Master Degree in Computer Science
Final Thesis

Linear Integer Optimization with SMT

Relatore/1st Reader: Prof. Roberto Sebastiani
Università degli Studi di Trento

Laureando/Graduant: Patrick Trentin

Controrelatore/Outside Examiner:
Prof. Luigi Palopoli
Università degli Studi di Trento

Anno Accademico 2013–2014
## CONTENTS

6 \( \mathcal{L}A(\mathbb{Z})\)-Solver optimization extension 41
   1 Basic Branch and Bound ............................................. 41
   2 Advanced Branch and Bound ......................................... 44
   3 Truncated Branch and Bound ......................................... 51

7 Performance Evaluation 55
   1 Test-bed setup .......................................................... 55
   2 Empirical Results ........................................................ 59

8 Conclusions and further work 73

Appendix 79
   A Model Encoders ....................................................... 79
Chapter 1

Introduction

Boolean satisfiability (SAT) is the problem of determining if there exists a (possibly partial) truth assignment for the variables within a propositional formulae that makes it evaluate to true, satisfying it. A complete, backtracking-based, efficient algorithm for deciding satisfiability of propositional logic formulas in conjunctive normal form (CNF), known as DPLL, was first introduced in [Davis, Logemann, and Loveland, 1962]. Modern SAT solvers implement a variant of DPLL that uses conflict-driven clause-learning (CDCL) [Silva and Sakallah, 1996], which speeds up the search through non-chronological backtracking, that is often referred to as back-jumping.

Satisfiability Modulo Theories (SMT) [Barrett et al., 2009b] is the problem of deciding the satisfiability of first-order formula with respect to a decidable first-order theory $T$ with equality ($\text{SMT}(T)$). SMT is applied to Equality and Uninterpreted Functions ($\text{EUF}$), Linear Arithmetic ($\text{LA}$) over the Rationals ($\text{LA}(\mathbb{Q})$) or the Integers ($\text{LA}(\mathbb{Z})$), theory of arrays ($\text{AR}$), theory of bit-vectors ($\text{BV}$), and any of their combinations. Nowadays most successful SMT solvers adopt the so called lazy approach which combines an efficient SAT solver, enumerating truth assignment which satisfy a Boolean abstraction of the input formula, with specialized $T$-Solvers, checking the consistency in $T$ of the set of literals corresponding to the enumerated assignments. Many problems of industrial interests can be encoded in SMT, e.g. resource planning and temporal reasoning, automated reasoning, formal verification of proof obligations in software systems, compiler optimization and real-time embedded systems. Also, some problems
that can be encoded in SAT, like microprocessor datapaths modelling and verification, are better encoded in SMT due to the higher level of abstraction it provides.

In the last years, work has been done to extend SMT solvers to deal with optimization problems [Nieuwenhuis and Oliveras, 2006; Cimatti A. and C., 2010]. In particular, [Sebastiani and Tomasi, 2012] introduced the concept of Optimization Modulo Theory (OMT), the problem of finding a Boolean assignment for an input formula \( \varphi \) that minimizes a cost function defined over arithmetical variables. This research work resulted in OPTIMATHSAT, a fork project of MATHSAT5 Cimatti et al., 2013, that is the first SMT solver capable of handling optimization over Linear Arithmetic with Rational cost functions (OMT(LA(Q))). This has many practical applications, since many SMT-encodable problems of interests may require the capability of finding models that are optimal wrt. some continuous arithmetical variable. E.g., in resource planning a plan for achieving a certain goal might not only be required to fulfil some resource constraints but also to minimize the usage of some of such resources; in SMT-based model checking of timed or hybrid systems it allows to find executions that minimize some parameter while fulfilling/violating some other properties of the system.

Many problems that traditionally belonged to other research communities, e.g. Constrained Programming (CP), Linear Programming (LP), Linear-Generalized Disjunctive Programming (LGDP) can now be encoded as SMT or OMT(LA(Q)) problems.

The main goal of this thesis is to take this research work one step further, and study the topic of linear arithmetic optimization with integer cost functions (OMT(LA(Z))). These kind of problems are traditionally solved with Integer Linear Programming (ILP) or Mixed Integer Linear Programming (MILP) techniques. However, one of the drawbacks of these solvers is the lack of a natural encoding for Boolean variables, usually mapped into Integer ones, and the inefficient handling of disjunctive reasoning. These two characteristics are on the other hand the main strengths of SMT-based solvers, which means that there is concrete chance that on certain categories of ILP/MILP problems our new OPTIMATHSAT implementation outperforms ILP-based solvers. It is needless to say that OMT(LA(Z)) has many practical applications, many of which retrace those of OMT(LA(Q)) restricted to the Integer domain. For example, resource planning problems involve assets that are naturally encoded into discrete variables with an indivisible unitary value (e.g. human resources, discrete time units, no. of items).
As part of the work of this thesis, OPTIMATHSAT is extended so that it now supports both $OMT(\mathcal{L}A(\mathbb{Q}))$ and $OMT(\mathcal{L}A(\mathbb{Z}))$ input problems, using some techniques identified with a preliminary research on the field. The solver now able to take as input a pair $\langle \varphi, cost \rangle$, s.t. such that $\varphi$ is a $SMT(\mathcal{L}A(\mathbb{Z}) \cup \mathcal{T})$ formula and cost is a $\mathcal{L}A(\mathbb{Z})$ variable occurring in $\varphi$, and return the optimal integral cost value $z$ and its associated solution model $m$. Then, this new software implementation is compared to other competitor CP/MILP solvers, in order to verify its correctness and performances. As we show later on to the reader, OPTIMATHSAT does not perform well against CP/MILP problems when tested on benchmarks taken from the CP research community, though it clearly outperforms any competitor on Bounded Model Checking (BMC) problems taken from its own repository. This sharp difference in performances is hardly clearly identified but, after examining the two input sets of problems, our best is guess that it resides in the different level of integer and disjunctive reasoning of the two input problems and, in particular, in the encoding used. When comparing OPTIMATHSAT to CP solvers it should be noted that the former comes with the additional capability of addressing a wider number of first-order theories combined to integer optimization. Also, where most CP solvers use fixed precision arithmetic, OPTIMATHSAT uses infinite-precision mathematics to guarantee the correctness of a result, despite of the performance hit of up to a third of running time it might cause.

This thesis is divided in two parts.

Part I will review the theoretical background of this work and the current state of the art of the research. It will start by providing the basic notion of integer linear programming in section 2, along to the most important techniques used to solve it. Then the focus will move to lazy satisfiability modulo theories in section 3, with a particular attention to the specific implementation of some MATHSAT5 internal features Cimatti et al., 2013. Finally, section 4 will review the current architecture of the linear arithmetic optimization over the rationals feature implemented within OPTIMATHSAT.

Part II is devoted to this thesis contributions into the development of OPTIMATHSAT in respect to linear arithmetic optimization over the integer variables. Its section 5 describes the CDCL main loop designed for optimization, while section 6 goes into the details of the symplex-based branch and bound integer optimizer added to the $\mathcal{L}A(\mathbb{Z})$-Solver. Section 7 will then verify the correctness and assess the performances of the new implementation. More specifically, the latter will be compared to solvers that belong to
the constrained programming community, which is devoted to linear integer optimization too. The last section 8 will finally draw some conclusions on the work done and on the way to go for future improvements of the current implementation.
Part I

Background and state of the art
In the first part of this thesis work all efforts will be focused on providing a generic reader, with a background in computer science, the necessary foundations for the later parts and a proper understanding of the current state of the research on the topic of linear integer optimization with SMT tools.

Chapter 2 will cover the definition of integer linear programming (ILP) and its formal notation – required throughout the rest of this thesis work. We will then examine the basic concept of cutting planes and look into the details of two of the most important mathematical tools that fall into this category, that is Chvátal inequalities and Gomory cuts. The last topic in this chapter will cover the most widely adopted technique for solving ILP problems, the simplex-based branch and bound approach.

Chapter 3 is devoted to the field of Boolean satisfiability (SAT) and to lazy satisfiability modulo theories (SMT). An initial introductive paragraph will rattle off all the fundamental notions related to logic, SAT and SMT research fields. After that, the general architecture of a DPLL-based theory solver (T-solver) will be described, with particular attention on the techniques most widely used for efficiency and versatility. Then, the focus will move on the specific implementation of the LA(ℤ)-Solver within MATHSAT5 and OPTIMATHSAT, using as reference the research work of Griggio. The motivation of this detailed review of LA(ℤ)-Solver internals comes from the fact that much of the development work presented in part II consisted on extending this T-Solver with optimization techniques.

Finally, chapter 4 will review the research work of Sebastiani and Tomasi on optimization over the theory of linear arithmetic with rational objective functions only (OMT(LA(ℚ))).

The latter chapter will put an end to our review of state of the art technology and background research notions. At this point the reader will possess background knowledge solid enough to deal with part II, in which we will develop our own implementation of optimizer over integer cost objective cost functions.
Chapter 2

Integer Linear Programming

A linear program (LP) is the problem of maximizing or minimizing a linear function over a convex polyhedron specified by linear and non-negativity constraints [Linderoth and Ralphs, 2004]. An integer linear program (ILP) is a PL in which the search space is limited to the lattice of points in $\mathbb{Z}^n$ contained within such convex polyhedron, thus requiring the optimal solution $x^*_ILP$ to be entirely integral.

An ILP is usually represented in its canonic form, using an objective function associated to the optimal value $z_{ILP}$ and a the set of feasible solutions $\mathcal{X}$

$$z_{ILP} = \min_{x \in \mathcal{X}} c^T x,$$

$$\mathcal{X} = \begin{cases} x \in \mathbb{Z}^n \\ Ax \geq b \\ x \geq 0 \end{cases}$$

where the vector of coefficients $c \in \mathbb{Z}^n$ defines the objective function, $b \in \mathbb{Z}^m$ and $A \in \mathbb{Z}^{m \times n}$ is the constraint matrix.

In literature an ILP problem is frequently associated to its LP continuous relaxation which, as the name says, gets rid of the requirement of integrality for $x^*$. Formally, we define it as

$$z_{LP} = \min_{x \in \mathcal{P}} c^T x,$$

$$\mathcal{P} = \begin{cases} Ax \geq b \\ x \geq 0 \end{cases}$$
It is easy to note that $\mathcal{X} \subset \mathcal{P}$, and this implies that $z_{LP} = \min \{ c^T x | x \in \mathcal{P} \} \leq \min \{ c^T x | x \in \mathcal{X} \} = z_{ILP}$. Thus $z_{LP}$ effectively constitutes a lower bound for $z_{ILP}$. It also follows that when the optimal solution $x_{LP}^*$ of a continuous relaxation LP is integral the equality $z_{LP} = z_{ILP}$ holds, because $x_{LP}^* \in \mathcal{P} \cap \mathbb{Z}^n = \mathcal{X}$.

The characterization of an ILP problem is never unique, since the set of feasible solutions $\mathcal{X}$ can be specified using an infinite number of combinations of linear constraints. Ideally, one would like to formulate the ILP problem such that the solution $x_{LP}^*$ of its LP relaxation $\tilde{\mathcal{P}} = \{ x \geq 0 | \tilde{A}x \geq \tilde{b} \}$ is always integral. Then, by simply using the Fourier-Motzkin procedure or the Simplex method [Schrijver, 1986; Dantzig and B.C. Eaves, 1973], two popular methods for deciding linear arithmetic, one would easily obtain the solution of the original ILP problem. Such an ideal characterization of the problem exists and is given by the convex hull $^{\dagger}$ of $\mathcal{X}$. Unfortunately, $\tilde{A}$ and $\tilde{b}$ are not easily determined and $\tilde{\mathcal{P}}$ tend to contain a too large number of linear constraints.

Historically, a number of techniques have been developed to tackle ILP problems, which are in general NP-complete. An in-depth review of these mathematical tools is certainly beyond the scope of this thesis work, you can refer to [Linderoth and Ralphs, 2004] and [Schrijver, 1986] for further informations. Here, I will describe two of the most important cutting planes methods, Chvátal inequalities and Gomory cuts, and the classic branch and bound approach.

1. Cutting Planes

The general idea of the cutting planes procedure is to solve the ILP problem by reducing it to a sequence of progressively-refined LP relaxations $\mathcal{P}_1, ..., \mathcal{P}_n$ through a sequence of cuts. Let $x_{LP}^* \in \mathcal{P}$ be the solution of the LP relaxation of an ILP problem, then a cut is an inequality of the form $\alpha^T x \leq \alpha_0$ s.t.

$$\alpha^T x \leq \alpha_0, \forall x \in \mathcal{X} = \mathcal{P} \cap \mathbb{Z}^n$$
$$\alpha^T x^* > \alpha_0$$

The pseudo code of this procedure is shown in figure 2.1. Initially, the LP relaxation is solved to find an initial solution $x_{LP}^*$ (row 1-2). If no such solution is found, then the

1. The convex hull of a set $S \subseteq \mathbb{R}^n$, $\text{conv}(S)$, is the smallest convex set which contains $S$. 
cutting_planes (matrix A, vector c, vector b)
1. \( lp \leftarrow \min \{ c^\top x | Ax = b, x \geq 0 \} \)
2. \( \langle state, x^* \rangle \leftarrow \text{solve_lp}(lp) \)
3. if (state == unsat) then
   4. return unsat
5. end if
6. while \((x^* \notin \mathbb{Z}^n)\) do
   7. \( \langle \alpha^\top, \alpha_0 \rangle \leftarrow \text{get_cut}(\mathcal{X}, x^*) \)
   8. add_cut_to_lp \(lp, \langle \alpha^\top, \alpha_0 \rangle \)
   9. \( \langle state, x^* \rangle \leftarrow \text{solve_lp}(lp) \)
10. end while
11. return \( x^* \)

**Figure 2.1.** Cutting Planes Algorithm

procedures returns unsat because ILP problem has no solution too since \( \mathcal{P} = \emptyset \land \mathcal{X} = \mathcal{P} \cap \mathbb{Z}^n \implies \mathcal{X} = \emptyset \) (rows 3-5). If \( x^*_{LP} \in \mathbb{Z}^n \) then the solution of the LP relaxation coincides with that of the original ILP problem and is returned at line 11. Otherwise, the algorithm loops in lines 6-10 up until when such condition is verified. The loop iteratively seeks for a cut valid for all feasible solutions in \( \mathcal{X} \) but invalid for \( x^*_{LP} \), adds it to the LP relaxation and solves it again.

**Figure 2.2.** An example of ILP problem on a 2D plane.
Figure 2.2, showing an example of polyhedron on the $x, y$ plane, gives an example of cut – depicted with a dashed line – for the initial ILP problem. In this example, $\mathcal{X}$ is equal to the set $\{(2, 1), (3, 1), (4, 1), (3, 2), (4, 2), (5, 2), (6, 2), (3, 3), (4, 3), (5, 3), (5, 4)\}$, while $\mathcal{P}$ is given by the entire blue area and the convex hull of $\mathcal{X}$ is given by the red one. The red arrow stands for the gradient of the cost function, that is the direction toward which there is the maximum gain by varying the value of the problem variables.

Note that the algorithm termination depends not only on the finiteness of $\mathcal{X}$, but on the quality of the cuts found by the get_cut procedure too. There exist several well known procedures capable of determining such cuts with a various degree of complexity and cut depth, some of which will be here briefly reviewed.

1.1 Chvátal Inequalities

One of the most known methods for obtaining cuts, although seldom used in practice, is to use the so-called Chvátal inequalities [Eisenbrand, 2000]. Though it guarantees the convergence of the cutting plane algorithm by making the n-th LP relaxation $\mathcal{P}_n$ equal to $\text{conv}(\mathcal{X})$, its main disadvantage is to require a considerable amount of steps in respect to other methods.

Let $\mathcal{P} = \{x \geq 0 | Ax = b\}$, take $u \in \mathbb{R}_+^m$ and derive by linear combination the following equation valid for $\mathcal{P}$

$$u^T Ax = u^T b$$

then we can easily see that, since $x \geq 0$, the following weakened inequality holds for $\mathcal{P}$ too

$$\alpha^T x = \lfloor u^T A \rfloor x \leq u^T b$$

where $\forall i \in [1, n]. \alpha_i = \lfloor u^T A j \rfloor = \lfloor \sum_{i=1}^m u_i a_{ij} \rfloor$. Finally, by defining $\alpha_0 = \lfloor u^T b \rfloor$ a valid cut for $\mathcal{X}$ that excludes the current solution $\mathcal{x}_{\text{LP}}^*$ is found

$$\alpha^T x \leq \alpha_0$$

The family of inequalities $\lfloor u^T A \rfloor x \leq \lfloor u^T b \rfloor$, in symbols $A^{(1)} x \leq b^{(1)}$, allows to define the first closure of Chvátal as

$$\mathcal{P}_1 = \{x \geq 0 | Ax = b, A^{(1)} x \leq b^{(1)}\} \subseteq \mathcal{P}$$
By applying this procedure iteratively, one obtains a sequence of polyhedrons $P \supset P_1 \supset \ldots \supset P_n = \text{conv}(\mathcal{X})$. Clearly, in the worst case an integral solution for the original ILP problem is not found up until when the highest rank $n$ is achieved, and this can be huge.

### 1.2 Gomory Cuts

Let $x^*$ be the optimal solution for a certain LP relaxation and $B$ its base. Then, the problem can be reformulated as

$$\begin{align*}
\min & \, \sum_{j \in NB} c_j^T x_j \\
x_B + & \, \sum_{j \in NB} \bar{a}_{hj} x_j = \bar{b}_h, \forall h \in [1, m] \\
x & \geq 0 \\
x & \in \mathbb{Z}^n
\end{align*}$$

If $x^* \notin \mathbb{Z}^n$ then exists and index $k$ s.t. $x^*_k$ is fractional and a row index $h$ for which $x_k$ is in base. The corresponding equation

$$x_k + \sum_{j \in NB} \bar{a}_{hj} x_j = \bar{b}_h, \beta[h] = k$$

is valid for $\mathcal{P}$ and fractional, thus it can be used to generate a Gomory cut through the Chvátal procedure:

$$x_k + \sum_{j \in NB} [\bar{a}_{hj}] x_j \leq [\bar{b}_h], \beta[h] = k$$

By construction this cut is valid for $\mathcal{X}$ and it’s easy to verify that it is violated by $x^*$:

$$\left( x_k + \sum_{j \in NB} [\bar{a}_{hj}] x_j \right)_{x=x^*} = x^*_k - \bar{b}_h > [\bar{b}_h], \beta[h] = k$$

### 2. Branch and Bound

The LP-based branch and bound is a divide-and-conquer algorithm that reduces the original ILP problem to a series of smaller sub-problems solved separately in a recursive fashion.
At first, the LP relaxation \( \min_{x \in P} c^\top x \) is solved, thus obtaining a fractional solution \( x_{LP}^* \in \mathbb{R}^n \) and a lower bound \( c^\top x_{LP}^* \) on the optimal value \( z_{ILP} \). The search upper bound \( u \) is set to \( +\infty \).

If \( x^* \in \mathbb{Z}^n \) then the search stops because the optimal solution has been found. Otherwise, the polyhedron \( P \) is partitioned into \( k \) disjoint polyhedral subsets \( P_1, \ldots, P_k \) s.t. \( X = \bigcup_{i=1}^k P_i \cap \mathbb{Z}^n \) and \( x^* \not\in \bigcup_{i=1}^k P_i \). The most common partitioning scheme uses \( k = 2 \) and implicitly defines the structure of a decision tree with \( 2^{n+1} - 1 \) nodes that in the worst case must be completely explored. The binary partitioning is also known as branching on a variable, since it selects a variable \( x_j^* \not\in \mathbb{Z} \) to create two LP sub-problems:

\[
\begin{align*}
\min \{ c^\top x | Ax = b, x_j \leq \lfloor x_j^* \rfloor, x \geq 0, x \in \mathbb{Z}^n \} \\
\min \{ c^\top x | Ax = b, x_j \geq \lceil x_j^* \rceil, x \geq 0, x \in \mathbb{Z}^n \}
\end{align*}
\]

The algorithm proceeds by selecting the next sub-problem, according to one of several available approaches (e.g. best-first, depth-first), and solving the associated LP relaxation to seek a new solution \( x_{LP}^* \in \mathbb{R}^n \). There are 4 possible outcomes of this operation:

1. if there is no such solution then \( P = \emptyset \) and the sub-problem is discarded (fathomed);

2. if \( c^\top x_{LP}^* \geq u \) then \( P \cap \mathbb{Z}^n \) can not contain any solution better than \( u \) and the sub-problem is fathomed;

3. if \( x_{LP}^* \in \mathbb{Z}^n \land c^\top x_{LP}^* < u \) then \( x_{LP}^* \) is the best solution found so far, the sub-problem is fathomed and \( u \) is set to \( c^\top x_{LP}^* \);

4. otherwise, a number of new candidate sub-problems is created through a new branching operation;

Since the number of sub-problems that can be created is bounded, this method always converges. Then there are no more nodes to be explored, the procedure returns the optimal value \( u \) and its associated best solution \( x_{LP}^* \in \mathbb{Z}^n \).

Note that in practice it might be convenient to combine the branch and bound procedure with some of aforementioned procedures to determine cutting planes. This ap-
proach can help improving the estimate of the *fathoming* phase, thus inducing a higher pruning level on the search space, though it comes at higher computational price.
CHAPTER 2. INTEGER LINEAR PROGRAMMING
Chapter 3

Lazy Satisfiability Modulo Theories

This introduction to SMT and to the internals of MATHSAT5 fully draws from publications on the topic like [Griggio, 2012; Griggio, 2009], rearranged and synthesized to fit the goal of providing a proper preliminary understanding of the tool onto which this thesis work is based.

This section will at first review some of the preliminary terminology and notation necessary to understand the topic, then the focus will shift on the standard architecture of a \( T \)-solver solver and finally give an abstract overview of the linear arithmetic solver currently implemented within MATHSAT5 onto which most of the work has been done.

1. Preliminaries

The setting is standard first order logic. Here \textit{constants} are 0-ary function symbols, whereas \textit{terms} are first-order terms built out of both function symbols and variables. If \( t_1, \ldots, t_n \) are terms and \( p \) is a predicate symbol, then \( p(t_1, \ldots, t_n) \) is an \textit{atom}. In this thesis, I will only deal with atoms that are either 0-arity predicates (i.e. Boolean constants), or \textit{linear equations and inequalities} \( \sum_i a_i x_i + c \bowtie 0 \), where \( \bowtie \in \{=, \leq\} \), \( c \) and the \( a_i \)'s are rational numbers and the \( x_i \)'s are uninterpreted integer constants. A \textit{formula} \( \phi \) is built in the usual way out of the universal and existential quantifiers, Boolean connectives, and atoms. A \textit{literal} is either an atom or its negation. A formula is called \textit{quantifier-free} if it does not contain quantifiers, and \textit{ground} if it does not contain free variables. A \textit{clause} is a disjunction of literals. A formula is said to be in \textit{conjunctive}
normal form (CNF) if it is a conjunction of clauses. For every non-CNF formula \( \varphi \), an equisatisfiable CNF formula \( \psi \) can be generated in polynomial time [Tseitin, 1968].

The usual first-order notions of interpretation, satisfiability, validity, logical consequence, and theory, are also assumed as given, e.g., in [Enderton, 2001]. The notation \( \Gamma \models \phi \) denotes that the formula \( \phi \) is a logical consequence of the (possibly infinite) set \( \Gamma \) of formulas. A first-order theory, \( \mathcal{T} \), is a set of first-order sentences. A structure \( \mathcal{A} \) is a model of a theory \( \mathcal{T} \) if \( \mathcal{A} \) satisfies every sentence in \( \mathcal{T} \). A formula is satisfiable in \( \mathcal{T} \) (or \( \mathcal{T} \)-satisfiable) if it is satisfiable in a model of \( \mathcal{T} \).

Given a first-order formula \( \phi \), the propositional abstraction of \( \phi \) is a propositional formula \( \psi \) obtained from \( \phi \) by replacing each theory atom in \( \phi \) with a fresh Boolean constant. The availability of a mapping \( \mathcal{T}2B \) (“theory to Boolean”) from theory atoms to fresh Boolean constants and its inverse \( B2\mathcal{T} \) (“Boolean to theory”) which can be used to obtain the propositional abstraction \( \psi \) from a formula \( \phi \) and vice versa is assumed.

Given a first-order theory \( \mathcal{T} \) for which the (ground) satisfiability problem is decidable, a theory solver for \( \mathcal{T} \) (\( \mathcal{T} \)-solver) is any tool able to decide the satisfiability in \( \mathcal{T} \) of sets/conjunctions of ground atomic formulas and their negations — theory literals or \( \mathcal{T} \)-literals — in the language of \( \mathcal{T} \). If the input set of \( \mathcal{T} \)-literals \( \mu \) is \( \mathcal{T} \)-unsatisfiable, then a typical \( \mathcal{T} \)-solver not only returns unsat, but it also returns the subset \( \eta \) of \( \mathcal{T} \)-literals in \( \mu \) which was found \( \mathcal{T} \)-unsatisfiable. (\( \eta \) is hereafter called a theory conflict set, and \( \neg \eta \) a theory conflict clause.) If \( \mu \) is \( \mathcal{T} \)-satisfiable, then \( \mathcal{T} \)-solver not only returns sat, but it may also be able to discover one (or more) deductions in the form \( \{l_1, \ldots, l_n\} \models_{\mathcal{T}} l \), s.t. \( \{l_1, \ldots, l_n\} \subseteq \mu \) and \( l \) is an unassigned \( \mathcal{T} \)-literal. If so, \( (\bigvee_{i=1}^n \neg l_i \lor l) \) is called a theory-deduction clause. Notice that both theory-conflict clauses and theory-deduction clauses are valid in \( \mathcal{T} \), and are thus called theory lemmas or \( \mathcal{T} \)-lemmas.

Satisfiability Modulo (the) Theory \( \mathcal{T} \) — \( SMT(\mathcal{T}) \) — is the problem of deciding the satisfiability of Boolean combinations of propositional atoms and theory atoms [Barrett et al., 2009b]. Any tool able to decide \( SMT(\mathcal{T}) \) is called \( SMT(\mathcal{T}) \) tool.

Hereafter the following terminology and notation is adopted. The symbols \( \varphi, \psi \) denote \( \mathcal{T} \)-formulas, and \( \mu, \eta \) denote sets of \( \mathcal{T} \)-literals; \( \varphi^p, \psi^p \) denote propositional formulas, \( \mu^p, \eta^p \) denote sets of propositional literals (i.e., truth assignments) and are often used as synonyms for the propositional abstraction of \( \varphi, \psi, \mu, \) and \( \eta \) respectively, and vice versa (e.g., \( \varphi^p \) denotes \( \mathcal{T}2B(\varphi) \), \( \mu \) denotes \( B2\mathcal{T}(\mu^p) \)). If \( \mathcal{T}2B(\varphi) \models \bot \), then \( \varphi \) is said to be propositionally unsatisfiable.
2. THE ONLINE LAZY SMT SCHEMA

Nowadays, SMT(\(\mathcal{T}\)) solvers are often designed around the so-called “lazy” approach [Sebastiani, 2007; Barrett et al., 2009a], also known as “DPLL(\(\mathcal{T}\))” [Nieuwenhuis, Oliveras, and Tinelli, 2006]. This design combines a number of \(\mathcal{T}\)-solvers with a propositional SAT solver based on the DPLL algorithm [Davis, Logemann, and Loveland, 1962].

In a DPLL(\(\mathcal{T}\)) solver the DPLL engine enumerates truth assignments \(\mu_i^p\) that propositionally satisfy the propositional abstraction \(\varphi^p\) of the input formula \(\varphi\), whereas the \(\mathcal{T}\)-solvers are used to check the \(\mathcal{T}\)-satisfiability of each \(\mu_i \overset{\text{def}}{=} B2\mathcal{T}(\mu_i^p)\): if the current \(\mu_i\) is \(\mathcal{T}\)-satisfiable, then \(\varphi\) is \(\mathcal{T}\)-satisfiable; otherwise, if none of the \(\mu_i\)’s are \(\mathcal{T}\)-satisfiable, then \(\varphi\) is \(\mathcal{T}\)-unsatisfiable.

Figure 3.1. An online schema of \(\mathcal{T}\)-DPLL based on modern DPLL.
Figure 3.1 shows an abstraction of the algorithm implemented in most state-of-the-art lazy SMT solvers based on a DPLL engine (see e.g. [Zhang and Malik, 2002]).

The $\mathcal{T}$-DPLL solver takes as inputs a $\mathcal{T}$-formula $\varphi$ and an (initially empty) set of $\mathcal{T}$-literals $\mu$, these are encoded in their corresponding propositional formulas $\varphi^p$ and $\mu^p$ using the $T2B/B2T$ bijective mappings.

$\mathcal{T}$-preprocess simplifies $\varphi$ into a simpler formula while preserving its $\mathcal{T}$-satisfiability. If this process produces some conflict, then $\mathcal{T}$-DPLL returns unsat. This step may require the conversion of $\varphi$ to CNF, and it combines most or all the Boolean preprocessing steps available from SAT literature with theory-dependent rewriting steps on the $\mathcal{T}$-literals of $\varphi$.

$\mathcal{T}$-decide-next-branch selects some literal $l^p$ and adds it to $\mu^p$ in an analogous way to the standard literal selection heuristic decide-next-branch in DPLL [Zhang and Malik, 2002], though it might take into consideration also the semantics in $\mathcal{T}$ of the literals to be selected.

$\mathcal{T}$-deduce, in its simplest version, behaves similarly to deduce in DPLL [Zhang and Malik, 2002], i.e. it iteratively performs Boolean Constraint Propagation (BCP). This step is repeated until one of the following events occur:

1. $\mu^p$ propositionally violates $\varphi^p$ ($\mu^p \land \varphi^p \models \bot$). If so, $\mathcal{T}$-deduce behaves like deduce in DPLL, returning conflict.

2. $\mu^p$ satisfies $\varphi^p$ ($\mu^p \models \varphi^p$). If so, $\mathcal{T}$-deduce invokes $\mathcal{T}$-solver on $B2T(\mu^p)$: if $\mathcal{T}$-solver returns sat, then $\mathcal{T}$-deduce returns sat; otherwise, $\mathcal{T}$-deduce returns conflict.

3. no more literals can be deduced. If so, $\mathcal{T}$-deduce returns unknown.

The $\mathcal{T}$-deduce procedure is often enhanced through the adoption of a number of important techniques. These are:

- Early Pruning (EP) adds an invocation to $\mathcal{T}$-solver on $B2T(\mu^p)$ if $\mu^p$ does not yet satisfy $\varphi^p$: if $\mathcal{T}$-solver returns unsat, then $\mathcal{T}$-deduce returns conflict.

- Weak Early Pruning [Sebastiani, 2007], consists of letting $\mathcal{T}$-solvers use an approximate but cheaper consistency check algorithm during EP calls in
order to limit the overall overhead. $T$-solvers can be imprecise in detecting conflicts during EP calls, that is sat can be returned even when the current truth assignment $\mu$ is $T$-inconsistent, as long as precision is recovered in non-EP calls (i.e. when $\mu^p \models \varphi^p$). Weak EP is particularly effective on “hard” theories, including $\mathcal{L}A(\mathbb{Z})$.

- **$T$-propagation**, enabled by EP calls, appends $l^p$ to $\mu^p$ to be propagated whenever $T$-solver is able to perform deductions in the form $\eta \models_T l$ s.t. $\eta \subseteq \mu$ and $l^p \triangleq T2B(l)$ is an unassigned literal in $\varphi^p$.

- **Layering** [Bozzano et al., 2005; Bruttomesso et al., 2007], which consists of using a collection of $T$-solvers $S_1, \ldots, S_N$ organized in a *layered hierarchy* of increasing expressibility and complexity. Each solver $S_i$ is able to decide a theory $T_i$ which is a sub-theory of $T_{i+1}$, and which is less expensive to handle than $T_{i+1}$. The solver $S_N$ is the only one that can decide the full theory $T$. If the solver $S_i$ detects an inconsistency, then there is no need of invoking the more expensive solvers $S_{i+1}, \ldots, S_N$, and unsat can be returned immediately.

- **Splitting on-demand** [Barrett et al., 2006], it allows $T$-solvers to not always decide the $T$-consistency of the current set of constraints $\mu$ but rather return unknown together with a list of new $T$-lemmas containing new $T$-atoms, which will be then taken into account in the DPLL search by branching on the new atoms and performing BCP and conflict detection on the new lemmas. This techniques exploits DPLL to perform disjunctive reasoning instead of handling it directly inside the $T$-solvers whenever this is necessary, thus exploiting all the advanced techniques (e.g. conflict-driven backjumping and learning) for search-space pruning implemented in modern DPLL engines and allowing for simpler $T$-solvers implementations.

$T$-analyze-conflict is an extension of analyze-conflict of DPLL [Zhang and Malik, 2002], that produces a conflict set $\eta^p$ and the corresponding decision level (blevel) where to backtrack. Its conflict analysis is performed either on the clause falsified during BCP in $T$-deduce (case (i) above), or on the propositional abstraction $\eta^p \triangleq T2B(\eta')$ of the $T$-conflict set $\eta'$ produced by $T$-solver (case (ii) above).
$\mathcal{T}$-backtrack, like backtrack in DPLL [Zhang and Malik, 2002], performs $\mathcal{T}$-learning by constructing the clause $\neg \eta^p$ from the conflict set $\eta^p$ and adding it to $\varphi^p$ either temporarily or permanently, and does $\mathcal{T}$-backjumping by backtracking up to blevel.

3. General Architecture of the $\mathcal{L}A(\mathbb{Z})$-solver

As it is shown in figure 3.2, the general architecture of the $\mathcal{T}$-solver for $\mathcal{L}A(\mathbb{Z})$ (also known as $\mathcal{L}A(\mathbb{Z})$-solver) is organized as a layered hierarchy of sub-modules, with cheaper and less powerful ones invoked earlier and more often.

![Figure 3.2. Architecture of the $\mathcal{L}A(\mathbb{Z})$-solver.](image)

The $\mathcal{L}A(\mathbb{Z})$-solver takes as inputs a set of $\mathcal{L}A(\mathbb{Z})$-constraints, and checks their consistency using the following strategy.

1. At first an internal Simplex-based $\mathcal{L}A(\mathbb{Q})$-solver, similar to that described in [Dutertre and Moura, 2006], is invoked on the rational relaxation of the input problem. If a conflict is detected the $\mathcal{L}A(\mathbb{Z})$-solver immediately returns unsat, otherwise the model is examined to check whether all integer variables are already assigned to an integral value. If that is the case, then the $\mathcal{L}A(\mathbb{Q})$-model is also a $\mathcal{L}A(\mathbb{Z})$-model and the solver immediately returns sat.
2. Otherwise the Diophantine equations handler, similar to the first part of the Omega test described in [Pugh, 1991], is invoked with the task of trying to eliminate input equations by computing a parametric solution of the system and then substituting each variable in the inequalities with its parametric expression. If the system of equations is infeasible in itself, the Diophantine module detects the inconsistency and returns unsat.

3. Otherwise, the inequalities obtained by substituting the variables with their parametric expressions are normalized, tightened and then sent to the \( \mathcal{L}A(\mathbb{Q}) \)-solver, onto which the \( \mathcal{L}A(\mathbb{Q}) \)-consistency of the new set of constraints is checked.

4. If the rational relaxation of the input problems is consistent its model is not a \( \mathcal{L}A(\mathbb{Z}) \)-solution, then the branch and bound module [Schrijver, 1986] is invoked, the latter being actually composed by an “internal” and an “external” sub-modules operating in sequence. The “internal” branch and module, activated for first, performs case splits directly within the \( \mathcal{L}A(\mathbb{Z}) \)-solver for a bounded (and small) number of branches. The module starts by selecting an integer variable assigned a rational non-integral value using a history-based greedy strategy inspired by the “pseudo-cost branching” rule described in [Achterberg, 2007]. This procedure selects the variable that resulted in the minimum number of violations of integral constraints in the previous branches. More specifically, let \( n_k^l \) and \( n_k^r \) be respectively the number of left and right branches on the variable \( x_k \) in the branch and bound search, \(^1\) and let \((g_k^l)_i\) and \((g_k^r)_i\) be respectively the number of integer variables with non-integer values after having performed the \( i \)-th left (resp. right) branch on \( x_k \). Then, the score of \( x_k \) is defined as the minimum between

\[
\sum_{i=1}^{n_k^l} (g_k^l)_i / n_k^l
\]

and

\[
\sum_{i=1}^{n_k^r} (g_k^r)_i / n_k^r
\]

, and heuristic always selects the variable \( x_k \) with the smallest score. Then, the “internal” branch and bound sub-module (recursively) divides the problem in two

\(^1\) Here, left branch is the branch on \( (x_k - \lfloor q_k \rfloor \leq 0) \), and right branch is that on \( (-x_k + \lceil q_k \rceil \leq 0) \).
CHAPTER 3. LAZY SATISFIABILITY MODULO THEORIES

sub-problems obtained by adding respectively the constraints

\[(x_k - [q_k] \leq 0)\]

and

\[(-x_k + [q_k] \leq 0)\]

to the original formula, until either a \(L\mathcal{A}(\mathbb{Z})\)-model is found by the \(L\mathcal{A}(\mathbb{Q})\)-solver, all the sub-problems are proved unsatisfiable or the maximum limit of case splits is reached\(^2\). If the “internal” branch and bound finds a conflict, an explanation can be recursively generated by combining, for each node of the branch-and-bound tree, the \(L\mathcal{A}(\mathbb{Q})\)-explanations for the conflicts on the two sub-branches.\(^3\)

5. If no \(L\mathcal{A}(\mathbb{Z})\)-solution is still found then the “external” branch and bound module is invoked. This module, also used for handling dis-equalities, adopts the splitting on-demand approach of [Barrett et al., 2006] and works by generating the \(L\mathcal{A}(\mathbb{Z})\)-lemma

\[(x_k - [q_k] \leq 0) \lor (-x_k + [q_k] \leq 0)\]

and sending it back to the DPLL engine. The idea is that of exploiting the DPLL engine for the exploration of the branches introduced by branch and bound, rather than handling the case splits within the \(L\mathcal{A}(\mathbb{Z})\)-solver.

Whenever an inconsistency is detected and unsat is returned, an explanation for the conflict expressed in terms of a subset of input equations is generated using both the Diophantine equations handler module and the \(L\mathcal{A}(\mathbb{Q})\)-solver. The reason for this is that the explanation is not meant to include substitute equations generated by the Diophantine module and unknown to the DPLL engine.

---

\(^2\) The actual bound on case splits is proportional to the number of variables in the input problem.

\(^3\) More specifically, if \(\neg \eta_l \land (x_k - [q_k] \leq 0)\) is the \(L\mathcal{A}(\mathbb{Q})\)-explanation of the left branch and \(\neg \eta_r \land (-x_k + [q_k] \leq 0)\) is the \(L\mathcal{A}(\mathbb{Q})\)-explanation of the right branch, then \(\neg \eta_l \land \neg \eta_r\) is the \(L\mathcal{A}(\mathbb{Z})\)-explanation of the current node.
Chapter 4

Lazy Optimization Modulo Theory

ŁA(ℤ) optimization techniques exhibited in this document are built on top of (or as simple extension of) the ŁA(ℚ) optimization support first introduced in [Sebastiani and Tomasi, 2012]. Therefore, in order to provide a meaningful background to frame what has been done to extend MATHSAT5 with ŁA(ℤ) optimization, the noteworthy elements of such research achievements will be here summarized and briefly reviewed.

1. Theoretical Setting

An Optimization Modulo ŁA(ℚ) ∪ ℰ problem (OMT(ŁA(ℚ) ∪ ℰ)) is a pair ⟨ϕ, cost⟩ s.t. ϕ is a SMT(ŁA(ℚ) ∪ ℰ) formula and cost is a ŁA(ℚ) variable occurring in ϕ. OMT(ŁA(ℚ) ∪ ℰ) addresses the problem of finding a model M for ϕ (if any) whose value of cost is minimum.

Let ϕ be in the form ϕ’ ∧ (cost < c) [resp. ϕ’ ∧ ¬(cost < c)] for some c ∈ ℚ, then c is an upper bound [resp. lower bound] for cost. If ub [resp. lb] is the minimum upper bound [resp. the maximum lower bound] for ϕ, then the interval [lb, ub] is the range of cost.

As in [Nelson and Oppen, 1979], ℰ is some (combination of) stably infinite theory (-ies) with equality that is signature-disjoint with ŁA(ℚ). Note that ℰ might be empty, in such a case OMT(ŁA(ℚ) ∪ ℰ) reduces to OMT(ŁA(ℚ)).

OMT(ŁA(ℚ)) allows for a straightforward encoding of various problem domains: LP, LDP and LGDP [Sebastiani and Tomasi, 2012].
2. Offline schema for $OMT(LA(Q))$

The simplest version of the $OMT(LA(Q) \cup T)$ schema presented in [Sebastiani and Tomasi, 2012], is shown in figure 4.1. This procedure performs a sequence of incremental calls to a SMT solver on formulas like $\varphi \land (cost \geq l_i) \lor (cost < u_i)$, each time restricting the range $[l_i, u_i]$ according to a linear-search or binary-search schema. To prevent the SMT-solver from repeatedly generating the same $LA(Q)$-satisfiable truth assignment, [Sebastiani and Tomasi, 2012] extended $LA(Q)$-solver so that the minimum value of $cost$ can be found with a $Minimize$ sub-routine.

Algorithm 4.1 takes as input an $OMT(LA(Q))$ problem, an optional range $[lb, ub]$ for the $cost$ variable (o.w. assumed $[-\text{inf}, +\text{inf}]$) and returns the couple $\langle M, cost \rangle$ – the model of minimum cost and its corresponding minimum cost value.

```plaintext
offline_omt(formula \varphi, variable cost, number lb, number ub)
1. \begin{align*}
    l & \leftarrow lb; u \leftarrow ub; PIV \leftarrow T; M \leftarrow \emptyset \\
    \varphi & \leftarrow \varphi \cup \{(cost < l), (cost < u)\}
\end{align*}
2. \begin{algorithmic}
    \While{\(l < u\)}
        \begin{algorithmic}
        \If{BinSearchMode()}
            \State pivot \leftarrow ComputePivot(l, u)
            \State PIV \leftarrow \{(cost < pivot)\}
            \State \varphi \leftarrow \varphi \cup \{PIV\}
            \State \langle res, \mu \rangle \leftarrow SMT.IncrementalSolve(\varphi)
            \State \eta \leftarrow SMT.ExtractUnsatCore(\varphi)
        \Else
            \State \langle res, \mu \rangle \leftarrow SMT.IncrementalSolve(\varphi)
            \State \eta \leftarrow \emptyset
        \EndIf
        \If{res = SAT}
            \State \langle M, u \rangle \leftarrow Minimize(cost, \mu)
            \State \varphi \leftarrow \varphi \cup \{(cost < u)\}
        \Else
            \If{PIV \notin \eta}
                \State l \leftarrow u
            \Else
                \State l \leftarrow pivot
                \State \varphi \leftarrow \varphi \setminus \{PIV\}
                \State \varphi \leftarrow \varphi \cup \{\neg PIV\}
            \EndIf
        \EndIf
        \EndWhile
    \EndAlgorithmic
\end{algorithmic}
3. \begin{align*}
    \text{return} \ & \langle M, u \rangle
\end{align*}
```

Figure 4.1. Offline $OMT(LA(Q))$ procedure based on Mixed Linear/Binary Search.
Variables \( l, u \) defining the current range are initialized to \( lb \) and \( ub \) respectively, the atom \( PIV \) to \( \top \) while \( M \) is initialized to be an empty model. The input formula \( \varphi \) is augmented with constraints bounding the search of an optimum value for variable \( cost \) within the range \([l, u]\). The loop at rows 3-26 iteratively explores the solution space, progressively reducing the current range \([l, u]\) until it is empty.

In **binary-search mode** the (possibly new) atom \( PIV = (cost < pivot) \) – with \( pivot \in [l, u] \) computed by an heuristic function \( \text{ComputePivot} \) (e.g. \( \frac{l+u}{2} \)) – is pushed on the formula stack of \( \varphi \) to temporarily restrict the search space to \([l, pivot]\) (rows 5-7). Then, in both search modes, the \( \text{SMT.IncrementalSolve} \) procedure is invoked to obtain a couple \( \langle res, \mu \rangle \) being respectively the satisfiability status of the input formula \( \varphi \) and the corresponding \( \mathcal{LA}(Q) \)-satisfiable truth assignment respectively – if any (rows 8,11). The incremental engine of the SMT-solver is crucial, since it allows to resume the search were it stopped in the previous step and to reuse of all previously learned clauses. The conflict set \( \eta \) is extracted from \( \varphi \) with the aid of procedure \( \text{SMT.ExtractUnsatCore} \) in **binary-search mode**, whereas it is ignored and thus set to the \( \emptyset \) in **linear-search mode** (rows 9,12).

If \( \varphi \) is \( \mathcal{LA}(Q) \)-satisfiable, then procedure \( \text{Minimize} \) is invoked in order to retrieve the model \( \mathcal{M} \) for \( \mu \) of minimum cost \( u - \inf \) iff unbounded – (row 15). The procedure \( \text{Minimize()} \) is a simple extension of the simplex-based \( \mathcal{LA}(Q) \)-solver of [Dutertre and Moura, 2006], using standard Simplex techniques. The minimum cost \( u \) is a stricter upper bound for the current search space, therefore a new constraint \( (cost < u) \) is pushed on the \( \varphi \) formula stack (row 16).

Otherwise if \( \varphi \) is \( \mathcal{LA}(Q) \)-unsatisfiable and **binary-search mode** is active, then two sub-cases are distinguished by examining the conflict set \( \eta \) (row 18). If \( PIV \notin \eta \) then \( \varphi \setminus \{ PIV \} \) is \( \mathcal{LA}(Q) \)-inconsistent and there is no model in the whole cost range \([l, u]\), thus the search is immediately terminated by forcing the current range to be empty (row 19). Otherwise, since **binary-search mode** is active, it is only possible to conclude that there is no model in the cost range \([l, pivot]\), so it is still necessary to explore the cost range \([pivot, u]\). Thus in rows 21-23 \( l \) is set to \( pivot \), \( PIV \) is popped from \( \varphi \) and its negation is pushed into \( \varphi \). In the case of **linear-search mode** \( \eta \) is always set to \( \emptyset \), thus search always immediately terminates.

Then the couple \( \langle \mathcal{M}, u \rangle \) is returned – with value \( \langle \emptyset, ub \rangle \) if there is no solution in the range \([lb, ub]\).
An important remark is that BinSearchMode() must return false infinitely often, thus forcing a “linear-search” step, to prevent algorithm 4.1 from being stuck in an infinite loop. This may happen because SMT.ExtractUnsatCore() can return a set $\eta$ containing PIV even if $\varphi \setminus \{PIV\}$ is $LA(Q)$-inconsistent.

3. Inline schema for $OMT(LA(Q))$

The inline version of the $LA(Q)$-optimizer moves the entire optimization procedure within the SMT-solver, by extending the conflict-driven clause-learning (CDCL) Boolean search loop of standard lazy SMT with the capability of performing range-minimization in linear and binary-search mode [Silva and Sakallah, 1996]. The steps that need to be taken are:

1. **Initialization**: variables $lb$, $ub$, $l$, $u$, $PIV$, $pivot$, $M$ are pushed inside the SMT-solver and initialized as in algorithm 4.1 (rows 1-2);

2. **Range Update & Pivoting**: each time the CDCL search backtracks to decision level 0 the range $[l,u]$ is updated s.t. $u$ [resp. $l$] is assigned the lowest [resp. highest] value $u_i$ [resp. $l_i$] such that the atom $(cost < u_i)$ [resp. $\neg(cost < u_i)$] is currently assigned at level 0. Search terminates if $u \leq l$ or literals $l$, $\neg l$ are assigned at level 0. In binary-search mode, $pivot \in [l,u]$ is computed with ComputePivot() and the (possibly new) atom $PIV = (cost < \text{pivot})$ is decided to be true (level 1) by the SAT-solver, as in steps 4-7 of algorithm 4.1;

3. **Decreasing the Upper Bound**: whenever a satisfiable assignment $\mu$ for $\varphi$ is generated, the former is also fed to the Minimize() procedure to retrieve the minimum $LA(Q)$-satisfiable value $min$ for variable $cost$. Then the unit clause $(cost < min)$ is learned, fed to the back-jumping mechanism thus forcing the SAT-solver to back-jump to level 0, and unit-propagated. This case, shown in figure 4.2-(iii), mimics steps 14-16 of algorithm 4.1, permanently restricting the cost range to $[l, min]$;

4. **Increasing the Lower Bound**: in binary-search mode, when conflict analysis of SAT-solver produces a conflict clause in the form $\neg PIV \lor \neg \eta'$ s.t. all literals in $\eta'$ are assigned true at level 0 (e.g. $\varphi \setminus \{PIV\}$ is $LA(Q)$-inconsistent) the SAT-solver backtracks to level 0 and unit-propagates $\neg PIV$. This case, shown also in
Figure 4.2. One piece of possible execution of an inline procedure. (i) Pivoting on \( (\text{cost} < \text{pivot}_0) \). (ii) Increasing the lower bound to \( \text{pivot}_0 \). (iii) Decreasing the upper bound to \( \min_{\text{cost}}(\mu_i) \).
figure 4.2-(ii), mirrors steps 21-23 of algorithm 4.1 and permanently restricts the cost range to $[pivot, u]$.

5. **Unit Clause Reuse**: in binary-search mode, whenever a novel minimum $min$ value is found, the unit clause $(cost < pivot)$ is learned too. The latter allows the SAT-solver to reuse all the clauses in the form $\neg PIV \lor C$ which have been learned when searching the cost range $[l, pivot]$;
Part II

Contributions
This part will examine the various approaches that we have tried in order to bring into OPTIMATHSAT the capability of performing optimization over linear integer equations.

In this part it will be assumed that the reader is comfortable with the notation and terminology introduced in the previous sections, and in particular with the formal definition of an ILP problem. It will be further assumed that the set of feasible solutions $\mathcal{X}$ is always bounded for all input problems, e.g. none of them is unbounded. This assumption allows for some simplifications in the following explanation and is compatible with the setting of the $\mathcal{L}\mathcal{A}(\mathbb{Z})$ optimization problems being benchmarked.

Note that most of the design choices we have made hereafter are not discussed in depth because they have a very solid and documented research background that has been already introduced in the previous part.

Chapter 5 describes our new DPLL scheme implementation with support for linear integer optimization through a number of search strategies: linear, binary and dynamic.

Chapter 6 examines the direct implementation of integer optimization within the $\mathcal{L}\mathcal{A}(\mathbb{Z})$-Solver. Though all of the adopted approaches make use of branch and bound theoretical framework, we will see that in OPTIMATHSAT has been possible to introduce specific optimizations specifically related to the way in which handles equations and solves the general problem of $\mathcal{L}\mathcal{A}(\mathbb{Z})$ satisfiability.

Chapter 7 compares the performances of our new OPTIMATHSAT implementation to that of other ILP/CP solvers under a common benchmarking framework.

Finally, chapter 8 will sum up the overall scientific contribution of this master thesis to the research field, the difficulties that have been engaged and the way to go for further developments.
Chapter 5

DPLL In-line Schema for

\( OMT(\mathcal{LA}(\mathbb{Z})) \)

Encouraged by the good performances obtained by the in-line implementation of the \( \mathcal{LA}(\mathbb{Q}) \) optimizer we have chosen to skip the implementation of an offline version, which is both trivial and inefficient, and directly extend OPTIMATHSAT DPLL procedure so that \( \mathcal{LA}(\mathbb{Z}) \) optimization is supported too.

The modified DPLL scheme, shown as algorithm 5.1, takes as input the usual formula \( \varphi \), an empty assignment \( \mu \) and a new value \( l_{\text{initial}} \). It can work both in linear and in binary search mode, depending on the configuration of the solver and the availability of a lower bound for the problem.

Initially, the input formula \( \varphi \) is preprocessed and transformed into an equisatisfiable one (row 1), as described in [Tseitin, 1968], and further simplified when possible (see [Cimatti et al., 2013]). If a conflict is detected then the input problem has no solution, \( X = P = \emptyset \) and unsat is returned (rows 2-3). Otherwise the search range \([l, u]\), stating the domain of possible values for the integer variable \( \text{cost\_var} \), is initialized to the interval \([l_{\text{initial}}, +\infty]\). The variable \( \text{pivot} \) is initialized to \( +\infty \), while the pivoting weight \( p \) is set to \( \frac{1}{2} \) so that newly computed pivots divide the current range interval in two exact halves (round approximations). After the initialization phase, the solution space is explored by combining the usual CDCL/DPLL loop (rows 5-46) with a preconfigured search exploration strategy that can be either linear or binary. The binary-search scheme depends on the availability of a lower bound for the optimal value.
SatValue DPLL (number \( l_{\text{initial}} \), formula \( \varphi \), assignment \& \( \mu \))

1. if preprocess \((\varphi, \mu) == \text{conflict}\) then
   return \(\langle + \infty, \text{unsat} \rangle\)
2. end if
3. \(l \gets l_{\text{initial}}; u \gets +\infty; \text{pivot} \gets +\infty; p = \frac{1}{2};\)
4. loop
5. if \((l == u)\) then
   return \(\langle u, \text{sat} \rangle\)
7. else if \((u \neq +\infty \text{ and } l \neq -\infty \text{ and binaryMode()}\) then
   \( \text{pivot} \gets [p \cdot u + (1-p) \cdot l] \)
   \( \text{pivot} \gets \langle \text{cost}_\text{var} < \text{pivot}\rangle\)
   \( \varphi = \varphi \cup \{ \text{pivot} \} \)
8. end if
9. decide-next-branch \((\varphi, \mu)\)
10. loop
11. status = deduce \((\varphi, \mu, \eta)\)
12. if \((\text{status} == \text{sat})\) then
13. \(u \gets \text{minimize}(\varphi, \mu, \text{cost}_\text{var})\)
14. \(\text{bound} \gets \langle \text{cost}_\text{var} < u \rangle\)
15. \(\varphi = \varphi \cup \{ \text{bound} \} \)
16. set-decision-level-to(0)
17. break
18. else if \((\text{status} == \text{conflict})\) then
19. \(\text{blevel} = \text{analyze-conflict}(\varphi, \mu, \eta)\)
20. if \((\text{blevel} \leq 0)\) then
21. if \((u \neq +\infty)\) then
22. return \(\langle u, \text{sat} \rangle\)
23. else
24. return \text{unsat}
25. end if
26. else
27. if \((\text{binaryMode}())\) then
28. if \((\text{piv} \notin \eta)\) then
29. \(l \gets u\)
30. else
31. \(l \gets \text{pivot}\)
32. \(\varphi = \varphi \setminus \{ \text{piv} \} \cup \{ \neg \text{piv} \} \)
33. end if
34. end if
35. \(\text{backtrack}(\text{blevel}, \varphi, \mu, \eta)\)
36. break
37. else
38. break
39. end if
40. \(\text{backtrack}(\text{blevel}, \varphi, \mu, \eta)\)
41. break
42. end if
43. end loop
44. end loop

Figure 5.1. An in-line schema of DPLL with linear and binary integer optimization.
In linear-search mode, when the range of the search is empty then the latest best cost value \( u \) is returned together with SAT response at lines 6–7, because the condition \( l == u \) is true (here, assuming \( l_{\text{initial}} \neq +\infty \) is fair). Rows 8–12 are skipped because \( l == -\infty \), therefore execution flow resumes at row 13 where the next decision literal from \( \varphi \) is picked up and added to \( \mu \) according to some heuristic. After that, a call to deduce() performs all possible unit propagations and adds them to \( \mu \) (row 15). If the procedure returns sat, then it means that the search found an assignment \( \mu \) for \( \varphi \) such that \( \text{cost}_{-\text{var}} \) is integral. Then, by calling minimize the \( \mathcal{LA}(\mathbb{Z}) \)-Solver can try to find out the optimum value of \( \text{cost}_{-\text{var}} \) for the same Boolean assignment \( \mu \) (row 17). When the solution space is finite this call is entirely optional for \( OMT(\mathcal{LA}(\mathbb{Z})) \) and can be substituted by the function get-value, though it can speed up the search by generating an immediate conflict within the current assignment \( \mu \). The new constraint \( \text{cost} < u \) is then pushed on the formula stack of \( \varphi \) in order to effectively restrict the search range to the new interval \([l, u]\) (rows 18–20), and the current decision level is set to be zero so that the CDCL procedure will not backtrack any further back than this point any-more. Otherwise, if deduce() detected an inconsistency the conflict \( \eta \) is analysed in order to get the decision level \( \text{blevel} \) at which the search should back-jump. If \( \text{blevel} \leq \text{zero} \), then there are two possible cases: either a satisfiable assignment \( \mu \) for \( \varphi \) with a corresponding optimal value \( u \) had already been found in the past, and thus the search can return sat, or the research never found any satisfiable assignment for \( \varphi \), meaning that the polyhedron does not contain any integral solution (e.g. \( \mathcal{X} = \emptyset \)) and unsat can be returned. In linear-search mode rows 31–38 are ignored, hence the search immediately backtracks to the \( \text{blevel} \) corresponding to the current conflict \( \eta \), so that a different Boolean decision can be taken.

Whenever \( l_{\text{initial}} \neq -\infty \) and binaryMode() returns true, as soon as the first upper bound \( u \) is found, the search switches to binary-search mode. Then, each time \( u \) is updated, a new pivot is computed and the constraint \( \text{cost}_{-\text{var}} < \text{pivot} \) is pushed on the formula stack to temporarily restrict the search within the interval \([l, \text{pivot}]\) (rows 8–12). The search proceeds as in the linear-search mode case whenever the procedure deduce() returns sat. If instead unsat is returned and the search needs not to backtrack to \( \text{blevel} \) 0 or less, then conflict \( \eta \) is further examined at lines 31–38. If piv does not belong to conflict \( \eta \) then it means that there is no satisfiable assignment \( \mu \) for the input formula \( \varphi \) such that \( \text{cost}_{-\text{var}} \) has values in \([l, u]\), therefore it can be deduced that \( u \) is
a *lower bound* for \( \text{cost}_\text{var} \) and \( l \) can be updated accordingly. The algorithm will then stop at lines 6 – 7, since now \( l == u \), and the optimal solution value \( u \) returned. Otherwise, if \( \text{piv} \) does belong to the conflict \( \eta \) then it is possible to conclude that there is no solution in \([l, \text{pivot}]\) only, since there still might be one in the interval \([\text{pivot}, u]\). Hence the lower-bound \( l \) is set to the current \( \text{pivot} \) value, and the current pivoting constraint is popped from the stack of the input formula \( \varphi \) and pushed back negated. After that range adjustment the search will go on normally, this time restricted in the interval \([\text{pivot}, u]\).

It is easy to show that algorithm 5.1 always terminate, since the set of feasible solutions \( \mathcal{X} \) is finite (by assumption) and each time a satisfiable assignment \( \mu \) is found the corresponding optimal value \( u \) is pruned from the search space by a newly introduced linear constraint \((\text{cost}_\text{var} < u)\).

1. A Dynamic Strategy

Provided that the solution space \( \mathcal{X} \) is finite, common sense dictates that its exploration by means of a *binary* strategy is more efficient than a *linear* approach because in the former case the set of potential optimum values is reduced by half or more at each iteration of the main loop, whereas the same is not guaranteed to happen in *linear* search mode. When it comes to SMT, however, it is no more inherently safe to jump at this conclusion since the time spent within the \( \text{decide()} \) procedure is not uniform, with \( \text{unsat} \) calls being generally more heavy than \( \text{sat} \) ones. Typically, the *binary-search* approach generates more unsatisfiable assignments due to the additional pivoting constraint being false whenever the optimal solution does belong to the interval \([l, \text{pivot}]\).

A proper solution to this issue is to modify algorithm 5.1 so that the search strategy is dynamically chosen at runtime based on the overall performances of the DPLL/CDCL solver, which cost is roughly captured by the number of conflicts required to take a \( \text{satunsat} \) decision. In order to properly assess the convenience of one strategy in respect to the other, it is also necessary to account for the different information gain they provide in terms of pruning of the solution space. Potentially, this estimator can be misleading because historical information can’t provide absolute certainty that the future behaviour of the solver will be similar under dissimilar circumstances, the latter being a different formula \( \varphi \) and truth assignment \( \mu \). This kind of issue does not have a
1. A DYNAMIC STRATEGY

correspondingly simple solution, yet implementing the dynamic strategy and verify its practical convenience through benchmarking is easy enough to be worth trying.

In linear-search mode we have set the price/gain ratio to be given by

\[ r_{\text{lin}} = \frac{\Delta_{\text{ub}}}{\text{conflicts}_{\text{lin}}} \]

where \( \Delta_{\text{ub}} \) stands for the latest upper bound improvement found in linear mode for the candidate optimal value of the cost variable \( \text{cost}_{\text{var}} \) and \( \text{conflicts}_{\text{lin}} \) stands for the corresponding number of conflicts experienced while computing such result. In binary-search mode, we have tried two different rates

\[ r_{1\text{bin}} = \frac{|\Delta_{\text{ub}}|}{\text{conflicts}_{\text{bin}}} \]
\[ r_{2\text{bin}} = \frac{|\Delta_{\text{ub}}| + |0.5 \cdot \Delta_{\text{lb}}|}{\text{conflicts}_{\text{bin}}} \]

where \( \Delta_{\text{ub}} \) (resp. \( \Delta_{\text{lb}} \)) stands for the latest upper bound (resp. lower bound) improvement in binary mode and \( \text{conflicts}_{\text{bin}} \), alike in the previous case, estimates the conflicts experienced during the corresponding loop iteration of the CDCL/DPLL solver.

After a first search cycle in linear mode followed by a binary one, necessary to use the dynamic strategy, at the beginning of each iteration of the main loop the condition \( r_{\text{lin}} < r_{1,2\text{bin}} \) is evaluated and the search mode set to binary if it holds, to linear otherwise.
Chapter 6

ŁA(Z)-Solver optimization extension

The main advantage of extending the ŁA(Z)-Solver architecture, reviewed in section 3, with the capability of optimizing the value of variable $\text{cost}_\text{var}$ is to put the current truth assignment for $\varphi$ – augmented with a new constraint $(\text{cost}_\text{var} < u)$ – in an inconsistent state, which will cause the CDCL/DPLL loop to immediately back-jump and search for a new truth assignment that improves the current solution. This does not necessarily happen whenever $u$ is not optimal for the current truth assignment $\mu$, therefore this technique has the potential of speeding up the overall DPLL search by reducing the number of iterations it performs.

As already discussed in section 2, the most common optimization technique used for integer linear arithmetic is the LP-based branch and bound algorithm. Therefore, a straightforward implementation of such an approach within OPTIMATHSAT will be examined at first. The focus will then shift onto a more elaborated implementation of the same approach, capable of exploiting the unsatisfiability core extraction of $T$-solvers in order to perform back-jumping within the branch and bound search tree itself. Finally, the opportunity of adopting a simplified version of the same algorithm will be discussed.

1. Basic Branch and Bound

Figure 6.1 shows an example of linear integer optimization through branch and bound using a recursive implementation that visits the search tree in a pre-order fashion. As it will be shown in the next section, it can be easily re-factored so that there isn’t any implicit recursion occurring on the function call stack.
minimize_backtracking (const TheoryLitSet& to_ignore)
1. if internal_bb_init(to_ignore, true)) then
2. \[ z_{ILP} = +\infty; \]
3. opt_lia_bb_node();
4. internal_bb_deinit();
5. end if
6. return \( z_{ILP} \);

opt_lia_bb_node()
1. if ! bb_solver_->check() then
2. return ;
3. end if
4. \( z_{LP} = \text{bb_solver}_->\text{minimize}() \)
5. if \( z_{LP} \geq z_{ILP} \) then
6. return ;
7. end if
8. if internal_bb_model_is_integral() then
9. \[ z_{ILP} = \min(z_{ILP}, z_{LP}); \]
10. return ;
11. else
12. BBFrame frame = get_bb_branch();
13. bb_solver_->push_constraint(frame.left);
14. opt_lia_bb_node();
15. bb_solver_->pop_constraint();
16. bb_solver_->push_constraint(frame.right);
17. opt_lia_bb_node();
18. bb_solver_->pop_constraint();
19. return ;
20. end if

Figure 6.1. LA(Z) minimization with simple backtracking mechanism

The function minimize_backtracking is responsible for initializing the internal instance of \( \mathcal{L}A(Q) \)-Solver (called bb_solver_) to a LP relaxation of the current ILP problem and resetting the value of the shared variable \( z_{ILP} \), memorizing the optimal cost function value, to \(+\infty\) (rows 1-2). After that, the branch and bound subroutine is invoked on the root of the search tree at row 3 and, upon its termination, all data structures cleared (row 4) and the optimal value \( z_{ILP} \) returned to the calling function (row 6).

Procedure opt_lia_bb_node encodes a recursive pre-order traversal of the search tree implicitly defined by the branch and bound approach. At rows 1-3 the continuous relaxation \( \mathcal{P}^i \) of the ILP associated to node \( N^i \) is checked to be satisfiable, if the condition is not met then the function immediately backtracks because \( \mathcal{X}^i = \mathcal{P}^i = \emptyset \). Otherwise, at line 4, the optimal solution \( z_{LP}^i \) of the LP relaxation is computed us-
ing the minimization procedure for $OMT(\mathcal{LA}(\mathbb{Q}))$. Now the condition $z_{ILP}^i \geq z_{ILP}$ is checked at rows 5-7, if it holds then entire search sub-tree can be immediately discarded by a return statement. The reason for this is that $z_{ILP}^i$ is a lower bound for all $z_{ILP}^j$ of the nodes $N^j$ that belong to the sub-tree of node $N^i$. If the current solution $\pi^i$ is found to be integral by procedure `internal_bb_model_is_integral()`, then the search shared optimum value $z_{ILP}$ is updated to the new value $z_{LP}$ (rows 8-10). Otherwise, it is necessary to partition the polyhedron $\mathcal{P}^i$ into two halves by branching on a fractional variable within the current model $\overline{x}^i$ (rows 12-19). This partitioning is done by procedure `get_bb_branch()`, which automatically selects the branching variable $x_k$ and randomly assigns the branching linear equations $\{(x_k - \lfloor q_k \rfloor \leq 0), (-x_k + \lceil q_k \rceil \leq 0)\}$ to the left and right branches of a BBFrame instance. Then both sub-problems are sequentially explored by means of a temporarily push and pop of their own cut on the current LP tableau interleaved by a recursive call to `opt_lia_bb_node`.

\[ \mathcal{P}_0 \]

\[ \vdots \]

\[ \mathcal{P}_k \]

\[ x_{ILP}^1 \]

\[ \text{c}^\top x_{ILP}^2 < \text{c}^\top x_{ILP}^1 \]

\[ \mathcal{P}_j \]

\[ \vdots \]

\[ \mathcal{P}_k \]

\[ x_{ILP}^2 \]

\[ \vdots \]

\[ \mathcal{P}_j \]

\[ \perp \]

**Figure 6.2.** An example of ILP optimization with *basic branch and bound*, the search stops when all nodes of the tree have been examined or fathomed;
The implementation of algorithm 6.1 mirrors one to one the theoretical setting defined in chapter 2, and it is thus guaranteed to terminate with the optimal solution to the original ILP problem.

Figure 6.2 shows an example of execution of algorithm 6.1 on a generic ILP problem. The search proceeds using a one-step backtracking approach on a search tree which number of nodes is exponential in the number of integer variables in the problem. Only when a node LP-relaxation is found to be unsatisfiable or its optimal rational value exceeds the last saved ILP solution, an entire sub-tree can be discarded at once.

Figure 6.3 shows a situation in which this algorithm performs particularly bad on an ILP problem, after the branch and bound procedure has made a sequence “wrong” decision in terms of branches, restricting the initial polyhedron $P_0$ to a non-empty polyhedron $P_i$ that does not contain any integral solution. In this case, the basic branch and bound algorithm has no instrument to detect such a situation, and it is thus forced to still explore the entire sub-tree instead of immediately discarding it.

Another issue of algorithm 6.1 is its completeness, since within the CDCL/DPLL search framework it might be seldom more convenient to perform quick non-exhaustive searches and return to the main loop for some more truth assignment propagation rather than an heavy complete minimization call.

2. Advanced Branch and Bound

As we have been mentioning in the previous section, one of the main disadvantages of algorithm 6.1 is that whenever a “wrong” choice is made, only after its entire sub-tree has been traversed it is possible to recover from it. The current state-of-the-art implementation of the $\mathcal{L}_A(\mathbb{Z})$-Solver satisfiability checker already provides the instruments required to overcome this issue with a back-jumping technique that skips entire subtrees whenever such a situation is detected.

The general idea of the new algorithm is to avoid the burden of a slow complete branch and bound search, and to rather rely on a sequence short-and-quick SAT calls to the $\mathcal{L}_A(\mathbb{Z})$-Solver. Each call seeks for a solution better than the previous one (if any) for the same fixed truth assignment and stops at the first integral solution $x^*$ found. An example of this scheme is depicted in figure 6.4. Using repeated calls of branch and bound allows to better exploit historical information to make such calls as fast and
Figure 6.3. The figure shows that basic branch and bound algorithm 6.1 might still visit all nodes of an entire sub-tree in which $\mathcal{X} = \emptyset$, because the fathoming condition $\mathcal{P} = \emptyset$ evaluates to false;
efficient as possible. Another advantage of this technique is that it makes it possible to benefit from all optimization techniques already implemented within the $\mathcal{L}A(\mathbb{Z})$-Solver for satisfiability.

The function `minimize_backjumping`, depicted in figure 6.5, is responsible for data structure initialization and ensuring that the overall search returns the correct optimum value $z_{ILP} \in X$. It starts by initializing the internal instance of $\mathcal{L}A(\mathbb{Q})$-Solver (the `bb_solver_`), to the LP relaxation of the current ILP problem (row 1). Then it enters a loop, spanning from row 3 to 12, which explores the search space by means of a sequence of linear cuts ($cost\_var < z_{ILP}$) that progressively refine the initial LP relaxation so that the current known optimum solution $x^*_{ILP}$ is removed from the search space. After such cut is added to the LP tableau at row 4, the latter is checked with the simplex method to verify that the search space is not empty (rows 5-7). If that is not the case, a new integral solution $x^*_{ILP}$ is sought with the aid of subroutine `opt_internal_bb()` (row 8). If a new solution is found, then the procedure returns sat
minimize_backjumping (const TheoryLitSet& to_ignore, int limit)
1. if (internal_bb_init(to_ignore, true)) then
2. loop
3. constr = (cost_var < z_{ILP})
4. bb_solver_-=>push_constraint(constr);
5. if (! bb_solver_-=>check()) then
6. break;
7. end if
8. if (unsat == opt_internal_bb(to_ignore, limit)) then
9. break;
10. end if
11. bb_solver_-=>pop_constraint();
12. end loop
13. bb_solver_-=>pop_constraint();
14. internal_bb_deinit();
15. end if
16. return z_{ILP};

Figure 6.5. minimization through repeated calls to LA(Z) satisfiable checks

and a new iteration of the loop with a stricter bound on the value of the cost variable is started. Otherwise, if unsat is returned it means that there are no integral solutions within the current LP relaxation and that the latest value found for z_{ILP} is the minimum. The internal solver is thus de-initialized (rows 13-14), and the optimum value returned (row 16).

It is easy to see that this minimization approach always terminates, because at each iteration of the loop the search space is restricted by means of a cut that excludes at least one possible solution x^*_{ILP} \in \mathcal{X}, and \mathcal{X} is finite by assumption.

It is worth examining into detail the inner working of procedure opt_internal_bb(), which pseudo-code is shown in figure 6.6. Though we designed it around the same principles and structure of the branch and bound LA(Z) satisfiability checker implemented in MATHSAT5, it distinguishes itself from it as a result of an additional minimization step of the LP relaxation that heuristically helps into leading the search toward the actual ILP optimal solution.

Let’s take for example the polytopes shown in figure 6.7-a. After a preliminary LA(Q) minimization step, the branch and bound might select the cuts \( y \geq \lceil 3.5 \rceil \) or \( x \leq \lfloor 2.5 \rfloor \), yet these cuts are ILP-unsatisfiable therefore the algorithm will immediately backtrack and find the optimal ILP solution by means of a single call to opt_internal_bb. Conversely, with a polytopes like the one shown in figure 6.7-b, by
opt_internal_bb (const TheoryLitSet& to_ignore, int limit)
1. bool backtrack = false;
2. loop
3. if (backtrack) then
   backtrack = false;
4. confl = bb_stack_.back().conflict;
5. loop
6. bb_solver_->pop_constraint();
7. bb_stack_.pop()
8. if (bb_stack_.size() == 0) then
   return unsat;
9. branch = bb_stack_.back()
10. if (branch.right_constr > 0 && confl.contains(constr_in(branch.left_constr))) then
    break
11. else
12. bb_stack_.push(find_bb_branch());
13. end if
14. BBFrame frame = bb_stack_.back();
15. if (frame.left_constr > 0) then
   frame.left_constr = - frame.left_constr;
16. bb_solver_->push_constraint(frame.left_constr);
17. if (!bb_solver_->check()) then
   frame.update_conflict(bb_solver_->get_conflict());
18. bb_solver_->pop_constraint();
19. else
   continue;
20. end if
21. end if
22. if (frame.right_constr > 0) then
   frame.right_constr = - frame.right_constr;
23. bb_solver_->push_constraint(frame.right_constr);
24. if (!bb_solver_->check()) then
   frame.update_conflict(bb_solver_->get_conflict());
25. bb_solver_->pop_constraint();
26. backtrack = true;
27. end if
28. end if
29. end loop

Figure 6.6. LA(Z) branch and bound procedure with minimization
2. ADVANCED BRANCH AND BOUND

Figure 6.7. In case a) the optimal ILP solution is found in a single step by algorithm 6.6, whereas the solution found in case b) might be sub optimal since it depends on the random choices made during branch and bound.
branching on variable $x$ both partitions of the search space contain at least one integral solution. Therefore, choosing to take the “wrong” branch will lead to a suboptimal ILP solution. It is possible to overcome this issue by means of a forward-looking strategy that preventively computes the LP optimal solutions $z_{LP}^i, z_{LP}^j$ of partitions $P^i, P^j$ and explores the branch $N^k$ s.t. $z_{LP}^k = \min(z_{LP}^i, z_{LP}^j)$. In this manner, the search algorithm selects for first the partition $P^k$ that contains the actual ILP optimal solution or one that has no integral solution at all. In the latter case, thanks to the back-jumping mechanism, the “wrong” choice will be eventually recovered and the overall procedure will lead to the optimal ILP solution anyway.

In my thesis work, we verified that it wasn’t necessary to implement this forward-looking strategy because experimental data on real world problems has shown that in 99% of cases procedure opt_internal_bb() could find the optimal ILP solution without further iterations, probably due to the nature of problems being benchmarked.

The branch and bound procedure 6.6 starts at row 1 by setting the backtrack flag to false, therefore the first time the main loop is executed the condition at line 3 is false and the execution resumes at row 18. The current LP relaxation, loaded in the internal $\mathcal{L}A(Q)$-Solver instance named bb_solver_, is minimized with the simplex method. If its model is integral, then current ILP optimal solution $z_{ILP}$ is updated with the value $z_{LP}$ retrieved from the current LP relaxation and, since this branch and bound is designed to stop at the first integral solution found, sat is returned (rows 19-21). Otherwise the algorithm selects a branching variable $x_k$ from those in base with a fractional value, so that the current polyhedron $P^i$ is partitioned in two halves $P^i_a, P^i_b$ s.t. $\{P^i_a \cup P^i_b\} \cap \mathbb{Z}^n = \mathcal{X}$. As usual, this is done by randomly assigning the cuts $\{(x_k - \lfloor q_k \rfloor \leq 0), (-x_k + \lceil q_k \rceil \leq 0)\}$ to the left and right branches of a BBFrame instance which is immediately pushed on a stack (row 23). Then the algorithm goes at row 26, where it retrieves the latest branching frame pushed on the stack to explore it. If it didn’t already explore the left branch (test $frame.left._constr > 0$), then it pushes its corresponding linear equation on the current tableau and checks it consistency (rows 28-30). If the tableau is consistent then it means that the new LP relaxation is non empty, and the main loop restarts (rows 33-35). Otherwise $P^i_a$ is empty, therefore the search should discard the current node and examine the right branch instead (rows 31-33). The solver annotates the frame with the linear equations that resulted in the unsatisfiability of the current branch for later use. Then the execution moves at lines 37-45, were the
right branch is examined. The code is almost exactly the same as for the left branch, though this time when an inconsistency is detected the backtrack flag is set to true so that when the loop is restarted a different execution flow will take place. The inner working of the backtracking mechanism is simple. It starts by picking up the set $\eta$ of linear equations that caused the unsatisfiability of both branches of the current branch and bound frame (row 5). Then it discards one stack at a time up until when the stack is empty (rows 9-11), meaning that there are no integral solution within $P$ and unsat should be returned, or the right branch of the top stack frame has not yet been examined and its left branch belongs to the conflict set $\eta$ (rows 12-15), meaning that at that point a “wrong” choice was made on the branch to be taken. After a back-jump, the execution resumes at line 37, because all other conditions evaluate to false.

Figure 6.8 shows how, differently from the basic branch and bound case depicted in figure 6.3, the new advanced algorithm successfully exploits back-jumping to immediately skip a the entire left sub-tree of node $P_1$ as soon a conflict $\eta = \{x_0 \leq \lfloor r_0 \rfloor, x_1 \leq \lfloor r_1 \rfloor\}$ is detected on node $P_{k+1}$. Though it should be noted that the effectiveness of this back-jumping mechanism depends on the capability of the $LA(Z)$-Solver to find the minimum conflict set causing the unsatisfiability of node $P_{k+1}$, it is also true that only in the worst case this approach degenerates to the simple one-step backtracking approach.

The actual opt_internal_bb() implementation makes use of the input parameter limit to stop the branch and bound procedure and return to the calling function whenever the search takes too long. It is in fact at times more convenient to restart the search and rely on the randomization factor affecting branch choice to reach a solution faster.

3. Truncated Branch and Bound

As we observed in the previous section, in the 99% of cases it has been sufficient a single call to procedure opt_internal_bb() to immediately find the optimal ILP solution $z_{ILP}$ for the current Boolean assignment $\mu$. Hence, procedure minimize_backjumping() would execute only 2 iterations of the main loop, the second one only to prove that the value $z_{ILP}$ found the first time was really optimal. Since the overall CDCL/DPLL optimization procedure can work correctly even when it is fed sub-optimal values by the $LA(Z)$ minimization procedure, we have chosen to develop a version of OPTIMATH-
Figure 6.8. The figure shows how the advanced branch and bound search is able to exploit the conflict set \( \{x_0 \leq \lfloor r_0 \rfloor, x_1 \leq \lfloor r_1 \rfloor\} \) information of node \( P_{k+1} \) to directly backtrack to the right branch of node \( P_1 \), where the “wrong” decision – the one causing \( X = \emptyset \) – was made;
SAT in which minimize_backjumping() would always stop at the first integral solution found.
Chapter 7

Performance Evaluation

When it comes to real world applications it is not only essential to deliver a correct result, but to maintain uniformly efficient performances for all inputs and to be as much as fast as possible. Therefore, this thesis could not be considered finished without an additional work of performance evaluation of the newly implemented OPTIMATHSAT version. Here we will start by describing the benchmark test-bed in section 1, and finally conclude the thesis discussion with the results shown in section 2.

1. Test-bed setup

The field of OMT is relatively new, therefore by the time we started collecting experimental data there weren’t many competitor SMT tools available for a comparison. Thus, we decided to look for a challenge directly within the constrained programming (CP) research community, despite the fact OPTIMATHSAT would obviously be at a disadvantage due to its young, unripened traits. The solvers have been tested on two identical machines with the following specifics:

- \textit{cpu}: Intel(R) Xeon(R) CPU E5-2407 0 @ 2.20GHz, 8 cores;
- \textit{ram}: 64384 MB;
- \textit{swap}: 65487 MB;
- \textit{os}: ubuntu 12.04;
- \textit{kernel}: 3.8.0-29-generic;
Note that Optimathsat has been developed with C++, cmake technologies on an unix system.

**Optimathsat configurations**

As we extensively discussed in part II, we extended Optimathsat with an integer optimization framework that can make use of different strategies and heuristics to accomplish the same task. For this reason one of the goals of this testing phase was to identify which configuration, among those in the following list, delivers the best performances.

1. *linear_trunc-rep-bb-min*: the CDCL/DPLL main loop uses linear-search strategy mode, while the \texttt{minimize()} call to the LA$(\mathbb{Z})$-Solver uses the truncated advanced branch and bound implementation described in section 3;

2. *linear_full-rep-bb-min*: the CDCL/DPLL main loop is run in linear-search strategy mode, while the \texttt{minimize()} call to the LA$(\mathbb{Z})$-Solver uses the complete advanced branch and bound implementation described in section 2;

3. *linear_classic-bb-min*: the CDCL/DPLL runs in linear-search mode, while minimization uses the classic branch and bound implementation described in section 1;

4. *binary_classic-bb-min*: the CDCL/DPLL runs in binary-search mode, while minimization uses the classic branch and bound implementation described in section 1;

5. *binary_full-rep-bb-min*: runs in binary-search mode, uses the complete advanced branch and bound implementation described in section 2;

6. *binary_trunc-rep-bb-min*: runs in binary-search mode, uses the truncated advanced branch and bound implementation described in section 3;

7. *adaptive1_full-rep-bb-min*: uses the dynamic-search mode with the estimator $r_{bin}^1$ for binary cycles, the minimization is performed using the complete advanced branch and bound implementation described in section 2;
8. *adaptive1_trunc-rep-bb-min*: uses the *dynamic-search* mode with the estimator $r_{\text{bin}}^1$ for binary cycles, the *minimization* is performed using the truncated advanced branch and bound implementation described in section 3;

9. *adaptive2_full-rep-bb-min*: uses the *dynamic-search* mode with the estimator $r_{\text{bin}}^2$ for binary cycles, the *minimization* is performed using the complete advanced branch and bound implementation described in section 2;

10. *adaptive2_trunc-rep-bb-min*: uses the *dynamic-search* mode with the estimator $r_{\text{bin}}^2$ for binary cycles, the *minimization* is performed using the truncated advanced branch and bound implementation described in section 3;

11. *binary_no-min*: runs in *binary-search* mode, the calls to function *minimize()* perform no optimization within the $\mathcal{LA}(\mathbb{Z})$-Solver and return the current cost function value only;

Though the list might seem redundant at a first glance, as it will be later shown in the results section 2, the performances of these configurations do not always respect the basic intuition.

**Benchmarks**

In order to limit the skewness given by the domain in which benchmarks are being taken, two sets of input problems have been used.

We took the first set of benchmarks from the Minizinc Challenge 2012 [2012], a contest in which many of the best performing CP solvers participate. We filtered this set of benchmarks to include linear integer optimization problems only, except for those using non-linear arithmetic which is not yet supported by OPTIMATHSAT. This led to a total of 676 problems being tested for each solver 1.

Note that the models benchmarked at Minizinc Challenge 2012 are encoded into a constraint modelling language known as MiniZinc, in contrast to the SMT-LIB format widely adopted within SAT and SMT communities. This incompatibility is not an issue,

---

1. The final set of problems was thus restricted to the following model templates: bacp, curriculum, cutstock, cyclic-rcpsp, ghoumb, golomb, grid-colouring, jobshop, jobshop2, open_stacks, p1f, photo, prize-collecting, prop_stress, radiation, roster, ship-schedule, shortest_path, still_life, table-layout, template_design and vrp
thanks to a couple of language translators I developed during my research project experience in my master studies. Though this should not be considered as part of the thesis work, I will briefly describe these useful encoders in appendix A for completeness.

We took the second set of benchmarks from the SmtLibv2 repository, in particular we selected bounded verification problems – Bounded Model Checking (BMC) of invariants and K-Induction – of a well-known parametric timed system, Fisher’s Protocol. BMC [resp. K-Induction] takes a Finite-State Machine $M$, an invariant property $\psi$ and an integer bound $k$, and produces a propositional formula $\varphi$ which is satisfiable [resp. unsatisfiable] if and only if there exists a $k$-step execution violating $\psi$ [resp. a $k$-step induction proof that $\psi$ is always verified]. The approach we have used leveraged to real-time systems by producing $\text{SMT}(\mathcal{L}A(\mathbb{Z}))$ formulas rather than purely-propositional one. Fischer’s Protocol ensures the mutual exclusion among $N$ processes using real-time clocks and a shared variable. The problem is parametric into two positive integer values, $\delta_1$ and $\delta_2$, describing the discrete delay of some actions. Mutual exclusion, and other properties included in the SAL model, are verified if and only if $\delta_1 < \delta_2$. Our $OMT(\mathcal{L}A(\mathbb{Z}))$ problems have been produced as follows. We fixed the value of $\delta_2$ to 4 and then generated six groups of formulas for each type of problem solved (BMC or K-Induction) and property addressed (e.g. mutex, mutual-exclusion, time-aux3 and logical-aux1). For each group, for increasing values of $N \geq 2$ and for a set of big values of $k \geq k^*$, with $k^*$ ranging from 5 to 10 depending on the problem, we used SAL to produce the corresponding parametric $\text{SMT}(\mathcal{L}A(\mathbb{Z}))$ formulas, and asked the tool under test to find the minimum value $\delta_1$ which made the resulting formula $\mathcal{L}A(\mathbb{Z})$-satisfiable. The final set of benchmarks was then restricted to a selection of 230 problems, due to the fact that some of the original problems were causing the tool mzn2fzn to crash and could not be successfully translated into FlatZinc format.

**Competitor Tools**

The tools chosen for being compared to OPTIMATHSAT are:

1. `flatzinc`;


4. *opturion* – first classified at *MiniZinc* challenge 2013 in the *free* and *fixed* categories;

The initial selection criteria was to include all the tools that reached the highest ranks across the several challenges. However, this has not been possible for a number of reason. For first, some tools are licensed and not freely distributed, with others there have been technical issues with the machines used to run the tests that could not be solved in time. During this phase one of the main difficulties has been dealing with a particularly mottled collection of tools with different types of restrictions on the behaviour or the input models, as well as different output formats.

### 2. Empirical Results

The detailed empirical results for the *MiniZinc Challenge 2012* set of benchmarks are shown table 7.1. The same data is presented in figure 7.1, which puts at a comparison the cumulative solve time taken by each solver and the number of input models correctly solved.

<table>
<thead>
<tr>
<th>Benchmark:</th>
<th>No. Problems:</th>
<th>No. Solved:</th>
<th>No. Invalid</th>
<th>Solve Time (s.):</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear_trunc-rep-bb-min</td>
<td>676</td>
<td>173</td>
<td>0</td>
<td>25469.504</td>
</tr>
<tr>
<td>linear_full-rep-bb-min</td>
<td>676</td>
<td>172</td>
<td>0</td>
<td>24861.788</td>
</tr>
<tr>
<td>linear_classic-bb-min</td>
<td>676</td>
<td>164</td>
<td>0</td>
<td>24052.944</td>
</tr>
<tr>
<td>binary_classic-bb-min</td>
<td>676</td>
<td>159</td>
<td>0</td>
<td>21055.296</td>
</tr>
<tr>
<td>binary_full-rep-bb-min</td>
<td>676</td>
<td>168</td>
<td>0</td>
<td>21575.296</td>
</tr>
<tr>
<td>binary_trunc-rep-bb-min</td>
<td>676</td>
<td>167</td>
<td>0</td>
<td>22103.808</td>
</tr>
<tr>
<td>adaptive1_full-rep-bb-min</td>
<td>676</td>
<td>169</td>
<td>0</td>
<td>22246.752</td>
</tr>
<tr>
<td>adaptive1_trunc-rep-bb-min</td>
<td>676</td>
<td>167</td>
<td>0</td>
<td>21269.928</td>
</tr>
<tr>
<td>adaptive2_full-rep-bb-min</td>
<td>676</td>
<td>168</td>
<td>0</td>
<td>22703.732</td>
</tr>
<tr>
<td>adaptive2_trunc-rep-bb-min</td>
<td>676</td>
<td>167</td>
<td>0</td>
<td>22531.452</td>
</tr>
<tr>
<td>binary_no-min</td>
<td>676</td>
<td>105</td>
<td>0</td>
<td>6744.552</td>
</tr>
<tr>
<td>flatzinc</td>
<td>676</td>
<td>244</td>
<td>0</td>
<td>31685.312</td>
</tr>
<tr>
<td>opturion</td>
<td>676</td>
<td>364</td>
<td>0</td>
<td>16970.24</td>
</tr>
<tr>
<td>JaCoP</td>
<td>676</td>
<td>169</td>
<td>238</td>
<td>4565.66</td>
</tr>
<tr>
<td>Gecode</td>
<td>676</td>
<td>251</td>
<td>0</td>
<td>12556.896</td>
</tr>
</tbody>
</table>

**Table 7.1.** Summary of solvers performances on *MiniZinc Challenge 2012* benchmarks.
Figure 7.1. Cumulative solve time for all solvers tested with MiniZinc challenge benchmarks as input.
Overall, the best solver on this set of benchmarks is `opturion`, which solves 364 out of 676 models within the time limit of 20 minutes. The absolute worst one is JaCoP, due to the fact that at least 238 of the solutions it found have been proved sub-optimal by the z3 solver.

The OptiMathSAT version that performed the worst is `binary_no-min`, with 105 over 676 benchmarks solved only. This abides by my expectations, because the cuts in the form in the form \((\text{cost}_\text{var} < u)\) that the CDCL/DPLL main loop pushes on the input formula stack \(\varphi\) are weak, due to the absence of any \(LA(\mathbb{Z})\)-Solver optimization phase for the value \(u\) for the current truth assignment \(\mu\).

Surprisingly, the version of OptiMathSAT that outperformed the others is `linear_trunc-rep-bb-min`, with its 173/676 benchmarks solved in 25470 seconds. Recall, from section 1, that this configuration runs the CDCL/DPLL main loop in linear mode and does not guarantee the optimality of the result of minimization within the \(LA(\mathbb{Z})\)-Solver. The fact that `linear-search` beats `binary-search` mode is coherent with the observation that the latter configuration causes a greater number of unsat calls which take more time to be explored. As we have mentioned in section 3, the branch and bound procedure finds the optimal value through a single call to `opt_internal_bb()` in 99% of times, which means that is a full waste of time trying to ensure its optimality with additional checks rather then immediately return to the CDCL/DPLL main loop.

Overall, we note that the various versions of OptiMathSAT are pretty much aligned in terms of performances, except for the worst one. This, together with the observation that all CP solvers outperform OptiMathSAT, leads to the conclusion that so far the impact of the different kinds of configurations is very little.

Table 7.2 shows the results for the Fischer set of benchmarks. Differently from the previous set of problems, this time OptiMathSAT clearly outperforms all the CP solvers. The `binary_no-min` configuration still stands behind the others, with only 192 out of 230 benchmarks solved. The best version is `adaptive1_full-rep-bb-min`, solving 195/230 benchmarks in only 17160 seconds. The other configurations achieve the same number of solutions with a maximum time gap of 1.18%. This means that, though the presence of any type of minimization at the \(LA(\mathbb{Z})\)-Solver level helps, the search is dominated so much by the Boolean constraint propagation at the CDCL/DPLL that it doesn’t even make a difference which strategy is chosen to explore the solution space.
### Table 7.2. Summary of solvers performances on Fischer benchmarks.

<table>
<thead>
<tr>
<th>Benchmark:</th>
<th>No. Problems</th>
<th>No. Solved</th>
<th>No. Invalid</th>
<th>Solve Time (s.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear_trunc-rep-bb-min</td>
<td>230</td>
<td>195</td>
<td>0</td>
<td>17280.128</td>
</tr>
<tr>
<td>linear_full-rep-bb-min</td>
<td>230</td>
<td>195</td>
<td>0</td>
<td>17334.632</td>
</tr>
<tr>
<td>linear_classic-bb-min</td>
<td>230</td>
<td>195</td>
<td>0</td>
<td>17188.008</td>
</tr>
<tr>
<td>binary_classic-bb-min</td>
<td>230</td>
<td>195</td>
<td>0</td>
<td>17216.716</td>
</tr>
<tr>
<td>binary_full-rep-bb-min</td>
<td>230</td>
<td>195</td>
<td>0</td>
<td>17205.304</td>
</tr>
<tr>
<td>binary_trunc-rep-bb-min</td>
<td>230</td>
<td>195</td>
<td>0</td>
<td>17362.52</td>
</tr>
<tr>
<td>adaptive1_full-rep-bb-min</td>
<td>230</td>
<td>195</td>
<td>0</td>
<td>17159.892</td>
</tr>
<tr>
<td>adaptive1_trunc-rep-bb-min</td>
<td>230</td>
<td>195</td>
<td>0</td>
<td>17261.576</td>
</tr>
<tr>
<td>adaptive2_full-rep-bb-min</td>
<td>230</td>
<td>195</td>
<td>0</td>
<td>17182.512</td>
</tr>
<tr>
<td>adaptive2_trunc-rep-bb-min</td>
<td>230</td>
<td>195</td>
<td>0</td>
<td>17349.844</td>
</tr>
<tr>
<td>binary_no-min</td>
<td>230</td>
<td>192</td>
<td>0</td>
<td>17096.6</td>
</tr>
<tr>
<td>flatzinc</td>
<td>230</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>opturion</td>
<td>230</td>
<td>1</td>
<td>0</td>
<td>0.004</td>
</tr>
<tr>
<td>JaCoP</td>
<td>230</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Gecode</td>
<td>230</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

So far, the empirical data has shown two apparently contradicting results: on the set of benchmarks designed for the MiniZinc Challenge 2012 the CP solvers clearly win, whereas on the benchmarks taken by the SmtLibv2repository OPTIHAMTSAT wallop them. This sharp difference arises one question: is it possible that the measured performance gap depends largely on the data sets used as inputs? As we tried to answer this question, we had to collect some statistics from the input models that could, from a theoretical point of view, affect or characterize the performance of the solvers under the assumption that the strength of OPTIHAMTSAT in respect to the CP solvers relies in its capability of efficiently handling disjunctive reasoning. This is the least of used estimators:

- **no. of bool variables** in the FlatZinc model (fig. 7.2) and in the SmtLibv2 model (fig. 7.3), generally greater in problems with higher disjunctive reasoning;

- **no. of integer variables** in the FlatZinc model (fig. 7.4) and in the SmtLibv2 model (fig. 7.5), generally greater in problems with heavier integer reasoning;

- **rough number of clauses** in the FlatZinc model (fig. 7.6) and in the SmtLibv2 model (fig. 7.7), generally greater in problems with heavier disjunctive reasoning;
• number of conflicts (fig. 7.8) and theory conflicts (fig. 7.9) experienced during OPTIMATHSAT search, configured in linear mode;

Figures from 7.2 to 7.9 show the result of this additional step of data collection, for each estimator listed above there is a graph split up on two sides, the left one reserved for the set of benchmarks taken from the MiniZinc Challenge 2012 and the right one for Fischer ones. Each graph has the estimator value on the $x$ axis and the ratio among the solving time required by OPTIMATHSAT linear_trunc-rep-bb-min configuration and opturion on the $y$ axis.

On none of these graphs can we see a clear sign of the hypothesized correlation among the estimator and the corresponding models, in other words, the difficulty of the input models is not captured by any of the estimators used.

This negative result suggests that there is something deeper going on related to the way in which the original problems are encoded into models. In fact, in both cases the input models are designed to exploit the inherent strengths of the target solvers. For example, while disjunctive reasoning is usually encoded with Boolean formulas within SmtLibv2, in FlatZinc models it is usually encoded with pseudo-Boolean constraints that map Boolean variables on integer ones, disjunction on addition and conjunction on multiplication. Due to the different characteristics of the CP/SMT solvers, it is clearly not possible to define a shared model encoding that is fair with both categories of solvers. To overcome this issue, there are two roads that can be taken:

1. a first possibility is to identify a certain number of domain problems for which it can be developed a specialized model encoding exploiting the characteristics of the target solvers in both input languages;

2. a second possibility is to further develop the model transcoders described in appendix A, so that they can abstract-away the snippets of input models encoded in a solver friendly way and completely restructure them so that they fit the characteristics of the new target language and solvers;
Figure 7.2. Number of Boolean Variables in FlatZinc models.
Figure 7.3. Number of Boolean Variables in SmtLib2 models.
Figure 7.4. Number of integer variables in FlatZinc models
Figure 7.5. Number of integer variables in SmtLib2 models.
Figure 7.6. Rough number of clauses in FlatZinc models
Figure 7.7. Rough number of clauses in SmtLibv2 models
Figure 7.8. Number of Boolean conflicts (OPTIMATHSAT)
Figure 7.9. Number of theory conflicts (OPTIMATHSAT)
Chapter 8

Conclusions and further work

As a result to this thesis work, now OPTIMATHSAT fully supports optimization over linear arithmetic with integer cost function $\text{OMT}(\mathcal{LA}(\mathbb{Z}) \cup T)$, where $T$ can be the empty set or any other first-order theory already supported by MATHSAT5 (e.g. $\mathcal{LA}(\mathbb{Q})$, $\mathcal{EUF}$, $\mathcal{AR}$, $\mathcal{BV}$). Not only the optimal solution value can be found, but its corresponding model can be dumped through the satisfiable assignment $\mu$.

The linear-search strategy coupled with a truncated branch and bound optimization approach revealed to be the best configuration for the input set of ILP problems. This result goes against the basic intuition, common among people with Computer Science background, that binary approach is always better than linear. As we have shown, that is not the case in our experimental framework due to the non-uniform solving time required by these two different approaches: binary-mode generally causes more unsat calls within the satisfiability checker than linear-mode, and these calls are much more expensive than satisfiable ones. The experimental data has also shown that the search largely benefits from the presence of a simplex-based branch and bound minimization step within the $\mathcal{LA}(\mathbb{Z})$-Solver, because it results in cuts that restrict the size of the search space in more depth.

Though OPTIMATHSAT obtained an absolute victory on Fischer input problems, characterized by high level of disjunctive reasoning as long as integer optimization, it has not been able to keep up the pace of CP solvers when ran on the benchmarks taken from the MiniZinc Challenge 2012.
We have then analysed the two input sets of benchmarks in order to identify which notable model characteristic, among those theoretically viable, is at the root of this sharp performance discrepancy in our measurements. This step of automated problem-analysis resulted into a negative response, showing no clear correlation among the selected estimators and the performance issues.

A further step of manual human-driven analysis revealed that the two sets of benchmarks adopt different encoding approaches to the problems of interest that might advantage one category of solvers in respect to the other. Some of these differences (e.g. the mapping of Boolean variables into Integer ones, the mapping of disjunction into addition and conjunction into multiplication) prevent OPTIMATHSAT from exploiting the full potential of its efficient DPLL/CDCL engine during the search and are a clear source of performance hit, despite it being not clearly quantified. The same encoding problems hold when problems are translated in the other direction, from SmtLibv2 domain into that of CP solvers, because the disjunctive/boolean reasoning is not mapped into pseudo-boolean constraints at the benefit of CP solvers.

Thus, future work should revolve around the development of a more sophisticated formulae translator that is able to re-encode complex sub-structures so that the output problem is optimized for the type of solver being targeted. Alternatively, the efforts should go in the direction of developing a library of benchmark problems with two separate encodings, each optimized for a different target type of solver. One of these two approaches might help removing the fairness problem experienced with the sets of benchmarks used in this research, and allow for a more proper comparison among the two category of tools.
Bibliography

BIBLIOGRAPHY


Appendix A. Model Encoders

In order to allow the comparison of the MATHSAT and OPTIMATHSAT tools performances to the solvers participating to the MiniZinc Challenge, we had to use a couple of compilers able to translate a model from the SmtLibv2 language to MiniZinc and vice-versa. These translators, have been developed by the author in a research project as part of his master studies, and separately from this thesis work. Detailed documentation on the model translation scheme can be found in [Trentin, 2013].

The SL2toMZ and FZtoSL2 compilers glue together with the third party mzn2fzn tool, as shown inf figure 8.1, to provide a bidirectional translation among three modelling languages: FlatZinc, MiniZinc and SmtLibv2.

Figure 8.1. The translation scheme implemented to allow benchmark comparison among the tools

This design choice has two clear advantages:
• it allows the mzn2fzn tool to apply optimizations during the flattening of the MiniZinc model toward the representation that best fits its usage;

• it simplifies the SL2toMZ tool logic, as a result of the similarities inbetween the syntax of the two languages;

Currently, FZtoSL2 and SL2toMZ tools translate only a restriction of the domain language, due to the fact that some features do not have any corresponding syntax construct in the target language. These limitations are deeply documented in [Trentin, 2013].

Here are the input modelling languages being used:

• **MiniZinc** is a Constraint Satisfaction Programming (CSP) and Constraint Optimization Programming (COP) modelling language that allows for modelling problems for a range of different solvers using Finite Domain (FD) and Linear Programming (LP) techniques. Some examples of such solvers include: Gecode, ECLiPSe, ILOG Solver, Minion, Choco.

MiniZinc is a first order language that provides three scalar types (Booleans, integers and floats) and two compound types (sets and arrays of fixed size), which can be instantiated as parameters or decision variables in the case of integers, floats, Booleans and set of integers.

Annotations can be used to add non-declarative information, such as search strategies, and solver specific information, such as variable representations, to be layered on top of declarative models.

From a MiniZinc it can be generated, through a flattening phase followed by some post-flattening operations, an equivalent FlatZinc model. The FlatZinc language is a low-level solver-input language that allows solvers to support MiniZinc with a minimum of effort.

• **SmtLibv2** standard is the result of an international effort to develop a common languages and interfaces for SMT solvers. The version 2.0 of the SmtLibv2 standard is indeed a major upgrade that simplified and extended the language, including a new command language to interface with a SMT solver. This language contains formalizations of arithmetic, arrays, bit vectors, algebraic data types, equality with uninterpreted functions and various combinations of these theories.
Acknowledgements

All people risk to live an entire life unaware of much, on certain occasions, they have been invaluable and irreplaceable to others, even when perfect strangers. Chances to show gratefulness in this fast-evolving modern world are very few, this is one of them.

My first thought goes to Giandomenico Orlandi, professor in mathematical analysis at University of Verona, who inspired me at an early stage of my studies and gave me the motivation to keep up. Then I would like to thank Roberto Sebastiani, professor in formal verification at University of Trento, who accepted to be my advisor in this thesis work as well as during my research project. He did not only grant me plenty of helpful advices when a pivotal issue with the development library arose, but kept up encouraging me exploring new solutions and approaches all along.

Thanks to Alberto Griggio, researcher at Fondazione Bruno Kessler (FBK) and major developer of MATHSAT5, for his constant availability at clarifying me MATHSAT5 internals and allowing me to quote part of his research work. Thanks to Silvia Tomasi, whose pioneering PhD study on $OMT(\mathcal{LA}(\mathbb{Q}) \cup T)$ has been the basis for the successfulness of my thesis work, that allowed me to quote part of her research as preliminary background.

A special thank goes to Michele Welponer, IT administrator of the server on which I ran the experiments, for his constant assistance and the patience he demonstrated despite of the frequent memory-due kernel panics I caused on the machines at the early stages of the benchmarking phase.

A heartily thank goes to my parents, who contributed to materially support my studies and my sisters, for their very presence. A thank to the friends who stubbornly endured at my side all these years.

Lastly, I would like to express my gratefulness to all public institutions – Universities of Verona and Trento included – that provided me the material, educational and financial support to continue my studies in Computer Science and thus to grow a sincere interest in the field of formal verification and automated reasoning.