dillig and Alex Aiken. Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers.