UNIVERSITÉ JOSEPH FOURIER - GRENOBLE I

THÈSE

pour obtenir le grade de
UJF Doctor
Specialité: Mathematiques et Informatiques Appliquées

Presentée et Soutenue Publiquement
par

Scott Cotton
le 25 juin 2009

Sur Quelques Problèmes de la Satisfiabilité

Preparé au laboratoire **Verimag**
sous la direction de
Oded Maler

Jury

Yassine Lakhnech	Président
Nikolaj Bjørner	Rapporteur
Sharad Malik	Rapporteur
Kenneth McMillan	Rapporteur
Armin Biere	Examinateur
Eugene Asarin	Examinateur
Andreas Podelski	Examinateur
Oded Maler	Directeur

Juin 2009

UNIVERSITY JOSEPH FOURIER - GRENOBLE I

THESIS

To obtain the grade of
UJF Doctor
Specialty: Mathematics and Computer Science

Presented And Defended in Public
by

Scott Cotton
on June 25th, 2009

On Some Problems in Satisfiability Solving

Prepared in the **Verimag** laboratory
under the supervision of
Oded Maler

Jury

Yassine Lakhnech	President
Nikolaj Bjørner	Reviewer
Sharad Malik	Reviewer
Kenneth McMillan	Reviewer
Armin Biere	Examinator
Eugene Asarin	Examinator
Andreas Podelski	Examinator
Oded Maler	Director

June 2009

Résumé

Le problème de la satisfiabilité est de déterminer si une formule donnée a une solution. Le problème de la satisfiabilité propositionnelle (SAT), où toutes les variables sont Booléennes, est un problème bien étudié avec plusiers d'améliorations en efficacité. Durant ces dernières années, le domaine de théories de satisfiabilitè modulo (SMT) a étendu les méthodes efficaces de SAT aux formules du premier ordre avec des variables non Booléennes, qui sont définies dans une théorie, par example un fragment décidable de l'arithmétique. Cette thèse explore des méthodes pour résoudre les problèmes de SAT et de SMT, en mettant l'accent sur ceux derniers.

Après un aperçu de la satisfiabilité propositionnelle, la thèse présente les résultats liés à la minimisation des clauses, les heuristiques, et les stratégies de redémarrage. Ensuite, la thèse donne un aperçu des méthodes SMT abstraites, qui sont relativement indépendentes de la théorie dans laquelle les variables sont définies. La thèse propose deux approches abstraites. La première est une variante d'une approche de SMT largement utilisée, appelée DPLL(T) [GHN$^+$04]. Cette approche favorise la souplesse et la clarité de l'interface entre un solveur de théorie et un solveur SAT. La seconde contient une classe d'algorithmes basées sur une généralisation de DPLL (GDPLL [MKS09]), qui cherchent un modèle directement dans l'espace de valuations des variables. Pour cette recherche directe, la thèse présente aussi des preuves de la correction, avec une notion du progrès et des conditions pour la terminaison.

La thèse étudie ensuite des instanciations de chacune de ces approches générales. L'approche basée sur DPLL(T) est instanciée avec un solveur efficace pour la logique de différences, ce qui donne lieu à un algorithme rapide pour la propagation de théorie. Cette méthode a été adaptée pour de diverses solveurs de SMT. La méthode de la recherche directe est instanciée pour l'arithmétique linéaire réelle. Pour ceci, divers mécanismes et propriétés utilisés dans les solveurs SAT ont été adaptées. Cette instanciation a permis de révéler certaines limitations de la méthode. Les résultats expérimentaux ont montré une variation de l'efficacité très différente de celle des méthodes traditionnelles.

Abstract

Satisfiability solving is the problem of determining whether a given formula has a solution. The most ubiquitous and well-studied satisfiability problem is propositional satisfiability (SAT), in which all variables are Boolean. In recent years, the field of satisfiability modulo theories (SMT) has extended methods in SAT solving to accommodate existential first order formulas with non-Boolean variables. Here the non-Boolean variables are related by a background theory, such as a decidable fragment of arithmetic. This thesis explores SAT solving and SMT solving, with an emphasis on the latter.

Beginning with an overview of propositional satisfiability, this thesis presents results related to clause minimization, heuristics, and restarts. The thesis then gives a general overview of abstract SMT methods, which are relatively independent of a given theory. Two approaches are proposed. The first is a variant of a widely used SMT framework, DPLL(T) [GHN$^+$04], which promotes flexibility and clarity in the interface between a theory solver and a propositional solver. The second is a class of generalized DPLL (GDPLL[MKS09]) algorithms which search directly for a model over the space of variable valuations. In this direct search case, the thesis presents proofs of correctness, with accompanying notions of progress and conditions for termination.

The thesis then studies instantiations of each of these proposals. The DPLL(T) based method is instantiated with an efficient solver for difference logic, giving a fast algorithm for theory propagation. This method has been adapted to various state-of-the-art SMT solvers. The direct search method is instantiated for real linear arithmetic. In this instantiation, various mechanisms used in modern DPLL solving are adapted to the generalized case, and some limitations of the method are recognized. Initial experimentation shows a very different performance profile than is found in traditional methods.

Acknowledgements

This thesis could not have been undertaken without the help and support of many. I would like to acknowledge my wife for her enduring support, and my daughters for their patience during the writing of this thesis. My advisor, Oded Maler, sought and found extraordinary opportunities for me to pursue this study, and was always encouraging of the pursuit of understanding. Verimag has provided a secure, friendly, and supportive environment. In preparation for this journey, Andreas Podelski offered his advisorship at IMPRS; Scott Weinstein provided at once outstanding and kind mentorship, without which I never would have undertaken this work; and my colleagues at University of Pennsylvania engaged me in many a stimulating discussion. Finally, my parents gave critical and unending love and support in undertaking this work.

Contents

Chapter 1

Introduction

The problem of satisfiability is the problem of finding whether or not there exists a solution of a given formula. Different kinds of formulas admit very different methods for determining whether there exists a solution, and there are of course very many kinds of formulas. The field of satisfiability solving also addresses a broad and diverse set of applications, touching on problems from verification to scheduling and optimization. Such applications often induce another, somewhat orthogonal, classification of problems related to the properties of the problem at hand. Overall, the range of issues related to satisfiability solving may seem over-encompassing.

At the same time, amongst this wide array of activity some principles and some problems have become ubiquitous. Over time some solution methods have become effective in a wide array contexts. On the other hand, some issues arise time and again and yet seem to always be addressed by *ad hoc* solutions. One ubiquitous problem is the problem of propositional satisfiability. Fortunately, solution methods for this problem are often effective and moreover tend to be structured around a single core methodology.

A harder ubiquitous problem is the problem of solving Boolean combinations of simple arithmetic constraints. For this problem, many *ad hoc* solutions involving the composition of a propositional solver with some conjunctive arithmetic solvers have been proposed [WW99, CAB+02, MR02, ACG+04]. These works fueled interest in the interface between propositional solvers and conjunctive solvers, leading to the DPLL(T) framework [GHN+04] and corresponding proof system [NOT05]. Later, a linear arithmetic solver was incorporated into the DPLL(T) framework [DdM06], yielding excellent results for the Yices solver and setting a standard for this class of problems.

The DPLL(T) framework and proof system demonstrated that it may be worthwhile in specific cases to consider the full problem of satisfiability solving in very abstract terms. Some literature has followed this line of thought [dMB08, BDdM08, MKS09] but the fundamental questions are difficult and the newer proposals have not yet been demonstrated in the form of a competitive general purpose solver.

The primary contributions of this thesis are as follows. We propose two abstract frameworks for satisfiability solving, and in each case examine concrete instantiations for solving Boolean combinations of simple arithmetic constraints. Our first proposal is a variant of DPLL(T) which promotes flexibility and clarity in the interface between a propositional solver and a theory solver. We instantiate this framework for simple difference constraints and demonstrate improvement on previous work. Second, we present a family of direct model search algorithms in the spirit of GDPLL [MKS09] based on variable-local consistency. We then examine a concrete instantiation for continuous linear arithmetic, and show how various aspects of propositional solving can be generalized to the case of linear arithmetic.

The rest of this thesis is organized as follows. Chapter 2 presents the topic of propositional satisfiability solving, and includes major developments from the last few decades. This chapter forms an important foundation of all the subsequent chapters. Chapter 3 gives an overview of abstract frameworks for satisfiability solving, and places our proposals in the context of existing work. In Chapter 4 we present a theory solver for simple difference constraints which implements the ideas we proposed for flexible propagation in the DPLL(T) framework. We also give efficient algorithms for the problem of theory propagation of difference constraints in the DPLL(T) framework. This work has provided a basis for how some competitive SMT solvers treat difference constraints. Chapter 5 examines how various mechanisms in propositional solving generalize to linear arithmetic in a direct model search framework. We conclude in Chapter 6.

Chapter 2

Boolean Satisfiability

Not only is the problem of propositional satisfiability central to computer science in general due to its relation to complexity theory, but it is also central the rest of the problems we consider in this thesis. In this chapter, we review propositional logic and its decision procedures.

2.1 Propositional Logic

Definition 2.1.1. *Propositional Formula*
The set of propositional formulas over a set of variables X is defined recursively as the smallest set containing all the following

1. The constants $0, 1$.
2. The variable x if $x \in X$.
3. $\neg\phi$ if ϕ is a propositional formula.
4. $\phi \wedge \psi$ if ϕ and ψ are propositional formulas.

A variable-free formula can be mapped to $\{0, 1\}$, that is evaluated as true or false, based on the truth tables of the two operators $\wedge$ and $\neg$:

$\wedge$	1	0
1	1	0
0	0	0

$\neg$	1	0
	0	1

Applying the truth tables recursively from the leaves of a formula to its root yields a value in $\{0, 1\}$ taken to be the *truth value* of the formula.

One can readily express other connectives such as $a \vee b \equiv \neg(\neg a \wedge \neg b)$ and $a \rightarrow b \equiv \neg a \vee b$.

Definition 2.1.2. *Satisfiability* A propositional formula ϕ over a set of variables X is said to be *satisfiable* if there exists a valuation $\alpha : X \rightarrow \{0, 1\}$ to its variables such that substituting $\alpha(x)$ for every x occurring in ϕ yields a formula with truth value 1.

A decision procedure for propositional logic is an algorithm which determines whether or not a formula is satisfiable. A restricted form of this problem was the first problem shown to be NP-complete [Coo71] and the result readily generalizes to general formulas as stated here.

2.2 Satisfiability Solvers

Modern satisfiability solvers are general purpose implementations of decision procedures which focus on efficient data structures, heuristics, and more or less anything which might make them solve problems faster. The efficiency of these solvers is not well described by the "intractable" label often associated with NP hard problems. For instance, when applied to satisfiability, the difficulty of NP hard problems is measured in terms of formula size, *i.e.* the number of variables and clauses in a given problem. However, current off-the-shelf solvers are able to decide non-trivial formulas with hundreds of thousands (even in some cases millions) of variables and at the same time unable to solve some problems with as few as 286 variables [SAJ$^+$08]. In general, while much certainly remains out of reach of current solvers, the problems arising from specific application domains are often easily resolvable by these solvers. Indeed this behavior is compatible with the fact that the complexity class is only a worst-case measure; it may be a bit premature to dismiss the potential of satisfiability solvers just because the general problem they attempt to solve is NP-complete.

2.2.1 CNF

Most Boolean satisfiability solvers work on formulas in conjunctive normal form, which we describe presently. A *literal* is a variable or its negation. A *clause* is a disjunction of literals. A formula in conjunctive normal form is a conjunction of clauses.

Any formula can be translated to CNF using standard Boolean equivalences. For example, one could take any formula ϕ using $\wedge$-, $\vee$- and $\neg$-operators, then translate it to CNF as follows. First, take the negation

$\neg\phi$. Second, push the negations to the leaves using De Morgan's rules $\neg(a \wedge b) \iff \neg a \vee \neg b$ and $\neg(a \vee b) \iff \neg a \wedge \neg b$. Third, apply distributivity, mapping sub-formulas in the form $a \wedge (b \vee c)$ to $(a \wedge b) \vee (a \wedge c)$. Now, after eliminating double negations, the formula is in the form $\bigvee_i \bigwedge_j l_{ij}$ with each l_{ij} a literal. Finally, taking the negation of this formula, applying De Morgans rules and eliminating double negatives again one arrives at a formula equivalent to ϕ in the form $\bigwedge_i \bigvee_j m_{ij}$. Observe that each $m_{ij} \equiv \neg l_{ij}$ is a literal and each $\bigvee_j m_{ij}$ is a clause.

The above process, and in fact any translation using only valid equivalences are problematic because the size of the formula can grow exponentially, as is readily verified by an attempt to translate an example of the form

$$\bigvee_{i \in I} x_i \wedge y_i$$

which grows exponentially with the size of the index set I.

However, if we allow the introduction of new variables we can arrive in linear time at an *equisatisfiable* formula. Based on the translation of Tseitin [Tse68], this is often accomplished as follows. Let s be a sub-formula of a given formula ϕ which is positive, *i.e.* not in the form $\neg\psi$. Associate with s a fresh variable z_s. Without loss, we can assume each such sub-formula is either a literal or the form $a \wedge b$ where a and b are positive sub-formulas or their negations. The formula

$$z_s \leftrightarrow (a \wedge b)$$

may be expressed as the following conjunction.

$$(\neg z_s \vee a) \wedge (\neg z_s \vee b) \wedge (z_s \vee \neg a \vee \neg b)$$

Using the above translation as a basic operation, we can translate an arbitrary non-cnf formula ϕ to an equisatisfiable CNF formula ϕ' as follows. First we assume the formula is represented in the signature $\wedge, \neg$. Let $a \wedge b$ be a smallest positive sub-formula of ϕ using the $\wedge$-operator. Observe that a, b are literals. Now create a new variable z and add the clauses representing $z \leftrightarrow a \wedge b$ to the translation ϕ'. If we then replace ϕ with $\phi[a \wedge b \mapsto z]$, we can continue this process eliminating every $\wedge$-operator in ϕ. Suppose that the root level operator in ϕ is a $\wedge$-operator. let z_r be the variable introduced to represent this sub-formula. We then add the conjunct z_r to the translation and we are done. Otherwise, we assume without loss that the root level operator is a negation, in which case we add the conjunct $\neg z_r$ to the translation and we are done. Each sub-formula from the original formula ϕ

is translated once, and each translation may be performed in constant time hence the procedure is linear.

In practice, while equivalence based translations may become too large, there is also penalty associated with introducing new variables. Hence more sophisticated translations which attempt to balance the two methods are also in use [EB05, EMS07].

2.2.2 Resolution

The notion of resolution [Rob65] forms a cornerstone of most modern satisfiability solvers. In the context of CNF, resolution is a simple proof rule allowing one to deduce clauses from an existing set of clauses. In particular, resolution is the following proof rule

$$\frac{x \vee C, \neg x \vee D}{C \vee D}$$

Observe that if $x \vee C$ and $\neg x \vee D$ are clauses, then $C \vee D$ is also a clause. In an application of the resolution rule, the variable x is called the *pivot* and the clause $C \vee D$ is called the *resolvent*.

For CNF formulas, resolution is refutation complete. Namely, if a CNF formula is unsatisfiable, then one can derive "false" in the form of an empty clause by resolution alone. Resolution has been used extensively in automated theorem provers, but has also often suffered from exponential space requirements as the number of clauses which can be derived grows as a function of the number of clauses.

Even without considering the number of clauses stored in memory during a proof, resolution proofs may require an exponential number of *steps*, as is well documented for example with problems coding the pigeon hole principle [Hak85]. Nonetheless, the simplicity of the proof rule and uniformity of representation by clauses makes resolution the method of choice for many proof systems, including satisfiability solvers.

One may view a resolution proof of a clause as a directed acyclic graph. In particular, given a proof π, define the graph (V_π, E_π) where V_π is the set of vertices, each consisting of a clause and where E_π is the edge relation denoting resolution steps as follows. For each resolution step, two clauses $x \vee C$, $\neg x \vee D$ are used to derive a new clause $C \vee D$, and this step is represented by the edges $(x \vee C, C \vee D)$ and $(\neg x \vee D, C \vee D)$. An example graph is depicted in Figure 2.1.

Viewed this way, one may define several sub-classes of resolution corresponding to topological constraints on proof graphs. Such requirements

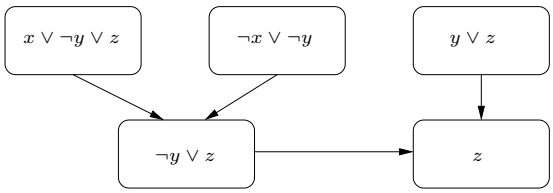


Figure 2.1: An example graphical representation of a resolution proof.

may impose further limitations on proof efficiency, measured in terms of the smallest graph capable of proving some fact. One common sub-class of resolution is *tree-like resolution*, in which derived clauses may appear more than once as nodes in the proof graph, but all *nodes* representing derived clauses play the role of antecedents in at most one application of the resolution rule. Another such class is *regular resolution*, in which no variable acts as a pivot more than once in any path in the proof graph. The class of proofs without topologic constraints on the proof graph is referred to as *general resolution*. It is well known that the smallest tree-like resolution refutations and the smallest regular resolution refutations can be exponentially larger than the smallest general resolution proof [BEGJ98, AJPU07].

2.2.3 Davis-Putnam-Loveland-Logemann

The Davis-Putnam-Loveland-Logemann [DP60, DLL62] algorithm forms the most fundamental basis for most modern SAT solvers. The main idea is one of depth first search over the space of variable valuations. The search is augmented with a restricted form of resolution called *unit propagation*, in which clauses with only one literal, referred to as *unit clauses*, are automatically added to the stack of assigned variables in the depth-first search. The original procedure also made use of *pure literal detection*, in which variables which never occur negated or never occur without negations are eliminated. In modern solvers, pure literal detection is generally not implemented because it has been found unworthwhile in modern implementations of the procedure. Pseudocode characterizing the algorithm, without pure literal detection, is presented in Algorithm 2.2.1.

Algorithm 2.2.1. *Davis-Putnam-Loveland-Logemann Algorithm for Boolean Satisfiability*

```
DPLL(φ)
    if ∃ an empty clause in φ return 0
    if ∃ a unit clause m in φ then
        if m ≡ x return DPLL(φ[x ↦ 1])
        else // m ≡ ¬x
            return DPLL(φ[x ↦ 0])
    if ∃ a free variable in φ then
        let x be a free variable in φ
        return DPLL(φ[x ↦ 0]) or DPLL(φ[x ↦ 1])
    else return 1
```

DPLL and Resolution

The DPLL algorithm above may be viewed as a restricted form of resolution. Algorithm 2.2.2 presents variant of the DPLL procedure, based on similar work by [GNT06] which explicitly returns an empty clause derived by resolution in the event the formula ϕ is unsatisfiable.

Algorithm 2.2.2. *DPLL Algorithm with resolution steps and non-chronological backtracking*

```
DPLL-BJ(φ, α)
    if ∃ a clause c in φ s.t. c[α] = 0 return (0, c)
    if ∃ a clause c in φ s.t. c[α] = x then
        let (a, w) = DPLL-BJ(φ, α ∪ {x ↦ 1})
        if a = 1 or x not in w return (a, w)
        return (0, resolve(w, c))
    if ∃ a clause c in φ s.t. c[α] = ¬x then
        let (a, w) = DPLL-BJ(φ, α ∪ {x ↦ 0})
        if a = 1 or x not in w return (a, w)
        return (0, resolve(w, c))
    if ∃ a variable x unassigned in α then
        let (a, w) =DPLL-BJ(φ, α ∪ {x ↦ 0})
        if a = 1 or x not in w return (a, w)
        let (a′, w′) =DPLL-BJ(φ, α ∪ {x ↦ 1})
        if a′ = 1 or x not in w′ return (a′, w′)
        return (0, resolve(w, w′))
    else return (1, ⊤)
```

In the above procedure, we keep track of the assignment α separately from the formula ϕ so as to enable reference to clauses found in ϕ. The procedure returns $(1, \top)$ if ϕ is satisfiable and $(0, \bot)$, where $\bot$ is the empty

clause derived by resolution, if ϕ is not satisfiable. The function `resolve()` simply returns the (unique) resolvent of its two arguments. The procedure "backjumps" over variable branch points which are not involved in dead-ends. While we do not give a formal proof here, we claim that the procedure is correct, that all the derived clauses always contain only assigned variables, and that they are always false under the assignment. We note also that the resolution in the procedure DPLL-BJ is both regular and treelike. For further analysis of this procedure and a proof of a related facts[1], see [GNT06] and [GNT01].

2.3 Modern Solvers

Although modern solvers employ many different techniques, the majority of them are strongly based on the DPLL depth first algorithm with unit propagation. Nonetheless, under the hood modern solvers are vastly more complicated than would be indicated by the simple recursive DPLL procedures above. In this section we outline some of the more important algorithmic and architectural components of modern satisfiability solvers.

2.3.1 Basic Structures

Modern DPLL based solvers are often centered around the architecture and data structures described in [ES03]. Modern DPLL based solvers use a sparse representation of clauses wherein a clause is just a list of the literals it contains. In turn, literals are often represented as integers as follows. If there are n variables in a given problem, then each variable x is represented by a value in $[0 \ldots n)$, and each literal m over a variable x is represented as follows. If m is not negated, then m is represented by $2x$, otherwise m is represented by $2x + 1$.

Since the solvers are generally not written recursively, there is an explicit representation of the call stack in the form of an array of literals. A distinction is made between assigned variables and unassigned variables, where all assigned variables are placed on the stack in the order of assignment. Assignments in turn are processed one at a time by scanning those clauses which contain the negation of the assignment in a search for unit clauses. Whenever a unit clause is found implying an unassigned literal m, m is added to the top of the stack. Generally assignments are processed in the

[1] While the citations refer to work addressing arbitrarily quantified formulas, the reasoning is readily applied to the existential case of DPLL.

order in which they are added to the stack. This creates a situation in which the stack also operates as an assignment processing queue. This structure can be represented as an array a with two pointers h, t in which $a[0 \dots h)$ contains assignments which have been processed and $a[h \dots t)$ contains assignments which have not been processed. Additionally, t acts a pointer to the top of the stack used for `push` and `pop` operations. When $h = t$, there are no unit clauses and the process branches on a choice.

An important part of modern solvers involves analyzing the dependencies of assignments. In particular, if a literal m is assigned because a unit clause $C \vee m$ is found, then m *depends on* the negation of the literals in C, all of which are assigned. The clause $C \vee m$ and sometimes also the negation of the literals in C are called *antecedents* of m. With clauses represented as lists, this dependency relation can be succinctly represented by associating a pointer to the clause $C \vee m$ with the literal m when m is assigned. This creates an adjacency list representation of the dependency graph and makes it easy to implement some graph algorithms on the dependency graph. The resulting dependency graph is referred to as the *implication graph* in the literature.

2.3.2 Learning

Beginning with [SS96], satisfiability solvers have employed a process known as "learning", in which new clauses are derived by resolution from the original problem and recorded for future use. Such clauses are derived in such a way that they increase the reach of unit propagation. Learning has a significant impact on practical performance, and also theoretical impact on the worst case complexity of the DPLL procedure. On the theoretic front, learning removes the restriction that resolution be tree-like or regular. In fact, it has been shown that DPLL with learning can simulate general resolution [HBPG08]. As a consequence, it is possible for a solver which learns to refute some problems exponentially faster than the best possible run of a solver which does not employ learning. On the practical side, it has been our experience that a solver which makes a good choice of which clauses to learn and which is able to limit the total number of learned clauses kept in memory effectively almost always greatly outperforms any solver without learning.

One of the well-researched [BKS04, ABH+08, DHN07, ZMMM01] problems concerning learning is exactly what clauses to learn. Currently almost all DPLL based solvers learn a clause most often described as a first unique implication point (1UIP) [MSS99] and much literature supports the notion

that a 1UIP clause is indeed a good choice. Referring to the DPLL-BJ procedure above, we can define the 1UIP clause derived from a conflict as follows. Let the *choice depth* of an assigned variable x denote the number of times the procedures has selected a free variable in the call stack at the time x is assigned. Now the 1UIP clause is the first clause, in the order of derivation by resolution, which contains exactly one variable with maximal choice depth. On any conflict occurring with positive choice depth, such a clause is guaranteed to exist because eventually the resolution steps will lead to a clause include the last chosen variable.

With a 1UIP clause in hand, the procedure may backtrack to a lesser choice depth than otherwise, and will never backtrack to a greater choice depth. This is a result of the fact that derivation steps can add variables with a new choice depth, but can never remove all variables of a given choice depth once they are present in a derived clause. With the 1UIP clause, the procedure may backtrack to the second greatest choice depth and then make a new assignment by unit propagation justified by the 1UIP clause. An interesting side effect of making this assignment is that the DPLL procedure no longer has any reason to explicitly test both truth values of an unforced variable. Rather, the procedure can assign an arbitrary truth value to unforced variables and simply backtrack over them. Such an altered procedure still terminates because it always induces strictly more unit propagation on the initial segment of the assignment [ZM03].

Other types of learned clauses also have the property that there is only one literal of maximal choice depth and they are referred to as *asserting* clauses in the literature. It is worth noting that control flow of a solver which systematically learns asserting clauses is quite different than the control flow indicated in the procedure DPLL-BJ above. In general, one finds algorithms with structure similar that indicated in the pseudocode below.

Algorithm 2.3.1. *Control Flow of a Modern DPLL Solver*

```
loop:
   if propagate() = 0
      if choice-depth = 0 then
         return 0
      derive-asserting-clause()
      backtrack()
   else if ∃ an unassigned variable x
      assign(x)
   else
      return 1
```

The function `propagate()` assigns all unit implied variables and returns 0 if there is a *conflict* or empty clause. The function `derive-asserting-clause()` derives and records any asserting clause. The function `backtrack()` backtracks to the 2nd deepest choice depth of a variable in the derived clause. The function `assign()` may pick an arbitrary truth value for the variable x.

In summary, the incorporation of learning into the DPLL algorithm is a fundamental development in satisfiability solving which significantly enhances practical performance and theoretically can refute problems more quickly by virtue of removing some of the restrictions on the underlying resolution.

2.3.3 Efficient Unit Propagation

Modern sat solvers tend to spend the majority of their time performing unit propagation, and have developed efficient data structures for accomplishing this task. The idea of *two literal watching* (2LW) was first introduced in [MMZ$^+$01], improving on the use of *head-tail pointers* [MSS96, ZS96]. Minor variations of 2LW are present present in most modern solvers. The idea is to limit the number of clauses that the solver must check for unit propagation as a result of an assignment. With two literal watching, two special "watched" literals are associated with each clause and an assignment of a literal m is associated with a scan of the clauses which contain the watched literal $\neg m$. By examining the other watched literal of such a clause, one may determine if the clause is solved. If the clause is not solved, a scan looks for a new watched literal, which must be either unassigned or true. If none is found, the clause is either unit or a conflict clause, depending on the value of the other watched literal. An important feature of two literal watching which sets it apart from head-tail pointers is that watched literals don't need to be updated as a result of backtracking.

In [JC07, SLKB07], mechanisms are proposed which allow scanning the *watched* literals of clauses without loading the list of literals for a given clause into memory in the event that the "other" watched literal is true. BarceLogic Tools [Oli08] furthers this idea by eliminating a degree of indirection in the scanning mechanism. MiniSAT [Sör08] moreover removes the need for the "other" blocking literal to be kept in the watchlist by associating it with each clause, thus removing the need to update associated watchlist information.

2.3.4 Variable Heuristics

A variable ordering heuristic is a mechanism by which variables are chosen when the DPLL procedure branches. Different variable orderings can make a big difference in solving time, or even solvability. Variable heuristics may be categorized as static or dynamic. Static variable orders are determined before the search process and tend to have very little computational overhead in the search process. However, static variable ordering are not in general as effective as dynamic variable orderings which glean information from the search process to determine better variable ordering based on the current state of the search process [Sil99]. Dynamic variable ordering ranges from simple literal counting schemes to expensive mechanisms such as unit propagation lookahead in which all variables are tried and one which propagates the most literals is selected. While unit propagation lookahead has proven effective on some hard problems it also is quite slow compared to more lightweight mechanisms [GMT03, Ber01].

Most modern solvers employ the MiniSAT implementation [ES03] of the idea of *variable state independent decay* (VSID) heuristics [MMZ+01]. The basic idea is to give variables a score based on the number and recency of conflicts in which the variable's assignment implied the conflict. Following [Bie08a], we may view the score of a variable in normalized form. Given some parameter $p \in (0 \ldots 1)$, which typically takes a value of about 0.95, the score $s_{v,k}$ of a variable v after the kth conflict may be expressed as

$$s_{v,k} \triangleq \begin{cases} p \cdot s_{v,k-1} + (1-p) & \text{if } v \text{ helps to imply the conflict} \\ p \cdot s_{v,k-1} & \text{otherwise} \end{cases}$$

In practice, this value is approximated in a way that allows the solver to update only the scores of those variables which are antecedents of the conflict. In [ES03], this is accomplished by incrementing the score of such "involved" variables by a quantity inc_k where $inc_{k+1} \approx \frac{inc_k}{p}$, and this mechanism is now commonly found in solvers. Also in practice, not every variable which contributes to a conflict has its score incremented. Often, only those which act as pivots in the resolution used to derive a learned clause are incremented.

2.3.5 Clause Minimization

Effective minimization of learned clauses was an unrealized objective until MiniSAT [ES05] which employs the notion of "self-subsumption" to minimize clauses. In particular, a clause c subsumes a clause d if all the literals in c are also found in d. Learned clauses are never subsumed by existing

clauses, however the idea can be extended to include a resolution step and be used for minimization. In particular, given a resolution step

$$\frac{x \vee C, \neg x \vee D}{C \vee D}$$

The resolvent $C \vee D$ may subsume an antecedent to the rule. If a learned clause plays the role of the antecedent, then a smaller learned clause will have been found. If the resolvent does not subsume the learned antecedent, resolution can be applied again, using any literal m found in the resolvent but not in the learned clause as the pivot. In [ES05], these resolution steps are taken with clauses responsible for the unit propagation of the value $\neg m$.

This mechanism is often implemented using a series of depth first resolution steps, one based on each literal in the original clause to see if it could be eliminated from the learned clause, yielding a theoretically quadratic worst case minimization algorithm. In [SB09], experiments demonstrate that clause minimization can be an effective tool for pruning the search space, and that recursive application of resolution steps generally performs better than restricting the resolution steps to literals appearing in a learned clause.

2.3.6 Solution Caching and Restarts

For many years satisfiability solvers have been employing the practice of restarting, wherein a solver simply backtracks to the root choice depth and then continues. Generally, while this practice makes some problems more difficult and can even render the method incomplete, it also tends to somehow guide the solver to a result in a substantial number of benchmark problems, when otherwise no result would be found. The mechanism behind this tendency is not well understood but has been justified intuitively as a countermeasure to the tendency of depth first search to become stuck in a bad portion of the search space [Bie08a, PD07]. That there can be bad portions of the search space corresponds to the idea that solvers tend to have high runtime variation across the parameter space, even when the parameters aren't terribly meaningful, for example as with a random seed [GS09].

One of the limitations to the effectiveness of restarting is the need for a solver to repeatedly find solutions to hard subproblems of a given problem. The solver RSAT [PD07] found a way to allay this problem with very little overhead. In particular, whenever a variable is assigned by choice, it is simply given the same truth value it had the last time it was assigned, while first-time assignments may be chosen by any means. Although the practice

of restarting still forms an impediment for some problems, this mechanism and variations are now prevalent amongst recent satisfiability solvers.

Currently, solvers employ a variety of restart strategies, in which the frequency of restarts is varied according to some scheme. The two most often employed strategies make use of the Luby series [M. 93], first employed in RSAT, and a two-level geometric series proposed in [Bie08c] and first employed in picosat. Both strategies have the property that on average restarts become less frequent over time while locally the restart frequency oscillates.

2.3.7 Preprocessing

Most modern satisfiability solvers employ some form of preprocessing which complements the strengths of the DPLL framework. A common preprocessor is SatELite [EB05], which attempts to minimize the CNF formula in a way particularly geared towards CNF formulas created from circuits by Tseitin's translation. Other methods employed include failed literal detection [Bie08b], in which literals are assigned at the root level and if they lead to a conflict their negations are added to the root level assignment.

2.4 Non-standard satisfiability solving

For completeness, we mention some of the more prevalent satisfiability solving techniques which are not based on DPLL or resolution.

2.4.1 Local Search

Some local search mechanisms, such as WalkSAT [SKC93] are effective mechanisms for finding satisfying assignments. Interestingly, local search can far outperform DPLL based algorithms for hard, satisfiable, uniformly generated random problems. However, not only do these processes fail when a problem is unsatisfiable, it is also known that local search is quite ineffective on problems with lots of dependent variables, a side-effect of functional structure. Although many methods have been proposed to overcome this problem [KMS97, Seb94], such adaptations have not kept up with DPLL-based developments.

The WalkSAT algorithm is parameterized by a probability p, which typically takes a value of about $\frac{1}{2}$. It keeps a valuation of the variables $\alpha : X \to \{0, 1\}$ and it keeps track of all clauses which are false or unit under α. Its basic operation is to choose, with probability p a random variable

which appears in a false clause and then flip its value in α. With probability $1 - p$ a variable which, if flipped, will solve the most clauses is chosen (with negative counts for the clauses that will be falsified as a result included). This process repeats for as long as desired, possibly restarting periodically, until a satisfying assignment is found.

2.4.2 Stålmark's Method

Stålmarks method [SS90] is a breadth-first search mechanism which can be very effective at refuting problems. The algorithm makes use of simple propagation rules similar to unit propagation while maintaining equivalence classes over the variables. In addition, the dilemma proof rule:

$$\frac{p \vdash a, \neg p \vdash a}{a}$$

is used as the basis for branching. Here the predicates p and a take the form of an equivalence assertion $x \leftrightarrow y$ where x, y are variables or constants, and the $\vdash$ relation indicates propagation. Once all possible combinations (modulo the equivalence classes) yield no new conclusions, the dilemma rule is applied recursively under the branches p and $\neg p$. In general, the recursion depth is only incremented once all possible applications of the dilemma rule have been exhausted.

The method refutes a formula when it derives a fact and its negation. While the method can arrive at a satisfying assignment in the event that the formula is satisfiable, it is generally not used in this manner because the recursion depth can be high and exhausting the dilemma rule at any given level is rather expensive. On the other hand, since the method exhausts the dilemma rule prior to engaging in deeper recursion, any resulting proofs of unsatisfiability are guaranteed to involve only a minimal amount of branching; this can lead to short refutations.

2.5 Experiences

This section provides informal summaries of a few ideas we explored relating to the standard mechanisms employed in modern DPLL solvers.

Linear Time Clause Minimization

The minimization of learned clauses in [ES05] is often implemented with a quadratic algorithm. In our experience, some large problems would induce

behavior approaching the quadratic worst bounds. It is possible to accomplish the minimization in linear time, resulting in a more robust solver in the sense that no problems would cause the solver to get bogged down in minimization, while all learned clauses would be fully minimized. The algorithm is described below in pseudocode.

Algorithm 2.5.1. *Linear time learned clause minimization.*

```
minimize(C)
    N ← ∅
    R ← ∅
    for each literal m ∈ C
        if isRedundant(¬m, C) then
            C ← C \ {m}
    return C

isRedundant(m, C)
    if m ∈ N return 0
    if m ∈ R return 1
    if m is unit-implied then
        let a be the clause which implied m
        for every n ∈ a, n ≠ m
            if ¬isRedundant(¬n, C) ∧ n ∉ C then
                N ← N ∪ {m}
                return 0
        R ← R ∪ {m}
        return 1
    else
        N ← N ∪ {m}
        return 0
```

The algorithm is very similar to the MiniSAT minimization code referred to in [ES05]. However, there and in other solvers, there is no explicit distinction made between the sets C, R, and N. Rather, a single set of "seen" literals is kept, initialized to C, and `isRedundant()` is applied to not-yet-seen literals which appear in antecedents of C. Without storing the distinction between redundant and irredundant literals, the depth first `isRedundant()` procedure may have to re-initialize the set "seen" before every root level call to `isRedundant()`, because it cannot be known whether a "seen" literal is redundant. It is possible to re-initialize only in the case that the literal is not redundant, but the procedure is still quadratic in the worst case.

By contrast, the procedure above classifies all reachable literals in the implication graph as either redundant (R) or not redundant (N). A literal m is not redundant if there is a path in the implication graph which leads to a guessed value (not forced by unit propagation) without passing through the negation of some literal in the initial clause C. All other literals have the property that they are consequences, by means of unit propagation, of an assignment $\neg C'$ where C' is some sub-clause of C. The critical observations to make in order to see the correctness of the algorithm follow.

1. The initial clause C and all the literals therein are false, while all unit implied literals in the graph are true.

2. A literal is classified as being in R or N only after all literals implying it via a unit clause have been classified.

3. If there is no path from a literal m to a guessed literal without passing through the negation of some literal in the initial clause C, then m is implied by unit propagation under an assignment $\neg C'$ where C' is some sub-clause of the initial clause C.

4. If every antecedent of a literal m is classified as R or is in C, then there is no path from m to a guessed literal which does not pass through the negation of some literal in the initial clause C.

The original procedure, as result of the re-initializations, has roughly quadratic worst case behavior, or more precisely $\mathcal{O}(n \cdot |C|)$ where n is the number of literals contributing to the conflict. By contrast with the formulation above, every literal is visited at most once and the performance is linear.

In practice, the algorithm above is implemented using bits associated with each variable corresponding to membership in R or N, and also with a non-recursive implementation of `isRedundant()`. In addition, the procedure `isRedundant()` may terminate early with a negative result if it comes across a literal whose choice depth is different from any choice depth of any literal in the learned clause C, or a literal whose choice depth is 0. Our experience shows that these algorithmic and implementation level changes yielded a more robust minimization procedure. Prior to the changes, occasionally large problems would result in the solver spending most of its time minimizing learned clauses. With the changes suggested here, profiling indicated the solver never spent more than 15% of its time in minimization, and usually spent far less.

There are other options available to limit the time spent in clause minimization; and moreover a solver developer may prefer a quadratic algorithm if it has better average-case behavior. From personal communication with Armin Biere, we were informed that such a linear time implementation was slightly slower than the quadratic implementation in picosat. Our own experience showed that carefully implemented linear time minimization gave an overall improvement in robustness when compared with a recursive quadratic implementation. Independently, a linear time implementation is presented in [Gel09] and shown to improve proof traces with modest overall speedup when dropped into MiniSAT.

Restarts and VSIDs

An effective choice for the recency parameter p in VSID heuristics may be directly related to restart frequency. Observe that in the presence of RSAT style solution caching, restarts only change the order in which variables are chosen. If $p \approx 1$, then a restart will essentially do nothing, since the variable order will not change. Interestingly, we were able to solve a number of problems, both satisfiable and not, from the SAT Race 2008 with a solver that restarts at *every* decision immediately following a conflict under *very* low recency settings such as $p \approx \frac{1}{2}$. In our experiments, this configuration even significantly outperformed the more conservative traditional restart strategies on a variety of satisfiable and unsatisfiable instances, which we found surprising because the strategy makes no real effort to exhaust the search space. However, while this strategy could solve many problems faster, there were always at least equally many problems it solved much more slowly – or not at all, making the configuration unworthwhile overall.

This suggests the possibility that solvers may benefit from correlating restart frequencies with recency parameters. Our own attempts to do this with dynamic restart frequencies led to significant numerical problems associated with updating the recency parameter which we were unable to overcome. However, we did find that the correlation seems to hold when comparing different solvers of fixed restart frequency. Namely solvers that restarted less frequently benefitted from higher values for the recency parameter p, levelling off at about 0.95 for solvers which did not restart. However, no fixed restart strategy and recency parameter proved worthwhile in comparison to the traditional dynamic restart strategies with a fixed p.

Restarts and Solution Caching

It is possible to use a heavy weight solution caching mechanism with restarts. Consider a partial solution as an maximally consistent ordered sequence of literals corresponding to choice points together with the learned clauses necessary to induce unit propagation under the literals. Consider now a solver which restarts by tracing a prefix of the best cached solution and choosing a point at which to deviate on every choice point which immediately follows a conflict. We implemented this idea by associating a parameter p_d at each choice depth d. While tracing the prefix of the best solution, the solver may deviate with probability p_d, in which case a VSID choice is made. Otherwise the p_d decays by some global factor $f \in (0 \dots 1)$, the solver makes the best choice and moves to the next position of the best cached solution, if it exists. If it does not exist, a VSID choice is made and the best solution is guaranteed to improve. If a VSID choice is made and it strictly improves on the best cached solution at depth d, then p_d is reset to some initial value.

This induces a search which is biased towards bettering the best solution but also always exploring new space *at every choice depth*. Using this mechanism, we were able to solve in about half an hour a problem from SAT-Race 2008 that was not solved in the competition, namely the problem `aloul-chnl11-13` with 286 variables. The solvers picosat-846 and a 2007 version of MiniSAT were not able to solve the problem given 2 hours. By varying the restart frequency as a function of the depth of deviation, as well as with parameter tuning, we were able to cut the time down to under 5 minutes. This problem is a wire routing problem which is essentially a hidden pigeonhole problem, and other methods are able to solve it [Sab05, Chapter 6]. Nonetheless, we found this method overall unworthwhile on the entire SAT-Race 2008 problem set. The overhead of the solution caching mechanism is significant, and the method may have interfered with the effectiveness of VSID heuristics. At the same time, this suggests that restart strategies which are biased towards globally optimal measures may be exploited.

Another restart mechanism based on restarting at different choice depths with varying frequencies was proposed in [RS08] and shown to be effective for unsatisfiable problems when applied to a 2007 version of MiniSAT.

2.6 Conclusion

While Boolean satisfiability does not comprise the main topic of this thesis, it does play a central role in all of what follows. Enormous progress has

been in satisfiability solving in the last 15 years, and satisfiability solvers have become both indispensable engines for numerous applications.

Chapter 3

Satisfiability Modulo Theories

Satisfiability modulo theories (SMT) extends propositional satisfiability solving to formulas which include non-propositional variables. For example, an SMT solver may be capable of determining the satisfiability of a propositional combination of linear constraints $\Sigma_i a_i x_i \leq b$. In this case, the variables x_i are numeric, and the background *theory* may be the theory of real or integer linear arithmetic. In the SMT literature, theories are generally treated at two levels. On the abstract level, a decision procedure is developed which handles a propositional combination of theory atoms, making as few assumptions as possible about the theory to which the atoms happen to belong. Given such a parameterized decision procedure, a concrete instantiation will interpret the atoms in the background theory, fulfilling any interface requirements imposed by the parameterized procedure. This chapter addresses topics concerning the abstract level. As such our exposition omits many interesting *ad hoc* methods for specific theories which have not been generalized.

This chapter is organized as follows. In Section 3.1, we recall basic logical notions used in the rest of this chapter, define and discuss *intersecting* SMT frameworks, including DPLL(T), and present Nelson-Oppen combination of theories. Section 3.2 presents a simple refinement of DPLL(T) [GHN$^+$04] based on decoupling consistency checks and theory propagation. Section 3.3 presents a non-intersecting SMT method based on GDPLL [MKS09]. Section 3.4 concludes.

3.1 Traditional SMT Methods

3.1.1 Logical Background

Syntactically, a formula consists of a set of symbols, which we characterize as either variables, logical symbols ($\vee, \wedge, \neg, \forall, \exists, =$), or non-logical symbols such as constants (such as $0, 1$), relation symbols (such as $<$), or function symbols (such as $+, \cdot$). A *signature* is a set of non-logical symbols. A *structure* may be associated with a signature and consists of set referred to as the *domain* together with an *interpretation* for each non-logical symbol in the signature. The interpretation of a constant is just an element of the domain. The interpretation of an n-ary relation is a subset of D^n where D is the domain. Likewise, the interpretation of an n-ary function is a subset of $D^n \rightarrow D$.

A *model* of a variable free formula ϕ is a structure for some signature which includes all the non-logical symbols in ϕ and which makes ϕ true[1]. A variable-free formula is *satisfiable* if there exists a model for it; we say that the model *satisfies* the formula. If a formula contains variables, one may consider the variables as extra constant symbols, in which case the above definition of satisfiability readily applies. A model for a set of formulas S is a model which satisfies every $s \in S$.

In the following, an *atom* or *atomic predicate* is a formula whose root symbol is either a relation symbol or equality. In other words, an atom corresponds to constraint which contains no Boolean connectives, such as $2x < y$. A *literal* is either an atom or its negation. As in propositional logic, a *clause* is a disjunction of literals. A *theory* T is then a set of formulas closed under first order logical deduction. A formula ϕ is satisfiable modulo a theory T if there is a model which satisfies both ϕ and T.

3.1.2 Propositional Abstraction

The notion of the *propositional abstraction* of a formula ϕ is a fundamental tool in SMT solving. Propositional abstractions allow us to speak meaningfully of interpreted and uninterpreted theory atoms appearing in a formula. Interpreted theory atoms are syntactic objects (theory atoms) associated with an interpretation in an appropriate structure; whereas uninterpreted theory atoms are syntactic objects which may be given a truth value independent of any interpretation.

[1]For establishing whether a variable-free formula is true, we assume the classical semantics of first order logic

We now define this idea formally. Let P_ϕ be the set of theory atoms occurring in ϕ, and let v_p be a fresh propositional variable for every $p \in P_\phi$. The propositional abstraction of ϕ is the formula

$$prop(\phi) \triangleq \phi[p \mapsto v_p]$$

The key point of $prop(\phi)$ is the correspondence between its models and conjunctions of theory literals. In particular, let $\alpha \models prop(\phi)$ and consider the conjunction

$$\phi_\alpha \triangleq \bigwedge\{p \mid \alpha(v_p) = 1\} \wedge \bigwedge\{\neg p \mid \alpha(v_p) = 0\}$$

It should be clear that any model of ϕ_α can be extended to a model of ϕ. Moreover, as there are a finite number of models of $prop(\phi)$, it is possible to determine the satisfiability of ϕ by checking the satisfiability of ϕ_α for every α such that $\alpha \models prop(\phi)$.

Example 3.1.1 (Propositional Abstraction)**.** Consider the formula

$$\phi \triangleq (x + 2y \leq 3) \vee (x = y)$$

To construct $prop(\phi)$, we generate propositional variables corresponding the constraints in ϕ, namely $(x + 2y \leq 3)$ and $(x = y)$. Let p_1, p_2 respectively correspond to these two literals. Then

$$prop(\phi) \triangleq p_1 \vee p_2$$

Consider the truth assignment

$$\alpha \triangleq \{p_1 \mapsto 0, p_2 \mapsto 1\}$$

This assignment induces the conjunction

$$\phi_\alpha \triangleq (x + 2y > 3) \wedge x = y$$

Observe that $\alpha \models prop(\phi)$, since $\alpha(p_2) = 1$. Also observe that ϕ_α is satisfiable, taking for example $x = y = 2$. As a result, $x = y = 2$ is a satisfying assignment for the original formula ϕ.

Most established SMT methods may be classified as *intersecting* methods, which is based on the notion of propositional abstraction. Given a formula ϕ, let L_ϕ denote the set of theory literals whose atoms occur in

ϕ. An intersecting method intersects the set of satisfiable conjunctions of interpreted theory literals:

$$\{X \mid X \subseteq L_\phi, \bigwedge X \text{ is satisfiable } \}$$

with all the conjunctions of uninterpreted theory literals whose truth values satisfies $prop(\phi)$:

$$\{X \mid \exists \alpha \,.\, \alpha \models prop(\phi) \text{ and } X = \phi_\alpha\}$$

Most SMT methods are based entirely or largely on this principle of intersection. However, a variety of different implementations of intersecting methods have been studied.

3.1.3 Eager Encodings

Eager SMT solving consists of encoding a theory-atom-laden formula ϕ into an equisatisfiable propositional formula ϕ', and then running a SAT solver on ϕ'. Generally, ϕ' is generated in the form $prop(\phi) \wedge \phi_T$, where ϕ_T describes the set of feasible truth valuations of atoms appearing in ϕ. For example, an eager encoding of

$$\phi \triangleq (x \geq 2) \vee (x \leq 0)$$

may replace $(x \geq 2)$ with p, $(x \leq 0)$ with q, and generate a formula

$$\phi' \triangleq (p \vee q) \wedge \phi_T$$

where ϕ_T describes the set of theory feasible combinations of $x \geq 2$ and $x \leq 0$ in terms of p and q. In the example above, ϕ_T would take the form $\neg p \vee \neg q$.

More formally, let P be the set of theory atoms found in a formula $\phi(x, y)$, where x is a set of propositional variables and y is a set of non-propositional variables. We can translate ϕ to an equisatisfiable propositional formula $\phi' \triangleq prop(\phi) \wedge \phi_T$ so long as ϕ_T is such that

$$\alpha \models \phi_T \iff \exists y \,.\, \phi_\alpha$$

The biggest challenge in the eager SMT method is designing an efficient procedure for generating ϕ_T. Some theories admit such procedures. For example, 2's complement arithmetic over fixed-width integers can be readily coded using standard circuit descriptions of arithmetic operators. Eager encodings have also been proposed for a number of infinite domain theories

[SSB02, Str06, Str02]. However, the generation of a propositional formula from an arbitrary set of models can be computationally difficult and so some ingenuity is often required in the coding. Nonetheless, eager methods allow off-the-shelf use of propositional SAT solvers and independent treatment of propositional and theory-specific reasoning.

3.1.4 Lazy SMT

In the literature, the term "lazy SMT" generally refers to the use of a SAT solver as a *driver of* a theory solver. Namely, a SAT solver calls a theory solver while it is running. There are many different ways such a process can be implemented and indeed many ways have been examined. Unfortunately, these variations are often best characterized by their own spectrum of laziness, leading at times to confusion over the meaning of the term "lazy".

At the lazy end of the spectrum, a SAT solver is used as an enumerator of full truth assignments to the atoms found in a formula ϕ. The sat solver generates full truth assignments α such that $\alpha \models prop(\phi)$. Then ϕ is satisfiable if there is an α such that ϕ_α is feasible in the background theory. An important aspect of this mechanism addresses what to do if ϕ_α is not feasible. In this case, a clause which excludes an infeasible subset of the assignment is added to the SAT solver. Most often, the SAT solver is modified to allow the addition of such clauses so as to avoid solving a whole new SAT problem each time a clause is added. This modification is not very involved, and generally simply involves backtracking to an appropriate point and noting any unit propagation which might occur as a result of the new clause at that point. In addition, generating a clause which excludes a *minimal* infeasible subset of the assignment is often desirable, much as clause minimization plays an important role in propositional SAT solvers. While the mechanism for generating minimal infeasible sets varies quite a bit from theory to theory, it is also possible to find minimal infeasible sets for an arbitrary theory using a decision procedure for conjunctions [MR02]. Perhaps due to the relative ease of implementation, fully lazy SMT solving was the first reported in the literature [MR02, WW99].

At the more eager end of the spectrum of lazy SMT solvers, conjunctions of literals are checked for feasibility by a theory solver in lock step with the *partial* truth assignments built up by the SAT solver. Every time a SAT solver extends its assignment with a truth value for an atom, a theory solver is notified and a feasibility check occurs. This has the advantage of directing the SAT solver to new assignments sooner but generally requires more feasibility checks. The majority of current SMT solvers work at this

more eager end of spectrum, at least when interpreting relatively simple theories. The cost of the increased number of feasibility checks is typically offset to some degree or other with the use of an incremental and backtrackable theory solver.

3.1.5 DPLL(T)

DPLL(T) [GHN+04] is a notable framework for intersecting lazy SMT solvers which operates at the more eager end of the lazy SMT spectrum. DPLL(T) has become a common method of integrating a propositional SAT solver with a theory solver. In particular, the DPLL(T) framework defines an interface for a theory solver and its usage with a SAT solver in such a way as to allow modular integration of an arbitrary theory solver with a SAT solver. This interface may be viewed as an implementation of modular proof system presented in [NOT04]. The interface provides for a degree of on-the-fly eagerness under the name *theory propagation*[2]. We describe this framework in some detail here to provide context for the next chapter, where we present a DPLL(T) theory solver for the theory of difference constraints.

The theory solver implements the following functions

`assign`(p, l) The function `assign` takes a literal p in the underlying theory and a list l as an arguments. The function adds p to the set of truth-assigned atoms and performs a feasibility check on that set. It returns true if the assertion of p succeeds. In this case, the list l is populated with literals whose atoms are found in the formula, each of which is entailed by the set of truth-assigned atoms under the given theory. Otherwise, the function returns false and the list is populated with an infeasible, assigned set of literals and containing p. Also, in the case of feasibility, the list need not be fully populated, allowing the theory solver to perform some propagation without requiring that it be complete, or even present.

`explain`(p) The function `explain` takes a literal p which was entailed by previously assigned atoms and returns an infeasible set of assigned literals X which contains $\neg p$. The set X is referred to as an *explanation*

[2]While the term *theory propagation* originated from the DPLL(T) work, the idea was reported in [CAB+02]. Earlier work incorporated propagation of theory specific consequences into a DPLL-like process [BSU97]; but the notion of separating the DPLL process from the theory and then communicating theory consequences in the form of lemmas appears to have been realized later.

and may be readily translated into a valid clause $\neg \bigwedge X$ for use by the DPLL procedure.

`unassign`(p) The function `unassign` takes the most recently assigned literal p as an argument, and the theory solver removes p from its assignment stack.

As originally presented in [GHN+04], a theory solver S works in lock step with a DPLL propositional SAT solver while the truth assignment is being extended. In particular, a DPLL sat solver is instrumented so that every assignment of a variable v_p standing for atom p is associated with a call to `assign()`. In our presentation, every unassignment during backtracking is associated with a call[3] to `unassign()`. In the event that the `assign` function propagates theory literals, the entailed literals are added to the assignment in the same way that literals entailed by unit clauses are added to the assignment. Performing theory propagation thus enables an interleaving of propositional reasoning and theory specific reasoning. However, in the context of a modern DPLL based SAT solver, this poses a problem for learning and non-chronological backtracking. Namely, learning normally would take place by means of resolving clauses which are already present in the formula; but when the truth value of a literal p is entailed by the theory, it may be that no such clause exists. The infeasible sets of literals X returned by the function `explain` solve this problem since they are readily converted to valid clauses $\bigvee\{\neg x \mid x \in X\}$. These clauses are transient, temporary, and are only referenced when the SAT solver is learning.

Exhaustive Theory Propagation

The DPLL(T) framework is often applied to simple theories with relatively inexpensive decision procedures. These in turn often allow relatively inexpensive exhaustive theory propagation, in which the `assign` method always propagates every implied truth value for an atom occurring in the formula. A nice side effect of exhaustive theory propagation is that it is no longer necessary to perform a feasibility check, since the DPLL driver will never pass the negation of an implied literal to the `assign` function. This idea was first exploited in [NO05].

[3] In the original, the theory solver would be asked to perform an entire backtrack sequence in one step. We prefer this fine-grained presentation for expository purposes and note that a theory solver capable of backtracking many steps at once can readily implement the `unassign()` procedure as a backtrack of length 1.

3.1.6 Multiple Theories and Nelson-Oppen

Another problem addressed in the SMT literature is the problem of theory combination, in which relations and functions belonging to multiple theories may be present in a given formula. Theory combination can be particularly useful when SMT problems are generated to analyze complex systems involving multiple functionalities. For example, a program analyzer may generate an SMT problem involving a theory of arrays and the theory of integer linear arithmetic. Such a formula may look something like

$$(ia[x] < ia[2x] + 3) \vee (2x = y + 1)$$

To address the problem of theory combination, it is useful to first define some notions. The *language* of a theory T, denoted $L(T)$ is the set of formulas constructed with symbols in the signature, logical symbols, equality, or parentheses. The *combination* of two theories $T \triangleq T_1 \otimes T_2$ refers to the theory (set of valid formulas) defined in terms of the union of the respective parts of T_1 and T_2:

- The signature of T is the union of the signatures of T_1 and T_2.
- The set of axioms of T is the union of the axioms of T_1 and the axioms of T_2.

The combination defines a set of valid formulas, namely those formulas which can be derived from the combined axioms. Note that the formulas in the combined theory may mix symbols from the respective component theories. For example, if T_1's signature contains a function symbol f and T_2's signature contains g, then $f(g(x))$ is a term in the combined theory.

Nelson-Oppen

The Nelson-Oppen[NO79] method (hereafter NO) is a standard mechanism for deciding formulas in a combination $T_1 \otimes T_2 \otimes \ldots \otimes T_i$ of theories. The procedure makes use of a decision procedure for conjunctions of atomic predicates in each theory T_i, and is applicable under the following conditions:

1. The formulas treated by the procedure fall in the quantifier free fragment of first order logic.
2. There is a decision procedure for conjunctions of quantifier-free atomic predicates for each theory T_i.
3. The signatures of each theory are disjoint.

4. There is a cardinal κ such that every satisfiable formula in each theory has a model of cardinality κ. (This condition differs from the traditional NO procedure and some of the implications of using this condition are discussed later in this section).

Purification

The NO procedure begins with an encoding step called *purification*, which is similar in spirit to the Tseitin CNF translation discussed in Chapter 2. The basic idea is to give a fresh variable to each application of a function. In particular, given a formula ϕ, purification selects any smallest sub-formula t rooted with a function symbol of arity > 0 appearing in ϕ and then rewrites ϕ into the form

$$\phi[t \mapsto x_t] \wedge x_t = t$$

where x_t is a fresh variable. Repeating this process recursively on $\phi[t \mapsto x_t]$ yields a *purified* formula. For example, purifying the formula

$$\phi \stackrel{\circ}{=} f(x) + g(y, z) < 2$$

may yield

$$\begin{array}{ll} & z_0 = f(x) \\ \wedge & z_1 = g(y, z) \\ \wedge & z_2 = z_0 + z_1 \\ \wedge & z_2 < 2 \end{array}$$

Observe that every atomic predicate in a purified formula contains at most one function symbol. Since the NO method applies to theories with disjoint signatures, it follows that every atomic predicate belongs to a single theory. In the example above, ϕ may be viewed as a combination of the logic of uninterpreted functions with equality[4] arithmetic. In the purified formula, each atomic predicate belongs to exactly one of these theories.

Combination

We finally are ready to state a decision procedure for combined theories. For simplicity, we assume that we work over a combination of two theories

[4]The logic of uninterpreted functions with equality allows function symbols to be applied to variables and equality tests between terms. The function symbols are uninterpreted, which is to say that the only assumptions made about them is that the result of a function application $f(a_1, a_2, \ldots, a_k)$ is unique. In other words, for all function symbols f, $\bar{x} = \bar{y} \rightarrow f(\bar{x}) = f(\bar{y})$.

$T \triangleq T_1 \otimes T_2$. Let ϕ be a quantifier free formula in the language of T. The NO method first purifies ϕ, resulting in a formula of the form $F_1 \wedge F_2 \wedge \phi'$, where

- ϕ' is a propositional combination of relation symbols applied to variables.
- F_1 (F_2) is a conjunction of equalities of the form $x_t = s$ with s a T_1 (T_2) function symbol.

The next step is to non-deterministically choose a truth assignment α to the atoms found in ϕ'. If $\phi'[p \mapsto \alpha(p)]$ is false, then a new assignment is chosen. If no new assignments exist, the formula is declared unsatisfiable. Otherwise, assume $\phi'[p \mapsto \alpha(p)]$ is true. Now define the formulas

$$T_i(\alpha) \triangleq \bigwedge\{p \mid \alpha(p) = 1, p \in L(T_i)\} \wedge \bigwedge\{\neg p \mid \alpha(p) = 0, p \in L(T_i)\}$$

for $i \in \{1, 2\}$.

The procedure then non-deterministically chooses a partition $\mathbf{P}$ over the variables occurring in the purified formula $\phi' \wedge F_1 \wedge F_2$. For each equivalence class in $\mathbf{P}$, choose a variable as a representative. For any variable x, let $\hat{x}$ refer to the representative of the equivalence class to which x belongs. We then define the formula

$$Ar(\mathbf{P}) \triangleq \bigwedge\{x = y \mid \hat{x} = \hat{y}\} \wedge \bigwedge\{x \neq y \mid \hat{x} \neq \hat{y}\}$$

We are ready to use the decision procedures for T_1 and T_2. In particular, we check the satisfiability of T_i for the formula $\psi_i \triangleq (T_i(\alpha) \wedge F_i \wedge Ar(\mathbf{P}))$ using the decision procedure for T_i. If both such formulas are satisfiable, then ϕ is declared satisfiable. Otherwise, we choose another partition of the variables if one exists. If not, then we choose another assignment if one exists. Otherwise, ϕ is declared unsatisfiable.

Theorem 3.1.2 (NO is Correct)**.**

Proof. (Sketch) We begin by observing that $F_1 \wedge F_2 \wedge \phi'$ is equisatisfiable to ϕ.

Suppose the procedure returns unsatisfiable, and assume for a contradiction that ϕ is satisfiable. Then $F_1 \wedge F_2 \wedge \phi'$ is satisfiable. Let α be a satisfying assignment to the variables in $F_1 \wedge F_2 \wedge \phi'$. Observe that there is a truth assignment α' to the atoms in ϕ' which satisfy ϕ'. Now we define a partition $\mathbf{P}$ with the property that $T_i(\alpha') \wedge F_i \wedge Ar(\mathbf{P})$ is satisfiable in

each corresponding T_i. Namely, $\mathbf{P}$ is the partition consisting of equivalence classes

$$\langle x \rangle \triangleq \{x' \mid \alpha(x') = \alpha(x)\}$$

for each variable x. Since such α' and $\mathbf{P}$ exist, the procedure must have returned satisfiable, a contradiction.

For soundness, let M_i be a T_i-model of ψ_i, $i \in \{1, 2\}$. Assume without loss that each M_i is of the same cardinality. Let α_i be an assignment mapping the variables in ψ_i to the domain of M_i witnessing the satisfiability of each ψ_i, $i \in \{1, 2\}$. We now construct a one-to-one mapping λ from the domain of M_1 to the domain of M_2:

$$\lambda(a) \triangleq \begin{cases} \alpha_2(x) & \text{if } \alpha_1(x) = a \\ \text{a distinct element of } dom(M_2) & \text{otherwise} \end{cases}$$

Observe that such a λ exists because M_1 and M_2 are of the same cardinality and each α_i is a witness to $Ar(\mathbf{P})$. With λ in hand, we may then give an interpretation to the symbols in the signature of T_1 over the domain of M_2. For example, if s is a symbol in the signature of T_1 with semantics $S_1 \subseteq dom(M_1)^w$, then we give it semantics

$$S_2 \triangleq \{a \in dom(M_2)^w \mid (\lambda^{-1}(a_1), \lambda^{-1}(a_2), \ldots, \lambda^{-1}(a_w)) \in S_1\}$$

Let $M_\star$ be the the structure M_2 extended to include interpretations of all symbols in the signature of T_1 in this way. Since $M_\star$ extends M_2, $M_\star \models \psi_2$. Since M_1 is isomorphic to the reduct of M_2 to the T_2 symbols, we have $M_\star \models \psi_1$. Hence $M_\star \models \psi_1 \wedge \psi_2$.

Since F_i is a conjunct of ψ_i, we have that $M_\star \models F_1 \wedge F_2$. It remains only to show that $M_\star \models \phi'$, since $F_1 \wedge F_2 \wedge \phi'$ is equisatisfiable to ϕ. Since $T_i(\alpha)$ is a conjunct of ψ_i, $M_\star \models T_1(\alpha) \wedge T_2(\alpha)$. From the definition of $T_i(\alpha)$, we have that $T_1(\alpha) \wedge T_2(\alpha) \models \phi'$, completing the proof of soundness.

Finally, we observe that since there are a finite number of truth assignments to the atoms in ϕ' and and a finite number of partitions of the variables in $\phi' \wedge F_1 \wedge F_2$, the procedure terminates. □

Convexity

A theory T is *convex* if for every finite set of atomic predicates A and every pair of equalities $x = y, x' = y'$ over variables if $A \models x = y \vee x' = y'$, then either $A \models x = y$ or $A \models x' = y'$. Convexity may be used to optimize the NO method. In particular, if we require that each T_i decision procedure performs complete equality propagation, then there is no longer a need to

select a partition over the variables. This observation fits in nicely with the DPLL(T) framework mentioned above. Namely, a DPLL(T) SMT solver can work directly with a combination of theories on a purified formula simply by using a T_i-solver for each theory which propagates all equalities.

However, not all theories are convex. The following example shows that the ordered theory of integers with equality is not convex.

$$\phi \triangleq x \leq 1 \wedge x \geq 0 \wedge y = 0 \wedge z = 1$$

In particular, we have that $\phi \models x = y \vee x = z$ but $\phi \not\models x = y$ and $\phi \not\models x = z$. Traditionally, non-convex theories are handled by propagating disjunctions of equalities. However, it has also been shown that using a SAT solver to branch on equalities and dis-equalities can be effective, even when theories are convex [BCGS06]. More recently in [dMB07], a model reconciliation approach in which models of component theories are used as a heuristic to incrementally build up a partition was shown to be effective and also allows for the generation of theory literals on the fly. This method defaults to [BCGS06] when the heuristic fails to find new guiding information.

Discussion

Though somewhat idiosyncratic in detail, the Nelson-Oppen methodology can be a powerful tool to increase the expressivity of theories handled by SMT solvers. Our presentation of the method has replaced the requirement that component theories be stably infinite[5] with the requirement that all satisfiable formulas in all component theories have a model of a given cardinality. The condition of being stably infinite implies the existence of a cardinal κ such that all component theories have models of cardinality κ by means of the upward Lowenheim-Skolem theorem. Our restriction allows component theories to have finite models and appears to simplify the correctness proof. The correctness proof for the requirement of being stably infinite may be found in [TH96]. The original proof [NO79] was incorrect in that it did not recognize the requirement of being stably infinite, and also replaced the idea of an arrangement of a variable partition with equality propagation on the part of the component theories. The idea of using arrangements of a variable partition was also presented in [TH96]. In addition, that work incorporates some easy optimizations which we have left out for simplicity. In particular, the variable partition may be restricted to shared

[5] A theory is stably infinite if every satisfiable formula has an infinite model.

variables and represented in a more compact manner by eliminating redundant (dis)equalities where the redundancy is due to the inherent transitivity and symmetry of the theory of equality.

3.2 Flexible Propagation

In this section we present simple mechanisms for optimizing SMT solvers by allowing more flexibility in the relative timing of theory-specific and propositional reasoning and subsequently exploiting this flexibility. We begin with the observation that different types of constraint propagation have different computational costs, and often performing one type of propagation or feasibility check can preclude the need to to perform operations of another type. For example consider the unsatisfiable formula

$$p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge \neg r$$

where p, q, r are atoms in some theory. Unit propagation leads to a conflict, but a theory solver may check the feasibility and/or consequences of the truth assignments to p, q, r as their values are forced by the DPLL procedure. If the theory specific reasoning is significantly more expensive than unit propagation, the solver will have wasted some resources. Conversely, consider the unsatisfiable formula

$$(x \leq 0) \wedge (x \geq 1) \wedge p_0 \wedge (\neg p_0 \vee p_1) \wedge (\neg p_1 \vee p_2) \wedge \ldots \wedge (\neg p_{n-1} \vee p_n)$$

Here, the theory component is trivially unsatisfiable while unit propagation is irrelevant and more expensive. While both examples are quite contrived and have little interest in and of themselves, a solver can encounter similar situations quite often in the form of sub-problems while it is executing.

We are interested in establishing a more flexible strategy of interleaving theory and propositional reasoning which allows a solver to better handle both kinds of situations. This is accomplished on two levels, within the SAT solver and within the theory solver.

3.2.1 SAT Solver Instrumentation

One may view a DPLL based SAT solver as process which combines two types of reasoning: unit propagation and branching. Unit propagation is less expensive than branching, and is thus given preference in the DPLL framework. In the DPLL(T) framework, additional types of reasoning are added to this combination; namely theory consistency checks and theory

propagation. To prioritize the processing of the theory solver with respect to the processing of the DPLL process, we instrument the SAT solver to send events to the theory solver and listen for events from the theory solver at certain points in the DPLL process. This instrumentation is relatively simple, and does not interfere with the various modern techniques described in chapter 2.

In particular, we have the DPLL process send notification of the following events to the theory solver:

- *Assignment.* When variables representing theory atoms are given a truth value.
- *Unassignment.* When variables representing theory atoms are unassigned (during backtracking).
- *NoBCP.* When the DPLL process is ready to branch non-deterministically on a truth value, *i.e.* when there are no unit clauses.

Each one of these events corresponds to a method implemented by a theory solver, specified below.

The theory solver in turn sends event notifications to the sat solver in the form of *theory implications* $\bigwedge R_p \rightarrow p$, where p is a literal and R_p is a set of literals each of which is asserted under the current truth assignment. These events may either indicate inconsistency, in which case $p \mapsto 0$ is part of the truth assignment and assigned later than all predicates in R_p; or they may indicate propagation, in which case p is unassigned. The SAT solver listens for these T-implication events after every assignment and also at *nobcp* events. As in the DPLL(T) framework, in the event of an inconsistency, the SAT solver will backtrack and in the event of a T-implication, the implied predicate will be added to the truth assignment, possibly inducing more unit propagation. The resulting control flow is indicated in Figure 3.2.1.

3.2.2 Theory Solver Processing

The communication of choice points from the SAT solver to the theory solver allows the theory solver to make a more informed decision about what kind of processing should be applied. In particular, the theory solver may implement light-weight operations on every assignment and implement more costly operations at choice points. The two-level processing in this framework induces a situation in which theory atoms undergo different degrees of interpretation by the theory solver. This leads to the question of what types of theory reasoning are appropriate for these minor and major degrees

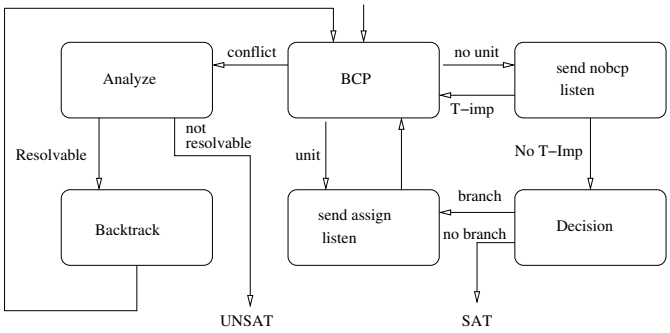


Figure 3.1: Control flow of a SAT solver instrumented for flexible theory propagation. The SAT solver sends *assignment* events and *nobcp* events to the theory solver and listens for theory implications from the theory solver. The theory solver may delay more expensive consistency checks or propagation until it receives a *nobcp* event notification.

of interpretation. One simple answer to this question is to have consistency checks play the role of minor reasoning (interleaved with Boolean constraint propagation) and to reserve theory propagation for major reasoning to help reduce the number of choice points.

While associating a full consistency check with every assignment is probably too expensive for some theories, this scenario has several benefits for theories with efficient incremental consistency checking algorithms. First, theory propagation is more expensive than consistency checking, since theory propagation may be reduced to a set of consistency checks[6]. Thus this division of labor conforms to the idea of delaying expensive reasoning while cheaper reasoning is relevant. Second, establishing consistency may facilitate theory propagation[7]. Third, theory propagation can reduce the cost of consistency checking simply because theory-implied predicates are already known to be consistent, and so fewer consistency checks are required.

Here we present a simple literal labelling mechanism which helps to realize the benefits of this scenario. In particular, we have the theory solver associate an annotation l_p with each literal p over an atom which appears in a given problem. The annotation can take any value from $\{\Pi, \Sigma, \Delta, \Lambda\}$

[6]In this context, we have a finite set of theory predicates, each which may or may not be a candidate for propagation. For every predicate p which is to be tested for propagation under an assignment A, the theory solver determines the validity of $A \models p$, which is equivalent to testing the consistency of $A \wedge \neg p$.

[7]We show an example of this in Chapter 4.

with the following intended meanings.

Π: Literals whose consequences have been found (propagated constraints). These literals have undergone both light-weight and heavy-weight processing.

Δ: Literals which have been identified as consequences of constraints labelled Π.

Σ: Literals which have been assigned, but whose consequences have not been found yet. These literals have undergone light-weight processing, but not heavy-weight processing.

Λ: Literals which have not undergone any processing and are not identified as consequences of literals labelled Π.

For convenience, we use the labels Π, Δ, Σ and Λ interchangeably with the set of literals which have the respective label.

It is straightforward to maintain labels with these properties via methods `assign`, `tprop` and `unassign`, which correspond to the assignment, nobpc, and unassignment event notifications received from the driving DPLL process: Whenever a literal p is passed to `assign` which is labelled Λ, the theory solver performs a consistency check. If consistent, we re-label p with Σ, otherwise an appropriate theory implication is generated and returned. Whenever `tprop()` is called, literals labelled Σ or Δ are labelled Π, one at a time. After each such relabelling, a search for theory consequences $\Pi \models p$ takes place where $p \in \Lambda$. If some such consequences are found, they are re-labelled Δ and theory implications are generated and returned to the driving DPLL process. Whenever backtracking occurs, all constraints which become unassigned are labelled Λ. The functions are detailed in pseudocode under Algorithm 3.2.1. Figure 3.2.2 shows the corresponding state transitions. The resulting control flow is indicated in Figure 3.2.1.

Algorithm 3.2.1. *Theory Solver Relabelling*

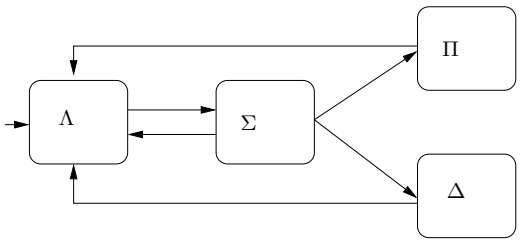


Figure 3.2: Transition diagram for literals during flexible propagation. Initially, literals are labelled Λ. Upon assignment, literals are labelled Σ. Theory propagation labels literals one of Π or Δ depending on whether Δ is a consequence of Π. When literals are unassigned, they are labelled Λ.

```
assign(p)
    if l_p = Λ then
        if sat(Π ∪ Σ ∪ {p}) then
            l_p ← Σ
            return ⊤
        else
            return explain (¬p)
    else return ⊤
```

```
unassign(p)
    l_p ← Λ
```

```
tprop()
    loop
        let C = {q | q ∈ Λ, Π ⊢ q}
        if C ≠ ∅ then
            l_q ← Δ, ∀q ∈ C
            return {explain(q) | q ∈ C}
        else if Σ ∪ Δ ≠ ∅ then
            let p ∈ Σ ∪ Δ
            l_p ← Π
        else
            return ⊤
```

Pseudocode for literal relabelling in a theory solver. The methods `assign`, `unassign`, and `tprop` correspond to event notifications from the driving DPLL solver. The function `explain` generates a theory implication which justifies a literal. The method `tprop` implements theory propagation with a theory consequence finder, whose consequence relation is denoted $\vdash$. The

value $\top$ is used to indicate an empty theory implication.

The labelling mechanism enforces the following properties.

1. The set $\Pi \cup \Delta \cup \Sigma$ is theory consistent. It is easy to see that Σ is consistent, because literals are added to this set only after consistency checks occur. Also, we observe that if Π is consistent, then $\Pi \cup \Delta$ is consistent since $\Pi \models \Delta$ in the given theory. Observing that literals are only labelled Π if they are first labelled Σ or Δ, the result follows by induction.

2. Consistency checking is only applied to literals labelled Λ. This guarantees a minimal number of calls for consistency checking.

3. If non-empty, the set Π is consistent with respect to unit propagation in the driving DPLL process. Since literals in Π are the only ones which act as antecedents in theory propagation, enforcing this property effectively filters calls to theory propagation.

4. The consequence relation $\Pi \models \Delta$ holds in the given theory.

3.2.3 Relabelling with Exhaustive Propagation

In Section 3.1.5 we observed a possible optimization for the DPLL(T) framework in the event the theory solver implements exhaustive propagation. Similarly, some optimizations are possible within the relabelling framework. First, we observe that since $\Pi \models \Delta$ we have that

$$\{p \mid \Pi \models p\} = \{p \mid \Pi \cup \Delta \models p\}$$

Since exhaustive propagation finds all consequences of Π, there is no need to search for new consequences arising from literals labelled Δ. This fact can be used to reduce the number of calls to an exhaustive consequence finder, and also allows the consequence finder to find consequences of a smaller set of literals. We detail the corresponding pseudocode in Algorithm 3.2.2.

Algorithm 3.2.2. *exhaustive tprop*

```
tprop-exhaustive()
   loop
      let  C = {q | q ∈ Σ ∪ Λ, Π ⊢ q}
      if  C ≠ ∅ then
         l_q ← Δ, ∀q ∈ C
         return  {explain(q) | q ∈ C}
      else if  Σ ≠ ∅ then
         let p ∈ Σ
         l_p ← Π
      else
         return  ⊤
```

Exhaustive theory propagation with relabelling. Consequences within the set $\Sigma \cup \Lambda$ are found, rather than the set Λ. Additionally, literals labelled Δ are never labelled Π, leading to a smaller number of calls to the consequence finder.

3.2.4 Discussion

Flexible propagation may be viewed as a variant of the DPLL(T) framework in which a global propagation strategy may exploit differences in the cost of various types of reasoning to arrive at a solution faster. In addition to the work presented here, in which all theory specific reasoning falls in one theory, an extension of this idea which addresses the case of progressively richer theories has been detailed in [CM06]. The framework described here is also put to use in Chapter 4 for the case of difference constraints. An experimental evaluation in that context shows that flexible propagation can provide significant performance improvements over the standard DPLL(T) framework. The idea of delaying theory propagation and consistency checks was also put forth in [CAB+02]. However, in that work it is suggested that consistency checks or other light-weight theory processes are not interleaved with unit propagation, and there is no mention of optimizations for the case of exhaustive propagation or for treatment of literals identified as consequences. The work [WGG06] also demonstrates that delaying theory processing until choice points is effective. In general, our use of labels facilitates reasoning about and specification of the interface between the driving DPLL solver and the theory solver.

3.3 Unate Consistent Model Search

In this section, we present a method for SMT solving which, in contrast to the intersecting methods described above, searches *directly* for a model of a given formula. Our method may be seen as a generalization of DPLL which searches in the space of variable valuations. It is most closely related to GDPLL [MKS09], but differs in that the abstract algorithm is formulated in terms of variable valuations, and is analyzed in terms of proof graphs in a manner similar to DPLL resolution proofs. Consequently our method extends GDPLL with the important notion of forgetting deduced facts. For convenience, we call this class of search algorithms *unate consistent search* (UCS), since it is based on 1-variable local consistency.

To describe UCS, consider a quantifier free CNF formula ϕ

$$\phi \equiv \bigwedge\{c_1, c_2, \ldots, c_k\}$$

where each c_i is a clause. Now given a variable x, we denote by

$$\phi|_x \triangleq \bigwedge\{c_i \mid vars(c) = \{x\}\}$$

That is, $\phi|_x$ is the set of all x-unate clauses. We say ϕ is *unate consistent* if for every variable $x \in vars(\phi)$ it is the case that $\exists x.\phi|_x$. Unate consistency is simply variable-local consistency.

With these notions at hand, we are ready to define an abstract algorithm for deciding quantifier free conjunction of clauses which is parameterized by three procedures, `select`, `isUC` and `resolve`. The procedures are theory-specific and must implement the following interface

`select`(ϕ, α). This procedure takes a formula ϕ and a partial assignment α as arguments and returns a pair (x, a) where a is in the domain of x, $x \in vars(\phi[\alpha])$ and $(\phi[\alpha]|_x)[x \mapsto a]$ is true.

`isUC`(ϕ) This procedures tests the unate consistency of a conjunction of clauses. It returns true if the formula ϕ is unate consistent and false otherwise.

`resolve`(ϕ, α, x) This procedure takes a formula ϕ, a partial assignment α and a variable x such that $\phi[\alpha]|_x$ is unsatisfiable, and returns a clause w such that

1. $\phi \models w$
2. $w[\alpha]$ contains no variables and evaluates to false.

The abstract algorithm `UC-Search` is detailed in pseudocode under Algorithm 3.3.1. It takes two arguments, ϕ, α, where ϕ is a formula and α is a partial assignment to the variables in ϕ. It returns either a pair $(1, \alpha)$ where α is a satisfying assignment for ϕ or a pair $(0, w)$ where w is a false clause *i.e* a clause whose every literal has no variables and such that each literal evaluates to false.

Algorithm 3.3.1. *Unate Consistent Search Algorithm*

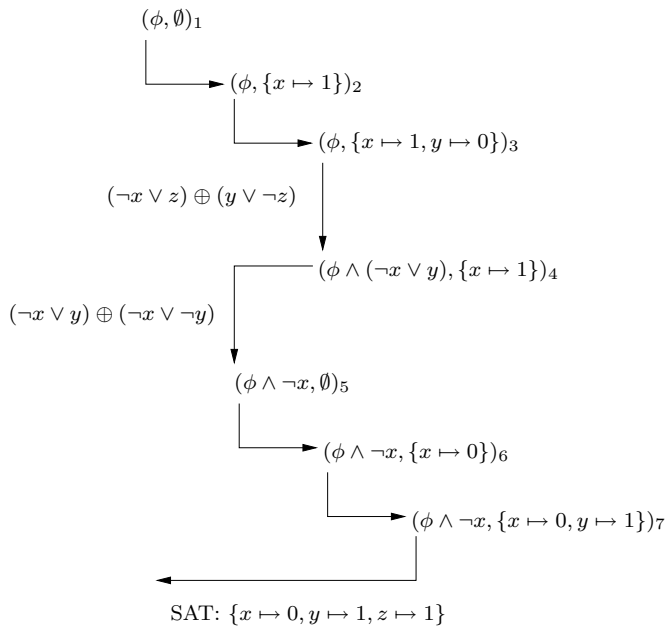
```
     UC-Search(φ, α)
1      if isUC(φ[α]) then
2        let (x, a) = select(φ, α)
3        if vars(φ[α]) = {x} then
4          return (1, α ∪ {x ↦ a})
5        let (r, w) = UC-Search(φ, α ∪ {x ↦ a})
6        if r = 1 or x ∉ vars(w) then
7          return (r, w)
8        else
9          return UC-Search(φ ∧ w, α)
10     else
11       let x be s.t. φ[α]|_x is unsat
12       let w = resolve(φ, α, x)
13       return (0, w)
```

`UC-Search` is much like `DPLL-BJ` except that the recursion depth does not correspond to the size of the variable assignment, and variables which are constrained are not automatically assigned. These facts highlight that `UC-Search` is in some ways fundamentally different than `DPLL`. Nonetheless many similarities exist, such as being a depth first search over the space of variable valuations coupled with a proof generation procedure. As such `UC-Search` qualifies as an extended class of instantiations of GDPLL.

Example 3.3.2 (`UC-Search` run)**.** Consider the propositional formula ϕ

$$\phi \triangleq (x \vee y) \wedge (\neg y \vee z) \wedge (\neg x \vee z) \wedge (\neg z \vee y) \wedge (\neg x \vee \neg y)$$

A diagram of a possible trace of the algorithm deciding ϕ follows.



The nodes in the graph are annotated with subscripted pairs $(\phi, \alpha)_n$ representing a sequence of calls to the procedure `UC-Search`. The recursion depth in which each call occurs is represented by horizontal position. Each call opens a new scope which in turn triggers at most 2 calls directly contained in that scope. The first such call occurs at line 5 and the second at line 9 in the pseudocode listing. No calls are triggered when the formula $\phi[\alpha]$ is not unate consistent nor when the assignment satisfies the formula. Outgoing edges of calls which fail the consistency check are labelled with resolution operations $c \oplus d$. Observe that the clause $(\neg x \vee y)$ derived between calls 3 and 4 is subsequently forgotten because it falls out of the scope of call 2.

We prove some basic facts about `UC-Search`.

Theorem 3.3.3. `UC-Search` *Soundness*

Proof. `UC-Search` returns either a value $(1, \alpha)$, or a value $(0, w)$. In the first case, the procedure must have returned $(1, \alpha \cup \{x \mapsto a\})$ at line 5 when $\phi[\alpha]$ is unate consistent and $x \mapsto a$ is a satisfying assignment for $\phi[\alpha]|_x$ and $\{x\} = vars(\phi[\alpha])$. Hence the assignment $\alpha \cup \{x \mapsto a\}$ satisfies the formula.

In the second, case, w is a clause made up of theory literals over constants, i.e. w is variable-free. The procedure `resolve()` produces w, and guarantees that w is a consequence of ϕ and that w evaluates to false. $\square$

The following theorem shows that the `UC-Search` procedure is guaranteed to make progress in the sense that the search space becomes more and more constrained over time.

Theorem 3.3.4. `UC-Search` *Progress*
Let $U(\phi, \alpha)$ denote the set of all unate-feasible 1-extensions of α

$$U(\phi, \alpha) \triangleq \{(x, a) \mid x \in vars(\phi[\alpha]), (\phi[\alpha]|_x)[x \mapsto a] \text{ is true}\}$$

Let $(\phi_1, \alpha_1)(\phi_2, \alpha_2) \ldots (\phi_k, \alpha_k)$ denote a sequence of calls to `UC-Search` during a run of the procedure. Consider a cycle with two calls i, j, such that $i > j$ and $\alpha_i = \alpha_j$. Let $\alpha \triangleq \alpha_i$. Then $U(\phi_i, \alpha) \subset U(\phi_j, \alpha)$.

Proof. Without loss of generality, we consider only a minimal sequence such that $i > j$ and $\alpha_i = \alpha_j$, since any subsequent cycles will only further constrain the space. Consider the following two cases.

1. Case 1. $\alpha \subseteq \alpha_k$, for all $j \leq k \leq i$. In this case, call i occurs at line 9, and $\phi_i \equiv \phi_j \wedge w$ for some clause w returned by resolve. The procedure `resolve()` returns a clause which excludes an assignment $x \mapsto a$ such that x does not have an assignment in α. Moreover $x \mapsto a$ was a feasible assignment under $\phi_j[\alpha]$. Hence $U(\phi_i, \alpha) \subset U(\phi_j, \alpha)$.

2. Case 2. There is a k with $j \leq k \leq i$ such that $\alpha_k \subset \alpha$. This case is impossible because when the procedure backtracks over α, it either terminates or a clause w' is conjoined which excludes some subassignment α' of α. Moreover, once this clause is conjoined, all subsequent calls either fall in a call scope in which ϕ contains the clause w' or the procedure backtracks out of this scope with a new clause v' which excludes some subassignment of α'.

□

Progress is an analog of the termination argument for a DPLL solver which learns asserting clauses [ZM03]. Of course, one cannot guarantee termination without specifying something more about the theory or the resolution procedure. However, progress is more subtle in UCS because of the fact that when a variable is unate constrained it may or may not be assigned under α. This situation leads to the possibility of the procedure cycling with respect to a partial assignment. Progress simply says that whenever this happens, the procedure is in a state in which the search space is properly constrained with respect to the variable order in which the variables

are assigned. Note that progress takes place even though `UC-Search` forgets clauses from deeper in the call stack. Thus progress allows a solver to safely forget clauses, but still there is no limit on the minimum number of clauses necessary to guarantee progress because many learned unate clauses can be recorded under a given assignment before the procedure backtracks over that assignment. Also note that as stated, progress requires that upon learning a clause w, `UC-Search` backtracks to the maximal assignment under which w is unate. A similar notion of progress holds if `UC-Search` backtracks to the minimal assignment, which we omit for simplicity.

The notion of progress is extremely weak by comparison to termination of DPLL by asserting clauses; which brings us directly to the question of exactly when `UC-Search` terminates. Having established progress, it is not hard to see that if the closure of a finite set of clauses under `resolve()` is finite, then the procedure terminates. However, for arbitrary variable domains, `resolve()` is not necessarily finite. For example, from

$$(x > 1) \wedge (y > x + 1) \wedge (x > y + 1)$$

`resolve` may produce $y > 2$ and subsequently $x > 3$, $y > 4$, etc.

To better address this issue, we introduce the following notion.

Definition 3.3.5 (Specificity)**.** We say that `resolve` is *specific* if

$$\texttt{resolve}(\phi, \alpha, x) = \texttt{resolve}(\bigwedge\{w \mid w \in \phi, vars(w[\alpha]) = \{x\}\}, \alpha, x)$$

Specificity ensures that `resolve` in effect only takes into account x-unate clauses when `resolve` eliminates x. If `resolve()` is specific, it becomes useful to think of a proof as a graph whose nodes are clauses similar to a propositional proof graph as presented in Section 2.2.2 (page 6).

We define the proof graph traced by `UC-Search` in terms of the input-output relation of the procedure `resolve`. If `resolve` is specific, then for each call

$$v = \texttt{resolve}(\bigwedge W, \alpha, x)$$

v is a consequence of some subset of clauses $W' \subseteq W$ where each $w \in W'$ is x-unate under α. We assume that for a given implementation of `resolve()`, W' is a concrete set. The proof graph then consists of one edge (w, v) for each $w \in W'$. For example, consider a call to $\texttt{resolve}(c \wedge d \wedge e \wedge f, \alpha, x)$ which results in the conclusion g. Then the proof graph representing this step may consist of the edges $(d, g), (e, g), (f, g)$, provided `resolve()` found that $d, e, f \models g$. The pivot variable corresponding to each edge would be x.

Consider the topologic properties of a proof graph generated by calls to `resolve()` in `UC-Search`. The graph is not necessarily tree-like nor necessarily regular, because the variables are chosen in any order. One may restrict the form of resolution using the following notion.

Definition 3.3.6 (Exhaustively Asserting)**.** We say that `UC-Search` is *exhaustively asserting* if any variable chosen after a conflict is the variable constrained by the last learned clause.

To illustrate this property, consider a backtrack sequence. Every time there is an inconsistency, `UC-Search` returns a clause w derived by `resolve`. Either w has no variables, and the problem is unsatisfiable, or the clause w has a variable x which is maximal in the search, and w excludes an assignment $\alpha \cup \{x \mapsto a\}$. In this later case, $(\phi \wedge w)[\alpha]$ may or may not be unate consistent. If it is not unate consistent, the backtrack sequence is not maximal and `resolve` is called again. Otherwise, $(\phi \wedge w)[\alpha]$ is unate consistent and a free variable is selected. In the case that `UC-Search` always selects x within such a context, we say it is exhaustively asserting.

Theorem 3.3.7 (Restricted Resolution)**.** If `UC-Search` is exhaustively asserting and `resolve` is specific then the proof graph traced by `UC-Search` is tree-like and regular.

Proof. (sketch) The key insight is to divide learned clauses into two cases. Every added learned clause w is associated with a call to `UC-Search` which triggers a consistency check. If the check succeeds, then the learned clause is immediately satisfied by some assignment $\alpha \cup \{x \mapsto a\}$ because `UC-Search` is exhaustively asserting. Since `resolve` is specific, w cannot be an antecedent of any other learned clause as long as the search explores a proper extension of α (including with assignments to x). When the search backtracks to a proper sub-assignment of α, w falls out of scope and may be an antecedent of a single subsequent learned clause, by a resolution step which pivots on x.

If the check fails for some variable x, `resolve` is called again resulting in a clause w', and w together with any other learned clauses constraining x may be an antecedent of w'.

In both cases, the learned clauses are forgotten once they fall out of scope, so they give rise to at most one consequence. Regularity then follows because the pivot variables always follow the reverse order in which they are assigned. □

Restricted resolution, in turn, allows us to relax the requirements on `resolve()` necessary for termination. Consider the property of finite width:

Definition 3.3.8 (Finite Width)**.** Given a formula ϕ and a distinguished variable $x \in vars(\phi)$, let

$$r(x) \triangleq \{w \mid \exists \alpha \,.\, \phi[\alpha]|_x \text{ is unsat, and } w = \texttt{resolve}(\phi, \alpha, x)\}$$

denote the set of all derivable clauses under any variable valuation around x. We say that `resolve` has *finite width* for a formula ϕ, if $r(x)$ is finite for all $x \in vars(\phi)$.

Theorem 3.3.9. *Termination Sufficiency*
`UC-Search` terminates if

1. `resolve` is specific; and
2. `resolve` has finite width; and
3. `UC-Search` is exhaustively asserting

Proof. Having established progress (Theorem 3.3.4), it will suffice to show that the set of learned clauses is finite. Since `resolve` has finite width it will suffice to show that every learned clause falls in the k-closure of `resolve` for some bound k. By Theorem 3.3.7, the proof graph is regular, and so for a formula of n variables, every learned clause falls in the n-closure of `resolve`. □

Since progress and termination implies completeness, we have a convenient criterion for establishing completeness provided an appropriate implementation of `resolve()`. Note however that the proof relies indirectly on the fact that the resolution graph is tree-like, *i.e* that `UC-Search` *forgets* clauses on backtracking. This is contrary to the intuition that the more clauses one adds to the formula, the "closer" to proving unsatisfiability. The potential problem introduced by keeping clauses if `resolve` has finite width but infinite closure is that one risks creating infinitely long chains of resolution steps.

We now briefly address the question of the range of possible theories which may be treated by UCS. Obviously UCS cannot treat any theory which is unate-undecidable; we are unaware of any such theories. The real question is whether or not a theory admits a `resolve()` procedure. Aside from the requirements on `resolve` for progress and termination, a potentially limiting factor is that `resolve()` must return a quantifier free clause. One possible avenue of further analysis on this limitation is the relationship

between `resolve()` and interpolation. Namely, `resolve(`ϕ, α, x`)` may be seen as computing an interpolant of the incompatible pair

$$(\bigwedge\{c \in \phi \mid x \in vars(c)\}, \bigwedge\{x = a \mid x \mapsto a \in \alpha\})$$

Thus we can conclude that UCS may be applied to theories which admit quantifier-free interpolation in the form of a clause whenever one formula from the incompatible pair is a conjunction of variable-constant equalities and the other is a conjunction of clauses. This criterion seems arbitrary, but it is sufficiently lax to allow for many interesting theories.

3.3.1 Discussion

Much work has been done generalizing DPLL to richer logics [BSU97, BT03, BvdPTZ07, MKS09]. Unlike many generalizations of DPLL, UCS (and several instantiations of GDPLL) do not case split on truth values of atoms, but rather on variable valuations. These methods thus search more directly for a model than intersecting methods. GDPLL is more general than UCS; it does not require the use of variable valuations. By focusing on variable valuations, we arrive at termination conditions for UCS based on the topology of the proof graph and along the way identify which derived clauses may be forgotten. Neither method has been extended to incorporate intersecting methods. However, it is noteworthy that some work has already been done which mixes model search with intersecting methods. In [GSF08] the model from a theory solver is used to guide literal selection in the driving DPLL process; in [BDdM08], new literals are derived by join operations, constraining the space of partial models which can be visited in the search. These works show that consideration of the interpreted models can both guide a DPLL search over the space of uninterpreted literals and be used to constrain it.

3.4 Conclusion

SMT solving vastly expands the expressivity of propositional satisfiability solvers, and in doing so also expands its applicability to verification. The major developments in the field vary in content from the purely theoretical, such as the principles behind Nelson-Oppen theory combination, to the practical. In intersecting frameworks, practical considerations at the abstract level largely address the problem of integrating a propositional satisfiability solver with one or more theory solvers. To date, all the practical

works we are aware of propose a relatively static balance between propositional and theory specific reasoning components. However, it is clear that there is no best static balance between these components. For this reason, we anticipate future work to address the issue of balancing components more dynamically. Direct methods side-step the problem of dynamically balancing propositional and theory-specific reasoning and facilitate direct model search unencumbered by a search over truth values. Such methods are not as well studied as intersecting methods but seem to possess some potential and studying them allows us to try to apply the principles underlying effective propositional solving more directly to the case of richer logics.

Chapter 4

A Difference Logic Engine

In this chapter we present a theory solver for the theory of difference constraints for use in a lazy SMT solver, organized as follows. In section 4.1 we place our solver in the framework of flexible propagation with exhaustive propagation and set forth some basic concepts regarding difference constraints. In sections 4.2 and 4.3 we describe our implementation of incremental consistency checks, and exhaustive theory propagation respectively. In section 4.4 we present experimental results and we conclude in section 4.5.

4.1 Background

4.1.1 Solver Instantiation

Referring to the flexible propagation framework described in Section 3.2, our solver requires concrete instantiations of

- Incremental consistency checks, as detailed in the `assign` function of Algorithm 3.2.1. To fulfill this requirement, we will present an algorithm implementing the call to `sat()` within the function `assign`.
- Incremental exhaustive theory propagation, as detailed in Algorithm 3.2.2. We will present an implementation of consequence finding, which in the pseudocode is indicated by defining the set of consequences C (line 3).

In both of these endeavors, we will make use the predicate labels $\{\Pi, \Sigma, \Delta, \Lambda\}$ described in Section 3.2 to denote the set of propagated, assigned, implied, and unassigned constraints respectively. In this chapter we do *not* address

the implementation of relabelling, as this has been sufficiently detailed in Section 3.2.

4.1.2 Difference Constraints and Graphs

Difference constraints are simple linear constraints of the form

$$x - y \leq c$$

where x, y are variables, and c is a constant. This class of constraints is of particular interest for timing related problems including schedulability, circuit timing analysis, and bounded model checking of timed automata [NMA+02, Cot05]. In addition, difference constraints can be used as an abstraction for general linear constraints and many problems involving general linear constraints are dominated by difference constraints.

Constraint Graphs

A well-known graphical representation of conjunctions of difference constraints provides a structure on which we can derive an efficient decision procedures and theory propagation:

Definition 4.1.1 (Constraint graph)**.** *Let S be a set of difference constraints and let G be the graph comprised of one weighted edge $x \xrightarrow{c} y$ for every constraint $x - y \leq c$ in S. We call G the constraint graph of S.*

An example constraint graph is shown in Figure 4.1. The following well-known theorem characterizes the relationship between conjunctions of difference constraints and shortest paths in constraint graphs.

Theorem 4.1.2. *Let Γ be a conjunction of difference constraints, and let G be the constraint graph of Γ. Then Γ is satisfiable if and only if there is no negative cycle in G. Moreover $\Gamma \models x - y \leq c$ if and only if y is reachable from x in G and $c \geq d_{xy}$ where d_{xy} is the length of a shortest path from x to y in G.*

The correspondence between constraint graphs and sets of constraints is clear, making it possible to refer unambiguously to a set of constraints by its constraint graph and *vice versa*. As a result and in the interest of simplifying notation, we do not distinguish between constraint graphs and sets of constraints. For example given a set of constraints S we may refer to *the graph S*, and likewise given a constraint graph G we may refer to *the set*

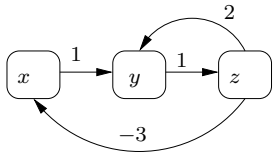


Figure 4.1: A constraint graph for the difference constraints $x - y \leq 1$, $y - z \leq 1$, $z - x \leq -3$, $z - y \leq 2$.

of constraints G. Similarly, we may refer to *the constraint* $x \xrightarrow{c} y$ even though the notation refers to an edge in a constraint graph. This practice allows convenient (and further) abuse of the notation associated with predicate labels $\{\Pi, \Sigma, \Delta, \Lambda\}$, which in the context of this chapter stand for sets of difference constraints as well as their respective constraint graphs.

Expressivity

One may encode a slightly broader class of constraints in the form of difference constraints. In particular, $x - y \geq c$ can take the form $y - x \leq -c$. Additionally, $x \leq y$ takes the difference constraint form $x - y \leq 0$.

Constant bounds on variables $x \leq c$ can be handled as follows. Let ϕ be a formula containing difference constraints and constant bounds on variables $x_i \leq c_i$ for $i \in I$. We consider the formula

$$\phi' \triangleq \exists z.\phi[(x_i \leq c_i) \mapsto (x_i - z \leq c_i), i \in I]$$

Clearly any satisfying assignment to ϕ can be extended to a satisfying assignment for ϕ' by letting $z = 0$. Conversely, any satisfying assignment α' to ϕ' induces a satisfying assignment α to ϕ where $\alpha(x) \triangleq \alpha'(x) - \alpha'(z)$.

Finally, we address the issue of strict inequalities $x - y < c$, which are a necessary result of negating non-strict inequalities. First, we observe that if the numerical domain consists of the integers, we can just replace $x - y < c$ with $x - y \leq c - 1$. For a continuous domain such as the rationals, we need a more general treatment of upper bounds, which can be accomplished using Minkowski sums. With each constraint $x - y \leq c$, we associate an interval $[-\infty \ldots c]$ and likewise with $x - y < c$ we associate $[-\infty \ldots c)$. We then adapt Theorem 4.1.2 to use constraints graphs whose edges are labelled with constraints' intervals rather than constraints' constants. Using the Minkowski sum of two intervals i, j as follows

$$i \oplus j \triangleq \{x + y \mid x \in i, y \in j\}$$

we may associate an interval to every path in the constraint graph. The theorem may be readily applied by defining a negative path as one whose interval contains only negative members. For notational clarity in the remainder of this chapter, we will only consider the case that all constraints are non-strict with the understanding that strict constraints can be treated as described here.

4.2 Consistency Checks

In light of Theorem 4.1.2, one way to show that a set of difference constraints Γ is consistent is to show that Γ's constraint graph G contains no negative cycle. This in turn can be accomplished by establishing a *valid potential function*, which is a function π on the vertices of a graph satisfying $\pi(x) + c - \pi(y) \geq 0$ for every edge $x \xrightarrow{c} y$ in G. A valid potential function may readily be used to establish lower bounds on shortest path lengths between arbitrary vertices (v_1, v_n):

$$\begin{aligned} \Sigma_{i=1}^{n-1}\pi(v_i) + c_i - \pi(v_{i+1}) &\geq 0 \\ \pi(v_1) - \pi(v_n) + \Sigma_{i=1}^{n-1}c_i &\geq 0 \\ \Sigma_{i=1}^{n-1}c_i &\geq \pi(v_n) - \pi(v_1) \end{aligned}$$

If one considers the case that $v_1 = v_n$, it follows immediately that the existence of a valid potential function guarantees that G contains no negative cycles. In addition, a valid potential function for a constraint graph G defines a satisfying assignment for the set Γ of difference constraints used to form G. In particular, if π is a valid potential function for G, then the function $v \mapsto -\pi(v)$ is a satisfying assignment for Γ.

In the flexible propagation framework, consistency checks occur during calls to `assign`, when a constraint $u \xrightarrow{d} v$ is added to the set of assigned constraints. If the constraint is labelled Δ, then there is no reason to perform a consistency check. Otherwise, the constraint is labelled Λ. In this latter case, `assign` must perform a consistency check on the set $\Pi \cup \Sigma \cup \{u \xrightarrow{d} v\}$. To solve this problem, we make use of an incremental consistency checking algorithm based largely on an incremental shortest paths and negative cycle detection algorithm due to Frigioni et al [FMSN98]. The algorithm and its presentation here are much simpler primarily because Frigioni et al. maintain extra information in order to solve the fully dynamic shortest paths problem, whereas this context only demands incremental consistency checks. In particular, we do not compute single source shortest paths, but

rather a potential function. This simplification as well as an exposition on dynamic consistency checking can be found in [RSJ95], which was brought to our attention after having completed this work.

Before detailing this algorithm, we first formally state the incremental consistency checking problem in terms of constraint graphs and potential functions:

Definition 4.2.1 (Incremental Consistency Checking)**.** *Given a directed graph G with weighted edges, a potential function π satisfying $\pi(x) + c - \pi(y) \geq 0$ for every edge $x \xrightarrow{c} y$, and an edge $u \xrightarrow{d} v$ not in G, find a potential function π' for the graph $G' = G \cup \{u \xrightarrow{d} v\}$ if one exists.*

The complete algorithm for this problem is given in pseudocode in Algorithm 4.2.2. The algorithm maintains a function γ on vertices which holds a conservative estimate on how much the potential function must change if the set of constraints is consistent. The function γ is refined by scanning outgoing edges from vertices for which the value of π' is known.

Algorithm 4.2.2. *Incremental Consistency Check*

```
γ(u) ← π(u) + d − π(v)
γ(w) ← 0 for all w ≠ v
while min(γ) < 0 ∧ γ(u) = 0
    s ← argmin(γ)
    π'(s) ← π(s) + γ(s)
    γ(s) ← 0
    for s −c→ t ∈ G do
        if π'(t) = π(t) then
            γ(t) ← min{γ(t), π'(s) + c − π(t)}
```

Incremental consistency checking algorithm, invoked by `assign` for a constraint $u \xrightarrow{d} v$ labelled Λ. If the outer loop terminates because $\gamma(u) < 0$, then the set of difference constraints is not consistent. Otherwise, once the outer loop terminates, π' is a valid potential function and $-\pi'$ defines a satisfying assignment for the set of difference constraints.

4.2.1 Proof of Correctness and Run Time

Lemma 4.2.3. *The value* $\min(\gamma)$ *is non-decreasing throughout the procedure.*

Proof. Whenever the algorithm updates $\gamma(z)$ to $\gamma'(z) \neq \gamma(z)$ for some vertex z, it does so either with the value 0, or with the value $\pi'(s)+c-\pi(t)$ for some edge $s \xrightarrow{c} t$ in G such that $t = z$. In the former case, we know $\gamma(z) < 0$ by the termination condition, and in the latter we have $\gamma'(z) = \pi'(s) + c - \pi(t) = (\pi(s) + c - \pi(t)) + \gamma(s) \geq \gamma(s)$, since $\pi(s) + c - \pi(t) \geq 0$. □

Lemma 4.2.4. *Assume the algorithm is at the beginning of the outer loop. Let z be any vertex such that $\gamma(z) < 0$. Then there is a path from u to z with length $L(z) = \pi(z) + \gamma(z) - \pi(u)$.*

Proof. By induction on the number of times the outermost loop is executed At the beginning of the first step, we only need to consider $L(v)$, and the result obviously holds. We now need to show that if $L(z)$ holds for steps $1..n$, then it is true after step $n + 1$ as well. Towards this end, we call s_n the vertex s as found in the nth refinement step; likewise, we call L_n, π'_n, and γ_n the values of L, π', and γ respectively as found at the beginning of the nth step. By the induction hypothesis, the invariant holds for vertex s_n at the beginning of step n, and moreover we may restrict our attention to those vertices z such that $\gamma_{n+1}(z) \neq \gamma_n(z)$. Now let $s_n \xrightarrow{c} z$ be an edge such that $\gamma_{n+1}(z) = \pi'_{n+1}(s_n) + c - \pi(z) < 0$. By the induction hypothesis, there is a path of length $L_n(s_n) + c$ from u to z. Moreover,

$$\begin{aligned} L_{n+1}(z) &= \pi(z) + \gamma_{n+1}(z) - \pi(u) \\ &= \pi(z) + \pi'_{n+1}(s_n) + c - \pi(z) - \pi(u) \\ &= \pi'_{n+1}(s_n) + c - \pi(u) \\ &= \pi(s_n) + \gamma_n(s_n) - \pi(u) + c \\ &= L_n(s_n) + c \end{aligned}$$

Hence L is correct at the beginning of step $n + 1$ as well.

□

Theorem 4.2.5. *The algorithm correctly identifies whether or not G' contains a negative cycle. Moreover, when there is no negative cycle the algorithm establishes a valid potential function for G'.*

Proof. We consider the various cases related to termination.

- **Case 1**. $\gamma(u) < 0$. From this it follows that $L(u) < 0$ and so there is a negative cycle. In this case, since the DPLL engine will backtrack, the original potential function π is kept and π' is discarded.

- **Case 2**. $\min(\gamma) = 0$ and $\gamma(u) = 0$ throughout.
  In this case we claim π' is a valid potential function. Let γ_i be the value of γ at the beginning of the ith iteration of the outer loop. We observe that $\forall v \,.\, \pi'(v) \leq \pi(v)$ and consider the following cases.

  - For each vertex v such that $\pi'(v) < \pi(v)$, $\pi'(v) = \pi(v) + \gamma_i(v)$ for some refinement step i. Then for every edge $v \xrightarrow{c} w \in G$, we have that $\gamma_{i+1}(w) \leq \pi'(v) + c - \pi(w)$ and so $\pi'(w) \leq \pi(w) + \gamma_{i+1}(w) \leq \pi(w) + \pi'(v) + c - \pi(w) = \pi'(v) + c$. Hence $\pi'(v) + c - \pi'(w) \geq 0$.
  - For each vertex v such that $\pi'(v) = \pi(v)$, we have $\pi'(v) + c - \pi'(w) = \pi(v) + c - \pi'(w) \geq \pi(v) + c - \pi(w) \geq 0$ for every $v \xrightarrow{c} w \in G$

  We conclude π' is a valid potential function with respect to all edges in G'.

In all cases, the algorithm either identifies the presence of a negative cycle, or it establishes a valid potential function π'. As noted above, a valid potential function precludes the existence of a negative cycle. □

Theorem 4.2.6. *The algorithm runs in time*[1] $\mathcal{O}(m + n \log n)$.

Proof. The algorithm scans every vertex once. If a Fibonacci heap is used to find $\operatorname{argmin}(\gamma)$ at each step, and for decreases in γ values, then the run time is $\mathcal{O}(m + n \log n)$. □

4.2.2 Experiences and Variations

For simplicity, we did not detail how to identify a negative cycle if the set of constraints is inconsistent. A negative cycle is a minimal inconsistent set of constraints and is returned by `assign` in the case of inconsistency. Roughly speaking, this can be accomplished by keeping track of the last edge $x \xrightarrow{c} y$ along which $\gamma(y)$ was refined for every vertex. Then every vertex in the negative cycle will have such an associated edge, those edges will form the negative cycle and may easily recovered.

In practice we found that the algorithm is much faster if we maintain for each vertex v a bit indicating whether or not its new potential $\pi'(v)$ has been found. With this information at hand, it is straightforward to update a single potential function rather than keeping two. In addition, this

[1] Whenever stating the complexity of graph algorithms, we use n for the number of vertices in the graph and m for the number of edges.

information can readily be used to skip the $\mathcal{O}(n)$ initialization of γ and to keep only vertices v with $\gamma(v) < 0$ in the priority queue. We found that the algorithm ran faster with a binary priority queue than with a Fibonacci heap, and also a bit faster when making use of Tarjan's subtree-enumeration trick [Tar81, CG96] for SSSP algorithms. Profiling benchmark problems each of which invokes hundreds of thousands of consistency checks indicated that this procedure was far less expensive than constraint propagation, although the two have similar time complexity.

4.3 Propagation

The method `tprop` described in Section 3.2 is responsible for constraint propagation. The procedure's task is to find a set of consequences C of the current assignment A, and a set of reasons R_c for each consequence $c \in C$. For difference constraints, by Theorem 4.1.2, this amounts to computing shortest paths in a constraint graph.

We present a complete incremental method for difference constraint propagation which makes use of the constraint labels Π, Σ, Δ, and Λ. The constraint propagation is divided into incremental steps, each of which selects a constraint c labelled Σ, relabels c with Π, and then finds the consequences of those constraints labelled Π from the set $\Sigma \cup \Lambda$, labelling them Δ. A single step may or may not find unassigned consequences. On every call to `tprop`, these incremental steps occur until either there are no constraints labelled Σ, or some unassigned consequences are found. Any unassigned consequences are returned to the DPLL engine for assignment and further unit propagation. We state the problem of a single incremental step in terms of constraint graphs and shortest paths below.

Definition 4.3.1 (Incremental complete difference constraint propagation)**.** *Let G, H be two edge disjoint constraint graphs, and let $x \xrightarrow{c} y \in H$ be a distinguished edge. Suppose that for every edge $u \xrightarrow{d} v \in H$, the length of a shortest path from u to v in G exceeds d. Let $G' = G \cup \{x \xrightarrow{c} y\}$ and $H' = H \setminus \{x \xrightarrow{c} y\}$. Find the set of all edges $u \xrightarrow{d} v$ in H' such that the length of a shortest path from u to v in G' does not exceed d.*

The preconditions relating the graphs G and H are a result of labelling and complete propagation. If all consequences of G are found and removed from H prior to every step, then no consequences of G are found in H and so the length of a shortest path from x to y in G exceeds the weight of any edge $u \xrightarrow{d} v \in H$.

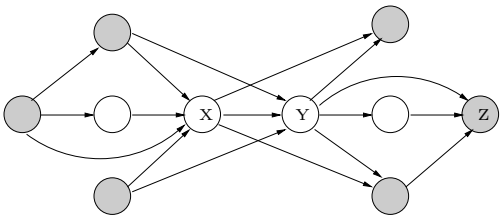


Figure 4.2: An example graph showing δ-relevant vertices with respect to the edge (x, y). For simplicity, all edges are assumed to have weight 1. The relevant vertices are white and the irrelevant vertices are shaded. As an example, the vertex z is not $\delta_x^{\rightarrow}$-relevant because there is a shortest path from x to z which does not pass through y. As a result, any constraint $u \xrightarrow{d} z \in H'$ is not member of the incremental complete propagation solution set.

As presented by Nieuwenhuis *et al.* [NO05], this problem may be reduced to solving two SSSP problems. First, for the graph G', the SSSP weights $\delta_y^{\rightarrow}$ from y are computed and then SSSP weights $\delta_x^{\leftarrow}$ to x are computed, the latter being accomplished simply by computing $\delta_x^{\rightarrow}$ in the reverse graph. Then for any constraint $u \xrightarrow{d} v \in H'$, the weight of the shortest path from u to v passing through $x \xrightarrow{c} y$ in G' is given in constant time by $\delta_x^{\leftarrow}(u) + c + \delta_y^{\rightarrow}(v)$. In accordance with Theorem 4.1.2, the weight of this path determines whether or not the constraint $u \xrightarrow{d} v$ is implied, in particular by the condition $\delta_x^{\leftarrow}(u) + c + \delta_y^{\rightarrow}(v) \leq d$. It then suffices to check every constraint in H' in this fashion. We now present several improvements to this methodology.

4.3.1 Completeness, Candidate Pruning, and Early Termination

A slight reformulation of the method above allows for *early termination* of the SSSP computations under certain conditions. That is, nodes for which it becomes clear that their minimal distance will not be improved due to the insertion of $x \xrightarrow{c} y$ to the constraint graph will not be explored. We introduce the idea of *relevancy* below to formalize how we can identify such vertices and give an example in Figure 4.2. For a new edge $x \xrightarrow{c} y$, relevancy is based on shortest path distances $\delta_x^{\rightarrow}$ (from x) and $\delta_y^{\leftarrow}$ (to y), in contrast to the formulation above. Under this new formulation, if the shortest path from u to v passes through $x \xrightarrow{c} y$, then the path length is $\delta_y^{\leftarrow}(u) + \delta_x^{\rightarrow}(v) - c$.

Definition 4.3.2 (δ-relevancy with respect to $x \xrightarrow{c} y$)**.** *A vertex z is $\delta_x^{\rightarrow}$-*relevant *if every shortest path from x to z passes through $x \xrightarrow{c} y$; similarly, a vertex z is $\delta_y^{\leftarrow}$-*relevant *if every shortest path from z to y passes through $x \xrightarrow{c} y$. A constraint $u \xrightarrow{d} v$ is δ-*relevant *if both u is $\delta_y^{\leftarrow}$-relevant and v is $\delta_x^{\rightarrow}$-relevant. A set C of constraints is δ-relevant if every $u \xrightarrow{d} v \in C$ is δ-relevant.*

Lemma 4.3.3. *The solution set for complete incremental difference constraint propagation is δ-relevant.*

Proof. Let $x \xrightarrow{c} y$ be the new edge in G, and suppose for a contradiction that some constraint $u \xrightarrow{d} v \in H'$ in the solution set is not δ-relevant. Then the length of a shortest path from u to v in G' does not exceed d. By definition of δ-relevancy, some path p from u to v in G' which does not pass through $x \xrightarrow{c} y$ is at least as short as the shortest path from u to v passing through $x \xrightarrow{c} y$. Observe that p is a path in G. By the problem definition, $u \xrightarrow{d} v \notin H$ and $H' \subset H$. Hence $u \xrightarrow{d} v \notin H'$, a contradiction. □

Corollary 4.3.4 (Early Termination)**.** *It suffices to check every δ-relevant member of H' for membership in the solution set. As a result, each SSSP algorithm computing $\delta \in \{\delta_x^{\rightarrow}, \delta_y^{\leftarrow}\}$ need only compute correct shortest path distances for δ-relevant vertices.*

Early termination is fairly easy to implement with most SSSP algorithms in the incremental constraint propagation context. First, for $\delta \in \{\delta_x^{\rightarrow}, \delta_y^{\leftarrow}\}$, we maintain a label for each vertex indicating whether or not it is δ-relevant. We then define an order $\prec$ over shortest path distances of vertices in a way that favors irrelevancy:

$$\delta(u) \prec \delta(v) \iff \delta(u) < \delta(v) \text{ or } \begin{cases} \delta(u) = \delta(v) \\ u \text{ is } \delta\text{-irrelevant} \\ v \text{ is } \delta\text{-relevant} \end{cases}$$

Since the new constraint $x \xrightarrow{c} y$ is a unique shortest path, we initially give y the label $\delta_x^{\rightarrow}$-relevant and x the label $\delta_y^{\leftarrow}$-relevant. During the SSSP computation of δ, whenever an edge $u \xrightarrow{d} v$ is found such that $\delta(u) \prec \delta(v) + d$, the distance to v is updated and we propagate u's δ-relevancy label to v. If at any point in time all such edges are not δ-relevant, then the algorithm may terminate.

To facilitate checking only δ-relevant constraints in H', one may adopt a trick described in [NO05] for checking only a reachable subset of H'. In

particular, one may maintain the constraint graph H' in an adjacency list form which allows iteration over incoming and outgoing edges for each vertex as well as finding the in- and out-degree of each vertex. If the sets of $\delta_x^{\rightarrow}$-relevant and $\delta_y^{\leftarrow}$-relevant vertices are maintained during the SSSP algorithm, the smaller of these two sets, measured by total in- or out-degree in H', may be used to iterate over a subset of constraints in H' which need to be checked.

4.3.2 Choice of Shortest Path Algorithm

There are many shortest path algorithms, and it is natural to ask which one is best suited to this context. An important observation is that whenever shortest paths $\delta_x^{\rightarrow}$ or $\delta_y^{\leftarrow}$ are computed, the graph G has been subject to a consistency check. Consistency checks establish a potential function π which can be used to speed up the shortest path computations a great deal. In particular, as was first observed by Johnson [Joh77], we can use $\pi(x)+c-\pi(y)$ as an alternate, non-negative edge weight for each edge $x \xrightarrow{c} y$. This weight is called the *reduced cost* of the edge. The weight w of path p from a to b under reduced costs is non-negative and the original weight of p, that is, without using reduced costs, is easily retrieved as $w + \pi(b) - \pi(a)$. Our implementation of constraint propagation exploits this property by using an algorithm for shortest paths on graphs with *non-negative edge weights*. The most common such algorithm is Dijkstra's [Dij59], which runs in $\mathcal{O}(m+n\log n)$ time. This is an improvement over algorithms which allow arbitrary edge weights, the best of which run in $\mathcal{O}(mn)$ time [CLRS90]. A direct result follows.

Theorem 4.3.5. *Complete incremental difference constraint propagation can be accomplished in $\mathcal{O}(m + n\log n + |H'|)$ time where m is the number of assigned constraints, n the number of variables occurring in assigned constraints, and H' is the set of unassigned constraints.*

Proof. The worst case execution time of finding all consequences over a sequence of calls to `assign` and `tprop`, is $\mathcal{O}(m + n\log n + |H'|)$ per call to `tprop` and $\mathcal{O}(m + n\log n)$ per call to `assign`. Thus if every call to `assign` is followed by a call to `tprop`, then the combined time for both calls is $\mathcal{O}(m + n\log n + |H'|)$. □

4.3.3 Adaptation of a Fast SSSP Algorithm

In order to fully exploit the use of the potential function in constraint propagation, we make use of a state-of-the-art SSSP algorithm for a graph with

non-negative edge weights. In particular, we implemented (our own interpretation of) Goldberg's smart-queue algorithm [Gol01]. The application of this algorithm to difference constraint propagation context is non-trivial because it makes use of a heuristic requiring that we keep track of some information for each vertex as the graph Π and its potential function changes. Even in the face of the extra book-keeping the algorithm turns out to run significantly faster than standard implementations of Dijkstra's algorithm with a Fibonacci heap or a binary priority queue.

The smart-queue algorithm is a priority queue based SSSP algorithm for a graph with non-negative edge weights which maintains a priority queue on vertices. Each vertex is prioritized according to the shortest known path from the source to it. The smart-queue algorithm also makes use of the *caliber heuristic* which maintains for each vertex the minimum weight of any edge leading to it. This weight is called the *caliber* of the vertex. After removing the minimum element of distance d from the priority queue, d serves as a lower bound on the distance to all remaining vertices. When scanning a vertex, we know the lower bound d, and if we come across a vertex v with caliber c_v and tentative distance d_v, we know that the distance d_v is exact if $d + c_v \geq d_v$. Vertices whose distance is known to be exact are not put in the priority queue, and may be removed from the priority queue prematurely if they are already there. The algorithm scans exact vertices greedily in depth first order. When no exact vertices are known it backs off to use the priority queue to determine a new lower bound. The priority queue is based on lazy radix sort, and allows for constant time removal of vertices. For full details, the reader is referred to [Gol01].

In this context, the caliber of a vertex may change whenever either the graph Π or its potential function changes. This in turn requires that the graph Π be calibrated before each call to `tprop`. Calibration may be accomplished with linear cost simply by traversing the graph Π once prior to each such call. However, we found that if, between calls to `tprop`, we keep track of those vertices whose potential changes as well as those vertices which have had an edge removed during backtracking, then we can reduced the cost of re-calibration. In particular, the re-calibration associated with each call to `tprop` can then be restricted to the subgraph which is reachable in one step from any such affected vertex.

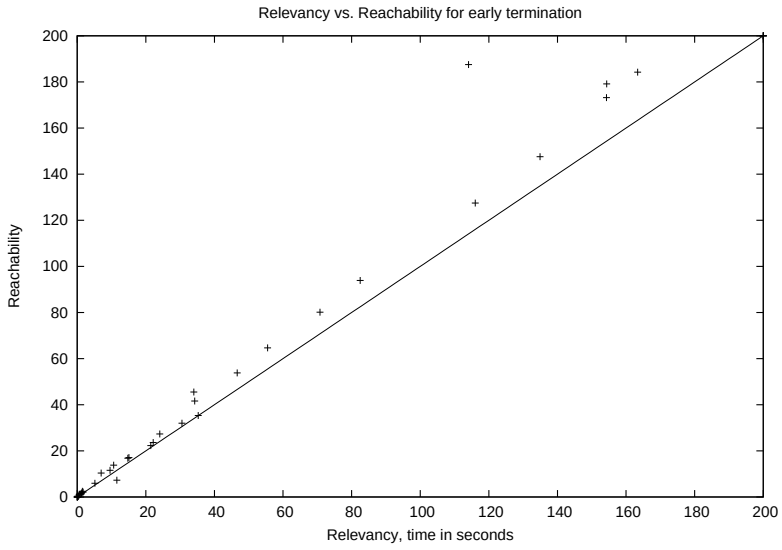
4.4 Experiments

We present various comparisons between different methods for difference constraint propagation. With one exception, the different methods are implemented in the same basic system: a Java implementation of flexible propagation which we call Jat. The underlying DPLL solver is fairly standard with two literal watching [MSS96], 1UIP clause learning and VSID+stack heuristics [GN02] as in the current version of ZChaff [MMZ$^+$01], and conflict clause minimization as in MiniSat [ES05]. Within this fixed framework, we implemented reachability-based and relevancy-based early termination. We also implemented the eager, exhaustive propagation strategy of DPLL(T) for comparison with our lazy, flexible propagation strategy. We present a comparison of reachability-based and relevancy-based early termination in Figure 4.3 as well as a comparison of lazy and eager strategies in Figure 4.4. These comparisons are performed on scheduling problems encoded as difference logic satisfiability problems on a 2.4GHz intel based box running Linux. The scheduling problems are taken from standard benchmarks, predominately from the SMT-LIB QF_RDL section [RT03]. In Figure 4.5, we also present a comparison of our best configuration, implemented in Java, against BarceLogicTools (BCLT) which is implemented in C and which, in 2005, won the SMT competition for difference logic.

Job-shop scheduling problems encoded as difference logic satisfiability problems, like the ones used in our experiments, are strongly numerically constrained and weakly propositionally constrained. These problems are hence a relatively pure measure of the efficiency of difference constraint propagation. While it seems that our approach outperforms the others on these types of problems[2], this is no longer the case when Boolean constraints play a stronger role. In fact, BCLT is several times faster than Jat on many of the other types of difference logic problems in SMT-LIB. Upon profiling Jat, we found that on all the non-scheduling problems in SMT-LIB, Jat was spending the vast majority of its time doing unit propagation, whereas in the scheduling problems Jat was spending the vast majority of its time doing difference constraint propagation. Although it is at best difficult to account for the difference in implementations and programming language, this suggests that the techniques for difference constraint propagation presented in this paper are efficient, in particular for problems in which numerical constraints play a strong role.

[2]Although we have few direct comparisons, this is suggested by the fact that BCLT did outperform the others in a recent contest on scheduling problems, and that our experiments indicate that our approach outperforms BCLT on the same problems.

Figure 4.3: A comparison of relevancy based early termination and reachability based early termination. The relevancy based early termination is consistently faster and the speed difference is roughly proportional to the difficulty of the underlying problem.



4.5 Conclusion

Our optimal difference logic engine incorporates efficient shortest paths algorithms in the framework of flexible propagation. Previously, exhaustive propagation in the DPLL(T) framework was the best known method for this class of problems. That work was based partly on the fact that consistency checks may be avoided as a result of exhaustive propagation. However, we demonstrated that in the case of difference logic, consistency checks in fact speed up propagation – both in theoretical worst-case terms and practically. Additionally, experiments demonstrated a significant performance advantage of using two-level flexible propagation to delay more expensive operations.

Figure 4.4: A comparison of laziness and eagerness in theory propagation for difference logic. Both lazy and eager implementations use relevancy based early termination and the same underlying SSSP algorithm. The lazy strategy is in general significantly faster than the eager strategy. This difference arises because the eager strategy performs constraint propagation more frequently.

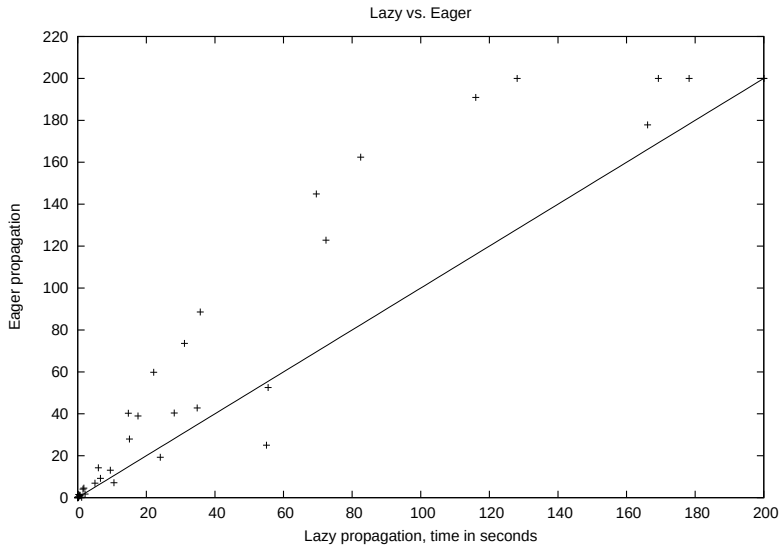
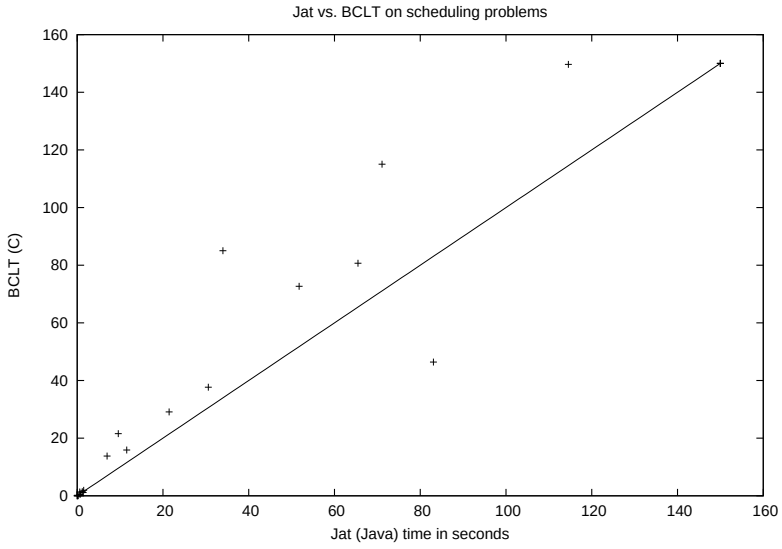


Figure 4.5: Jat with lazy propagation and relevancy based early termination compared with BarceLogicTools [NO05] on job-shop scheduling problems. The Jat propagation algorithm uses consistency checks and Goldberg's smart-queue SSSP algorithm as described in this paper, and is implemented in Java. Assuming BarceLogicTools hasn't changed since [NO05], it uses no consistency checks, eager propagation, a depth first search SSSP based $\mathcal{O}(mn)$ propagation algorithm, and is implemented in C. The Jat solver is generally faster.



Chapter 5

On Model Search with Linear Arithmetic

In this chapter we present a case study of the extent to which various properties of DPLL search are applicable to direct model search for infinite domain problems, in particular with respect to the UCS algorithm. The object of this study is the domain of linear arithmetic, which is sufficiently rich to offer a range of problematics and widely applicable enough to be well worth study in its own right.

This chapter is organized as follows. Section 5.1 defines the problems we wish to solve and discusses related work. Section 5.2 outlines the major gaps between UCS and DPLL in order to establish a focus on these issues. Sections 5.3 presents resolution. Section 5.4 considers branch selection and backtracking, which are tied together for technical reasons. Section 5.5 presents an implementation of UCS for continuous linear arithmetic, including backtrack-friendly lazy mechanisms for consistency checking and evaluation. Section 5.6 presents a number of experiments on a limited set of problems. Section 5.7 concludes.

5.1 Problem Statement

UCS addresses problems presented in conjunctive normal form. For linear arithmetic, such problems may look like

$$
\begin{array}{l}
((2x - 7y \leq 43) \vee (2x + 7y + z > 42) \vee (x > 9)) \\
\wedge \\
((2x - 7y \leq 41) \vee (2x + 7y + z > 49) \vee (z \leq 0)) \\
\wedge \\
\ldots \\
((x < 0 \vee x > 0))
\end{array}
$$

This is strictly a subset of what is handled by most SMT linear arithmetic solvers such as Yices and Z3, since there are no propositional variables. However, one can code propositional variables v as linear predicates by creating an associated numeric variable v_n and introducing new satisfiable predicates $p_v(v_n), q_v(v_n)$ such that $p_v \wedge q_v$ is unsatisfiable; taking p_v for the literal v and q_v for the literal $\neg v$, the resulting formula is equisatisfiable to the original. However, SMT solvers usually treat propositional variables in a more optimized fashion as they are treated in modern DPLL solvers. Thus we would expect different behavior resulting from this encoding. At the same time, working with this restricted format helps focus our attention on instantiating UCS for linear arithmetic rather than on how to combine different types of variables, which is a topic of future work.

More formally, we will assume that our problem format is a conjunction $\bigwedge D$ where each $d \in D$ is a disjunction $\bigvee P$ and each $p \in P$ is a linear predicate. For convenience, we also assume that each linear predicate p is either in the form $\Sigma_i a_i x_i \leq b$ or the form $\Sigma_i a_i x_i < b$ and moreover that all the coefficients a_i and b are integers which share no common divisor. In Section 5.5.1, we describe how we normalize predicates to this form using a standard method.

5.1.1 Related Work

A model search algorithm (GDPLL-QFLRA) for the class of problems defined above was presented in [MKS09]. UCS may be viewed as a variant of GDPLL-QFLRA with dynamic variable orderings and clause forgetting. While the formal UCS algorithm achieves termination with dynamic variable orderings by tree-like resolution, we experiment primarily with dynamic variable orderings with clause recording, which is incomplete. In our experience

this usually performs better than the complete formal algorithm. Nonetheless, UCS provides one basis on which dynamic variable orderings may be applied without sacrificing termination.

Most solvers capable of dealing with this class of problems fall in the category of lazy SMT solvers. Most of these, in turn, follow the architecture spelled out in [DdM06], which uses an incremental simplex method together with preprocessing and bounds propagation. This general framework has proven more effective than most and some of the available solvers are highly optimized. The method reported in [KV05] performs on-the-fly Fourier-Motzkin elimination of an interpreted literal with respect to the uninterpreted literals. This work is closer to ours in that theory deductions are performed and also in that to a limited extent their method allows search over models.

5.2 Differences between UCS and DPLL

In this section we discuss the differences between UCS and DPLL. We invite the reader to recall Section 3.3, in particular with respect to Algorithm 3.3.1 and the requirements for the procedures `select()`, `isUC()` and `resolve()`.

Space Efficiency

Perhaps the most important aspect of DPLL based solvers in comparison to other methods in propositional reasoning is the fact that such solvers can be space efficient. While solvers often do make use of a lot of space, they have the capacity to operate with very limited space. This allows one to control the amount of used space in an effort to find solutions quickly, for example by effective clause garbage collection. The unate consistent search algorithm provides a hint of how one may accomplish space efficiency in a given instantiation but it provides no such guarantee. In particular, Theorem 3.3.4 demonstrates that the algorithm makes progress even though it forgets learned clauses. Nonetheless UCS provides no bounds on the number *or size* of clauses. We present cubic space bounds in the number of variables for the worst case minimal number of coefficients necessary to represent clauses and guarantee progress. This is accomplished by a cooperation between resolution and branching. We note that unlike in propositional logic, where small clauses are stronger than large ones, our space bounds are derived in part by making learned clauses smaller in a fashion which *weakens* them. This highlights that space efficiency does not readily generalize from DPLL to UCS.

Learning and Branching

Modern DPLL solvers benefit largely not only from learning clauses, but also from the careful construction of learned clauses. In particular, conflict analysis attempts to find a succinct reason behind a dead end in the search. We thus would like our solver to clearly identify a minimal set of resolution steps necessary to derive such a succinct clause. In Section 5.3.2, we present a conflict analysis mechanism which efficiently produces a minimal number of clause resolution steps to derive learned clauses. However, in the UCS algorithm, clause resolution steps are abstracted away and are determined on the fly during backtracking. With DPLL solvers, clause resolution steps correspond to variable resolution steps and a global implication graph is available for exploiting topological properties (such as 1UIP and self-subsumption minimal clauses). It is not yet clear how such properties may generalize to the case of UCS algorithms, as the issue of global topological properties is complicated by the fact that more than one clause may play a role in constraining a variable.

More importantly, DPLL solvers can make use of unconstrained resolution without sacrificing termination because only a finite number of clauses can be derived. In the case of linear arithmetic, termination requires some form of constraint on resolution. In GDPLL-QFLRA, resolution is directed and so termination is guaranteed. In UCS, if the resolution is regular, then termination is guaranteed, but the most straightforward ways of accomplishing this are by using a fixed variable ordering or tree-like resolution, both of which are severely restrictive.

Restricting resolution in these ways complicates the question of the role of variable selection. Nonetheless, variable selection is meaningful in the context of an incomplete solver (with clause recording) or in the context of a complete solver which forgets clauses (with treelike resolution).

Lazy Evaluation and Consistency Checking

The ideas behind backtrack-friendly data structures such as two literal watching may be extended to the UCS context. Section 5.5.4 presents a method for performing numerical evaluation lazily without updating the clause or predicate databases during backtracking. Section 5.5.2 presents a method for maintaining unate consistency inspired by two literal watching.

Control Flow

The control flow of UCS differs from DPLL primarily in that it is necessary to interleave consistency checks with backtracking and resolution. We illustrate the problem by considering a point in `UC-Search` at which a learned clause w containing some variables is derived. Since w is false under the assignment at this point, there is some variable $x \in vars(w)$ which is assigned deepest in the search, and there is a corresponding assignment $\alpha \cup \{x \mapsto a\}$ passed as an argument to `UC-Search`. Referring to Algorithm 3.3.1 (page 43), this call occurs at line 5. The subsequent call (line 9) takes the form `UC-Search`$(\phi \wedge w, \alpha)$. This in turn induces a unate consistency check on $(\phi \wedge w)[\alpha]$, which may or may not succeed. If it does not succeed, a new clause will be derived and the backtrack sequence will continue. This variable-wise daisy-chaining of resolution, backtracking and consistency checks degenerates in the propositional case to a search over the implication graph; allowing the procedure to unassign variables after the final learned clause is derived. In `UC-Search`, this mechanism is more complicated.

5.3 Resolution

The most substantive challenge to instantiating linear arithmetic for UCS is the design of the procedure `resolve()`. Resolution occurs under the following conditions

1. There is a working set of clauses $\phi \equiv \bigwedge D$.

2. There is a partial variable assignment α.

3. There is a variable $x \in vars(\phi[\alpha])$ such that $\phi[\alpha]|_x$ is unsatisfiable.

We are charged with producing a clause w such that

1. $\phi \models w$

2. $w[\alpha]$ contains only constants and evaluates to false.

5.3.1 Unate Resolution

To address this problem, consider an unsatisfiable set of unate clauses about the variable x. For example,

$$\begin{array}{ll} & (x > 1) \vee (x > 2) \\ \wedge & (x < 9) \vee (x < 0) \\ \wedge & (x < 0) \vee (x > 3) \\ \wedge & (x < 2) \vee (x > 5) \\ \wedge & (x < 4) \vee (x > 6) \\ \wedge & (x < 5) \vee (x > 8) \\ \wedge & (x < 7) \vee (x > 10) \end{array}$$

In the example, each clause d excludes an interval $[l_d..u_d]$ from the set of feasible assignments for x, even though the clause may contain several upper or lower bounded predicates. In particular, one may simplify any clause d containing only strict predicates to the following form

$$\begin{array}{rcl} d & \equiv & x < \max(\{u \mid (x < u) \in d\} \cup \{-\infty\}) \\ & & \vee \\ & & x > \min(\{u \mid (x > u) \in d\} \cup \{\infty\}) \end{array}$$

where $x > \infty$ and $x < -\infty$ are taken to be false. More generally, for mixed strict and non-strict comparison, one can disjoin a weakest representative of all the lower bounds (or just false if there are no lower bounds) with a weakest representative of all the upper bounds (or just false if there are no upper bounds).

Once clauses have been so normalized, it is useful to identify a minimal subset of them which eliminates the variable x. Towards that end, consider the following

Proposition 5.3.1. *Minimal Unate Unsatisfiability*
Let $C \equiv \bigwedge D$ be a set of unate clauses. Then C is unsatisfiable if and only if there is a chain of non-trivial clauses

$$d_1, d_2, \ldots d_n$$

such that

1. $n \geq 2$
2. d_1 contains no lower bound predicates.
3. d_n contains no upper bound predicates.

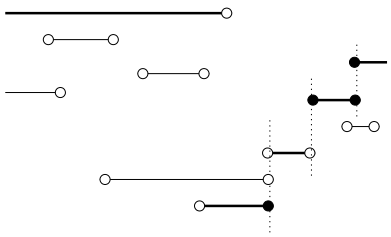


Figure 5.1: A pictorial representation of an unsatisfiable set of unate clauses. Each horizontal line represents a clause as the interval forbidden by a clause. Clauses which do not forbid an interval are considered trivial and are not represented. The endpoints of the lines have the following meanings. If there is no endpoint on the left (resp. right) then the clause has no upper bound (resp. lower bound). If an endpoint is filled, then the bound is strict and the forbidden interval includes the endpoint. If an endpoint is not filled, then the bound is not strict and the forbidden interval does not include the endpoint. The set of bold lines represents a (in this case unique) minimal set of unsatisfiable clauses.

4. The weakest upper bound of d_i is incompatible with the weakest lower bound of d_{i+1} for $1 \leq i < n$.

Moreover, the set $\{d_1, d_2, \ldots, d_n\}$ is minimal if the weakest upper bound of d_{i+1} is stronger than the weakest upper bound of d_i for $1 \leq i < n$ and the weakest lower bound of d_{i+1} is not incompatible with the weakest upper bound of d_{i-1}.

Rather than give a formal proof of Proposition 5.3.1, we refer the reader to Figure 5.1.

5.3.2 Resolution with Predicates

The task of deriving a resolvent for predicates which have multiple variables may be divided into 3 components: transitivity, pairs of clauses, and eliminating a variable from a chain of clauses.

Transitivity

Consider the following proof rule

$$\frac{f < x, x < g}{f < g} \ (T)$$

The rule simply enforces transitivity of $<$. One can easily generalize the rule to incorporate mixed strict and non-strict predicates: if both predicates are non-strict, then the result is non-strict and otherwise the result is strict. This proof rule forms the foundation of Fourier-Motzkin elimination, a well known doubly exponential method for eliminating a set of variables from a conjunction of linear predicates.

Referring again to the unate case, one always derives a comparison of constants, which always evaluate to true or false. For example

$$\frac{3 < x, x < 2}{3 < 2}$$

This provides a basis for which we can derive predicates from pairs of incompatible predicates which are weakest in each clause. To accommodate the non-unate case, we simply think of the constants from the unate case as functions which happen to give rise to inconsistent constants under the current assignment. For example, let $\alpha = \{x \mapsto 0\}$ and consider the conjunction

$$(x - y < -3) \wedge (x + y < 1)$$

Under the assignment, this simplifies to $y > 3 \wedge y < 1$. Centering the predicates above around y gives

$$\frac{x + 3 < y, y < 1 - x}{x + 3 < 1 - x}$$

which simplifies to $x < -1$.

Resolving 2 Clauses

Let l, u be two clauses, let x be a variable and α a partial assignment. Suppose that $l[\alpha]$ and $u[\alpha]$ have no true predicates, are unate about x, and that a weakest lower bound on x in $l[\alpha]$ is incompatible with a weakest upper bound on x in $u[\alpha]$, as in Proposition 5.3.1. As in GDPLL-QFLRA [MKS09], one may apply the transitivity rule to all pairs of x-lower bounds in l and x-upper bounds in u, resulting in a set of predicates T_x. Let l' be the subclause of l which contains no x-lower bounds and u' be the subclause of u which contains no x-upper bounds. It is straightforward to derive a clause resolution rule which concludes $\bigvee T_x \vee l' \vee u'$ from l and u, which we denote by $l \otimes_x u$. This leads to the following

Theorem 5.3.2. *Correctness of Clause Resolution*
Let l, u be two clauses, and x a variable such that l contains a lower bound

on x while u contains an upper bound on x. Then

$$l \wedge u \models l \otimes_x u$$

Proof. Take the DNF of $l \wedge u$, apply transitivity to all terms containing mixed upper and lower bounds on x, translate back to CNF. □

Example 5.3.3. *Clause Resolution*
Consider the clauses

$$\begin{array}{rcl} l & \stackrel{\circ}{=} & (x > 4) \vee (x < 17) \\ u & \stackrel{\circ}{=} & (x < y) \vee (x < z) \vee (x > 32) \vee (z < 0) \end{array}$$

Then $l \otimes_x u$ is

$$(y > 4) \vee (z > 4) \vee (x < 17) \vee (x > 32) \vee (z < 0)$$

Eliminating a variable

Eliminating a variable x occurs when there is a set of unate clauses $\phi[\alpha]|_x$ which is unsatisfiable. One can think of $\phi[\alpha]|_x$ as evaluation under the assignment α, where this evaluation is a sort of a *mask* over the real set of clauses which are unsatisfiable. Namely, suppose $\phi[\alpha]|_x = \bigwedge D$. Each $d \in D$ corresponds to a clause d' occurring in ϕ which is unate around x under α. Note that we can assume that $d'[\alpha]$ contains no true predicates; otherwise it would not contribute to the unsatisfiability of $\phi[\alpha]|_x$.

Proposition 5.3.1 identifies a minimal set of unsatisfiable unate clauses in the form of a chain

$$(c_1, c_2, \ldots c_k)$$

of mutually incompatible forbidden intervals such that c_1 contains only upper bounds over x and c_k contains only lower bounds over x To eliminate a variable we simply resolve along this chain as follows. Let $r_1 \stackrel{\circ}{=} c_1' \otimes_x c_2'$ and $r_i \stackrel{\circ}{=} r_{i-1} \otimes_x c_{i+1}'$ for $1 < i < k$. We then claim the following theorem.

Theorem 5.3.4. *Sufficiency of Elimination for Progress*
Let ϕ, α, x be such that $\phi[\alpha]|_x$ is unsatisfiable. Let $(c_1, c_2, \ldots, c_k)$ be the minimal chain of unsatisfiable unate clauses defined in Proposition 5.3.1 and let r_i be defined as above for $1 \leq i < k$. Then r_{k-1} is sufficient for progress (Theorem 3.3.4) .

Proof. (sketch) We need to show that

1. $\phi \models r_{k-1}$.

2. $r_{k-1}[\alpha]$ is variable free and evaluates to false.

The first item follows from Theorem 5.3.2 and the fact that r_{k-1} is the resolvent of clauses found in ϕ.

To show that $r_{k-1}[\alpha]$ is variable free, we first observe that $vars(c_i'[\alpha]) = \{x\}, 1 \leq i \leq k$ since the c_i are unate about x by assumption. Since resolution does not introduce variables, we need only to show that $x \notin vars(r_{k-1})$. We consider the chain $c_1 \ldots c_k$ and observe that since c_1 contains no lower bounds on x, r_1 contains no lower bounds on x. The same holds true for all r_i by induction. Hence we need only show that r_{k-1} contains no upper bounds on x. Since c_k contains no upper bounds, the final resolution step eliminates x. Thus r_{k-1} is variable free.

To show that r_{k-1} evaluates to false under α, it suffices to consider case by case all the possible predicates which are added to r_{k-1} in the definition of resolution. □

Observe that variable elimination is *specific* in the sense of Definition 3.3.5, namely resolution only takes into account x-unate clauses in eliminating x. We also want to establish that elimination has finite width. Towards that end, consider that the set of possible unsatisfiable chains of clauses, as in Proposition 5.3.1 is finite, and so we can establish finite width by showing that clause resolution is finite. Since the transitivity rule has exactly one result for every pair of x-predicates of opposite sign for x, the number of predicates that can be generated in one step is finite. We can then conclude that variable elimination is a sufficient implementation of `resolve` for termination, as in Theorem 3.3.9.

5.3.3 Space Efficiency

Resolution leads to the possibility of clause growth. To address this issue, we consider a simple notion of clause normalization. Consider two linear predicates $f < a, f < b$ where a, b are constants such that $a < b$. Then the disjunction $(f < a) \vee (f < b)$ is equivalent to $(f < b)$. Assuming clauses are simplified according to this observation, the only way that clauses can grow beyond some bound n is when there are more than n predicates $f < a$, each with a distinct f. As a result, some classes of linear constraints cannot grow exponentially. For example, in a problem with simple unate bounds $x < a$ or $x > a$, there will be at most two predicates over each variable in a clause. Similarly, in a difference logic problem, no clause will contain more than the square of the number of variables. However, it is straightforward to create a problem with arbitrary linear constraints which may induce exponential

clause growth. In situations where clause growth becomes a problem, it may be advantageous to consider a method for deriving smaller clauses. To this end, we present the notion of weakening.

Weakness

For generating a proof that a predicate is a weakest predicate in a clause, one may begin with the unate case and then generalize it by substituting constants with functions. Consider the following proof rules

$$\frac{x < a \vee x < b}{x < a \vee a < b}\ (wk_<) \quad \frac{x > a \vee x > b}{x > a \vee a > b}\ (wk_>)$$

For example in the unate case, consider the clause

$$x < 3 \vee x < 5$$

From this clause we can derive

$$x < 5 \vee 5 < 3$$

or we can derive

$$x < 3 \vee 3 < 5$$

The former simplifies to $x < 5$ while the later simplifies to true.

To extend the weakness rules to address mixed strict and non-strict bounds, observe that the rules are essentially stating "either predicate $p(x)$ is true or it is not the weakest predicate on x in the disjunction". In applying these rules, the strictness of p carries to the conclusion, but strict comparison is always used to say that p is not the weakest predicate. For example,

$$\frac{x \leq f \vee x \leq g}{x \leq f \vee f < g}$$

or likewise

$$\frac{x < f \vee x \leq g}{x < f \vee f < g}$$

Given a clause l, an assignment α, and a variable x such that $l[\alpha]$ is unate about x and contains a lower bound on x, we write $wk_>(l, \alpha)$ to denote the clause l' resulting from all possible applications of the rule $wk_>$ about x, where a weakest predicate is selected with respect α. Similarly, we write $wk_<(u, \alpha)$ to denote the clause u' resulting from all possible applications of the rule $wk_<$ about x to the clause u where a weakest predicate is selected with respect to α.

Example 5.3.5. *Applying Weakness*
Consider the clause

$$w \triangleq^{\circ} (x < 0) \vee (x < 3) \vee (x < y) \vee (x > 7) \vee (z > f)$$

and the assignment $\alpha \triangleq^{\circ} \{y \mapsto 5\}$ then

$$wk_{<}(w, \alpha) = (x < y) \vee (y < 3) \vee (y < 0) \vee (x > 7) \vee (z > f)$$

and

$$wk_{>}(w, \alpha) = w$$

One can apply the weakness rules one variable at a time to learned clauses with respect to the current assignment. Once the rules have been applied with respect to one variable x, there are at most two associated predicates h_x, l_x which still contain that variable. It is then possible to apply the weakness rules to all the predicates in the clause *except* h_x, l_x with respect to a new variable. This process may continue recursively on smaller and smaller subclauses until there are at most two predicates attributed to each variable.

As a result, we have a tool at our disposal which can limit the size of learned clauses so that they contain at most twice as many predicates as variables. Note that this still does not address the issue of the total *number* of learned clauses, which can also grow exponentially in the UCS framework despite the fact that it forgets learned clauses. Section 5.4 addresses this issue.

Generally, there is a time-space tradeoff involved in applying the weakness rules, since the rules do in fact weaken clauses. As a result, we expect that weakening should be used sparingly in any implementation.

5.3.4 Strict Bounds Again

Working with strict bounds with continuous variable domains is problematic for branching on extreme points. In [DdM06], this issue is addressed by extending the domain of the variables from rationals to pairs of rationals (v, e), where e is the coefficient of a symbolic positive infinitesimal ϵ. One can define ordering over this extended domain in the natural way, where e always is less important than v. Also, one can define addition $(v, e) + (v', e') \triangleq^{\circ} (v + v', e + e')$ and likewise multiplication by a *rational* $r(v, e) \triangleq^{\circ} (rv, re)$. With this extended domain semantics, it is straightforward to evaluate a linear expression and then compare the result to a constant. Consider also

that $f < g$ is equisatisfiable to $f + \epsilon \leq g$ for some positive ϵ. This property lifts to systems of inequalities and further to arbitrary Boolean combinations by translation to DNF. Hence, it is also possible to translate a formula to an equisatisfiable formula without strict inequalities.

In [DdM06], these two observations are put together to allow simplex pivots on extreme points without the need to keep the variable ϵ explicit in the simplex tableau. One may similarly consider an extended variable domain semantics together with non-strict inequalities which may contain the special variable ϵ in a UCS instantiation for continuous linear arithmetic. This idea is attractive because it can be used to guarantee that the clauses which constrain a variable chain together. However, one needs to re-consider the transitivity and weakness proof rules in light of this representation. In particular, the transitivity rule no longer distinguishes between strict and non-strict predicates and instead computes the coefficient of ϵ in the conclusion in the same manner one computes the coefficient of any other variable. As presented above, the weakness rules explicitly introduce strict predicates. Rather than introducing strict predicates, we have these rules increment the ϵ coefficient. This representation is only necessary for branching on corners, and branching on corners in turn is only necessary to guarantee that the mechanism we describe to limit the number of learned clauses per variable works without any a priori numeric analysis.

5.4 Branching, Backtracking, Learning, and Termination

The selection of a variable and its associated value may be considered as a *branch*, the intuition being that the underlying search tree has a potentially infinite number of branches. Branching selection is of course very important in terms of heuristics. However, the problem of branching is also related to the problem of completeness. Recall that Theorem 3.3.9 (page 48) demands that the process be exhaustively asserting. In addition, a subtle point about branching and the property of exhaustive assertiveness arises when we consider a process that backtracks more deeply than indicated in the formal UCS algorithm. In particular, suppose that upon learning a clause $w \vee p$, we backtrack to the smallest assignment α under which $w[\alpha]$ is false and $p[\alpha]$ is x-unate. In this case, the practice of always choosing x for the next branch resembles exhaustive assertiveness. However, with this type of assertion-level backtracking, it is possible to derive another clause $w \vee q$ where $q[\alpha]$ is y-unate for some variable y distinct from x. If we choose y at this point, then

the process may use $w \vee p$ as an antecedent of subsequent learned clauses and similarly if we choose x the process may use $w \vee q$ as an antecedent of subsequent learned clauses. Both cases introduce the possibility of infinite chains of deduction.

Consider a branching mechanism which allows assertion level backtracking whenever possible and slowly builds up fixed-order variable selection chains whenever the above situation occurs. In particular, we can ensure that upon backtracking to α, we always select x and whenever we arrive at $\alpha \cup \{x \mapsto a\}$ we always select y. This can be easily implemented by associating an exhausting variable with each branch depth. Whenever the process selects a variable, it first checks to see whether there is an exhausting variable associated with the current branch depth. If so, the exhausting variable is selected and if not a dynamic variable choice may be made. Whenever the process unassigns a variable at a given depth, the exhausting variable associated with that depth becomes nullified. To compute a deeper-than-natural backtrack level, first compute the asserting backtrack depth and then follow any chain of exhausting variables to the end, yielding an extended asserting backtrack level. This mechanism retains the necessary properties for restricted resolution (Theorem 3.3.7), and so termination follows if the process forgets clauses.

The problem of restricting resolution to a bounded set while retaining progress is difficult. There are many possible methods of accomplishing this, often requiring dedicated implementations while there is no guarantee the result will prove worthy. It is possible, for example, to retain progress while bounding the derivation height of the proof graph. It is also possible to derive a fixed variable order during the solution process, for example by dampening the effect of VSIDs over time.

Limiting the Number of Learned Clauses

Consider what happens when a clause w is recorded after a maximal backtrack sequence. The clause w excludes some portion of the feasible space for some variable x. We branch with the variable x on the corner of the maximal forbidden interval containing the space forbidden by w. Denote by I_w this interval. Consider a subsequent learned clause w' which also constrains x. Since w' excludes a point which is adjacent to I_w, the resulting maximal forbidden interval $I_{w'}$ contains I_w. As a result, we can *replace w and w'* with the resolvent associated with a chain of clauses excluding $I_{w'}$. Using this mechanism, it is possible to retain only one learned clause per assigned variable. Taken together with weakening rules described in Section 5.3.3,

we arrive at cubic space bounds for the total number of coefficients required to maintain progress.

5.5 Implementing UCS

We describe here the implementation of our UCS LA solver used in the experiments below. There is a wide space of possibilities for the design of basic data structures and algorithms used in this context. We have certainly only explored and experimented with a small fragment of the available options.

Our solver has an expression graph front end which allows one to produce Boolean combinations of linear predicates and propositional variables via a programming interface. These expression graphs are translated by encoding Boolean values such as the propositional variables and Tseitinization variables as predicates over continuous variables. Given an expression graph with a root expression, one can create a solver and ask it to solve the formula. Our solver cannot treat propositional variables in the way that a DPLL solver would. This is impractical but allows for much simpler implementation and for a more focused study of the problems involved in generalizing DPLL.

5.5.1 Representing Predicates and Clauses

Predicate Representation

We maintain predicates in the form

$$\Sigma_i a_i x_i + b + c\epsilon \leq 0$$

where a_i, b, and c are integer coefficients. Rather than distinguish b and c, we consider a predicate simply as a coefficient vector $(d_1 d_2 \ldots d_{n+2})$ where $d_i = a_i$ for $1 \leq i \leq n$, $d_{i+1} = b$ and $d_{i+2} = c$.

One can easily normalize predicates with rational coefficients to this form. In particular, suppose there are n variables and the coefficients are given as a vector $(\frac{p_1}{q_1} \frac{p_2}{q_2} \ldots \frac{p_{n+2}}{q_{n+2}})$ Let $m = \text{lcm}(\{q_i \mid i \in [1..n+2]\})$, the least common multiple of all the denominators of the coefficients and constants. Then multiply all the coefficients by m and observe that the resulting predicate is equivalent to the original and each $\frac{mp_i}{q_i}$ is an integer n_i. Finally, compute the greatest common divisor d of all n_i and divide all n_i by d. The resulting predicate only uses integer constants and is unique. Of course, one can treat predicates over arbitrary expressions $f \leq g$ in the form $f - g \leq 0$.

The coefficient list is maintained in sparse form, keeping only non-zero coefficients. The list is not pre-sorted but the constant and epsilon coefficients are always kept at the end to ease the task of scanning for unassigned variables a bit. The coefficient list takes the form of an array and allows swapping of coefficient positions to aid the lazy evaluation mechanism described below in Section 5.5.4.

In addition, each predicate is attributed with a valuation summary in the form of a pair of rational numbers representing the value of a variable together with an ϵ coefficient as described above in Section 5.3.4. The valuation summary holds the result of lazy evaluation, which only occurs when the predicate has no unassigned variables or exactly one unassigned variable. If the predicate has no unassigned variables, then a comparison of the valuation summary with 0 tells whether the predicate is true or false. If the predicate has one unassigned variable, the valuation summary is used to represent the bound induced on the free variable, and the coefficient of the free variable is swapped with the coefficient in the initial position so that the free variable's coefficient is always in first position. With this mechanism, one can always find the direction of the bound by the sign of the first coefficient and the constant of the bound in the evaluation summary.

We use a simple hashing and reference-counting mechanism to ensure that we only ever create one copy of any given predicate. This turns out to be important with regards to the lazy evaluation mechanism.

Clause normalization

As noted in Section 5.3.3, we normalize clauses as follows. For any predicates in the form $f + b + c\epsilon \leq 0$, we may keep only the weakest such predicate over f in a given clause. We found that this mechanism reduced the size of learned clauses and solving time significantly.

5.5.2 Consistency Checking

Variable consistency checking, *i.e.* an implementation of `isUC()` plays an important role in the `UC-Search` algorithm because it occurs very frequently. In `UC-Search`, the method `isUC()` is called initially and in response to variable assignments as well as in response to clause learning. Each call to the method checks the unate consistency of all free variables. Of course, one may simply consider the constraints placed on each variable x independently. Over the course of a `UC-Search` run, the constraints over a variable x are asserted incrementally and may be subsequently un-asserted if the procedure

backtracks or forgets a clause.

Thus consistency checking begs a per-variable incremental and backtrack friendly implementation. Section 5.5.4 describes data structures centered around identifying when clauses become unate, free of true predicates, and non-trivial on the fly. Accordingly, we will assume that consistency checks occur upon the assertion of one non-trivial x-unate clause at a time, for every variable x. More particularly, we consider a sequence of clause assertions $c_{x,1}c_{x,2}\ldots c_{x,k}$ where each clause $c_{x,i}$ is a non-trivial x-unate clause. A convenient means to maintain consistency for x incrementally is to under-approximate the feasible set with lower and upper bounds l_x, u_x such that

$$l_x \wedge u_x \models \bigwedge_{1\leq i\leq k} c_i$$

It is possible to maintain such an under-approximation with constant time updates to the values l_x, u_x upon assertion of an x-unate clause c as follows. Initially, we let $l_x = u_x = \top$. Let $l_x(c)$ and $u_x(c)$ denote the weakest lower and upper bounds for x found in c, defaulting to $\bot$ in the case that there is no respective bound in c. After assertion of c, the next state l'_x, u'_x of the under-approximation may be computed as

$$(l'_x, u'_x) \doteq \begin{cases} (l_x, u_x) & \text{if} & l_x \wedge u_x \models c \\ (l_x(c), u_x) & \text{else if} & u_x(c) = \bot \\ (l_x, u_x(c)) & \text{else if} & l_x(c) = \bot \\ (l_x(c), u_x) & \text{else if} & l_x(c) \models l_x \\ (l_x, u_x(c)) & \text{else if} & u_x(c) \models u_x \\ (\bot, \bot) & \text{otherwise} & \end{cases}$$

If $l'_x \wedge u'_x$ is satisfiable, the under-approximation may defer a real consistency check until a new x-unate constraint is asserted. Otherwise, a real consistency check needs to take place with respect to the current set of clauses to find whether there is a chain of unsatisfiable clauses as described in Proposition 5.3.1. In the event that there is, resolution takes place. Otherwise, a new under-approximation needs to be computed.

While it is certainly possible to do consistency checking without an under-approximation, it would necessitate maintaining a sorted structure of x-unate clauses for every x on every clause assertion or un-assertion. Since `UC-Search` is a depth-first backtracking algorithm, an under-approximation based mechanism can filter out a lot of the potential work required to maintain such a sorted structure. Our initial experience indicates that this under-approximation mechanism is efficient in the sense that profiling always indicated that very little time overall was spent in consistency checks.

5.5.3 Value Selection

Value selection offers a richer array of possibilities in the case of numeric variables than in the case of propositional variables. We identify and compare various value selection strategies which are based on the idea of branching on corners of forbidden intervals. We break down value selection according to when a corner was identified. For each time that a corner may be selected, there is a potential degree of freedom as to whether we select upper or lower extreme points of forbidden intervals. We associate with each variable a *bias* which determines whether an upper or lower extreme point is preferred. We examine the practice of bias toggling at different points in the solution process. We have not yet considered different constant selection strategies for the case where a variable is unconstrained.

Selection Time

In our experiments, we examine the following timings of corner selection.

1. Current corners. A branch value is selected on some corner of an interval under the current partial assignment, if there is a forbidden interval for the variable.

2. Recent corners. A branch value is selected based on the consistency checking mechanism of Section 5.5.2. One may choose a value based on the under-approximation of the feasible set. This allows finding values which are often current corners and sometimes cached partial solutions without the overhead of finding a current corner.

3. Last corners. Since the consistency checking mechanism uses under-approximations, it may exclude the last assignment value. It is well known in the propositional case that branching on last values can be helpful.

Bias Toggling

For each possible timing of corner selection, we vary bias toggling according to the following.

1. Fixed Bias. The biases toward upper or lower corners are set to a fixed value initially, according to the parity of an integer identifier associated with each variable.

2. Assertion Bias Toggling. Upon learning a new asserting clause, the constrained variable's bias is toggled. All other biases are fixed.

3. Assignment Bias Toggling. Prior to every assignment, the bias of the variable towards upper or lower values is toggled.

5.5.4 Lazy data structures

Lazy Evaluation

A combination of a fairly straightforward and small extension of 2LW with predicate evaluation caching gives a backtrack-friendly lazy evaluation mechanism. In particular, a 2LW mechanism determines when a clause becomes unate as follows. Every variable maintains a list of watched clauses and every clause is watched by two variables. Upon assignment of variable $x \mapsto a$, the watched clauses associated with x are scanned. If some clause c has the other watched variable assigned, then we know that $x \mapsto a$ is unate consistent since a was chosen from a unate consistent set. Otherwise, the other watched variable x' of clause c is unassigned. In this case, we need to find out if c is x'-unate. In this case, we scan the predicates in c to see if there is another unassigned variable y. If there is one, then we remove c from the watchlist of x and add c to the watchlist of y.

If there is no other unassigned variable, then the clause is x'-unate, but may contain a true predicate and we perform numerical evaluation. Predicates are scanned again to see whether the solver has backtracked over the variable in the predicate which is assigned deepest in the search tree. If not, we use that evaluation summary rather than re-evaluating the predicate. Otherwise, the predicate needs to be evaluated. Evaluation is the most time consuming activity in our solver. The check for the deepest assigned variable occurs in constant time because on every predicate evaluation we place the deepest assigned variable first in the list of terms appearing in the predicate.

If a predicate p does not contain x then it must be either true or false. If p is true, then we find the most recently assigned variable z in p and add c to the watchlist of z and remove c from the watchlist of x. This guarantees that c will be scanned upon any assignment which might make it unate after z is unassigned. Until that point, c will never be unate because it contains a true predicate.

If a predicate contains x, then we evaluate the bound on x and check and see if it is a weakest bound. In case it is weakest, we record the position of p in c's list of predicates as the location of the weakest lower or weakest upper bound depending on the sign of x's coefficient in p. To accommodate

this mechanism, each clause maintains two indices indicating the location of the predicate defining weakest upper and lower bounds when c is unate. Since the predicates keep an evaluation summary, this allows us to access the weakest upper and lower bounds of a unate clause in constant time, which helps the consistency checking mechanism.

Profiling has indicated that predicate evaluation has dominated the runtime of all non-trivial problems we have tested. The evaluation caching mechanism helped a great deal, and could be further refined without too much effort. Model mutation as in [MKS09] has the advantage that only *changes* in variable valuations trigger numerical processing, and for linear expressions, this can be accomplished with a few simple operations. However, model mutation may require evaluation of predicates which play no role in any unate clauses. It remains unclear whether model mutation or lazy evaluation with evaluation caching is more helpful or whether they can be combined.

Unate Index

Upon an assignment $x \mapsto a$, the watched clauses associated with x are scanned and each such clause c may or may not be a non-trivial unate clause for some variable y. In the event that c is non-trivial y-unate, we want to add c to a list of unate clauses associated with y. At the same time, when the solver backtracks and x becomes unassigned, we want to remove c from y's list of unate clauses. To accomplish both these operations in constant time, we associate two lists of clauses for each variable v. One list indicates v-unate clauses and the other list indicates clauses which became unate upon assignment of v. The lists are maintained by attributing two additional pointers to each clause and both are structured as stacks. Hence when v becomes unassigned, we can pop the clauses which became unate upon v's assignment and each such clause is guaranteed to be on the top of the stack associated with the variable for which the clause is unate. This variable is in turn accessible from the first position of a weakest upper or lower bound predicate. Note however, that this also is related to the backtracking choice. In particular, if one does not backtrack to the asserting level, then the unate index needs to be updated as if the unate clause were propagated further back in the assignment stack.

These data structures allow the solver to backtrack without ever scanning the predicates of a clause or the coefficients in a predicate. The removal of clauses which became non-trivial unate clauses upon an assignment $x \mapsto a$ is the only task which needs to be performed upon unassigning x.

5.5.5 Numerical Considerations

Our solver uses a pair of fixed width 64-bit integers to represent rationals and correspondingly a 64-bit representation of integers. This has sufficed for initial experimentation.

5.6 Experimentation

We ran experiments on 3 subsections of SMT-LIB [BRST08]: QF_RDL/scheduling, QF_IDL/diamonds, and QF_IDL/parity. These subsections consist of difference logic, do not cause any numerical problems and are straightforward to code as linear real arithmetic. All experiments were run on a Sun Java VM version 1.6.0_11 on a Debian Linux machine with dual Xeon 3.20 GHz processors and 4GB RAM.

5.6.1 Job Shop

Job shop scheduling problems are formulated as a set of jobs, each of which is a sequence of tasks. Each task makes use of a pre-specified resource for some fixed duration. With a fixed set of resources, a query is generated as to whether all tasks can complete within some given time, referred to as the *makespan*. These problems become exponentially more difficult as the query approaches the optimal makespan from above or below. These problems fall within the difference logic fragment of linear arithmetic and intersecting SMT solvers usually make use of dedicated algorithms such as those we presented in Chapter 4.

Wide Net Experiment

Our first experiment consisted of casting a wide net over the configuration space for jobshop problems. We identified a set of configurations for experimentation; namely all reasonable combinations of variable selection, value selection, and backtrack depth selection. The fixed variable ordering does not make sense with varying backtracks, and so we only tested one fixed variable selection configuration. Otherwise, all combinations were tried, yielding a total of 19 configurations to run on 105 benchmark problems. To limit total computation time, we limited each try of a configuration on a benchmark to 15 seconds. There were a total of 1995 problem/configuration tries, of which only 576 were solved in the 15 second time limit.

configuration			solved	time (s)	sat	time	unsat	time
var	val	bias toggle						
evsid	cur	all	31	78.1	18	58.0	13	20.1
evsid	cur	asrt	30	100.9	14	50.2	16	50.7
evsid	cur	no	28	68.2	15	43.1	13	25.1
evsid	last	all	33	93.7	17	63.9	16	29.8
evsid	last	asrt	38	87.9	21	59.3	17	28.6
evsid	last	no	31	69.3	16	57.5	15	11.8
evsid	rec	all	33	75.3	17	26.6	16	48.8
evsid	rec	asrt	29	52.2	14	21.9	15	30.3
evsid	rec	no	28	49.6	13	29.5	15	20.1
fix	last	no	9	23.6	2	7.0	7	16.6
vsid	cur	all	32	81.4	16	44.1	16	37.3
vsid	cur	asrt	30	91.2	15	72.9	15	18.4
vsid	cur	no	24	97.5	9	54.6	15	42.9
vsid	last	all	33	86.4	16	50.9	17	35.5
vsid	last	asrt	36	85.8	19	53.8	17	32.1
vsid	last	no	35	84.6	19	66.4	16	18.3
vsid	rec	all	32	74.3	16	44.9	16	29.3
vsid	rec	asrt	31	58.1	16	43.4	15	14.7
vsid	rec	no	33	95.0	17	71.0	16	24.0

The variable selection entries may be one of

1. *fix*. The solver uses a fixed variable ordering based on variable identities[1].

2. *vsid*. All variables are selected according to VSID heuristics, and all variables are incremented on every clause resolution step.

3. *evsid*. The solver does extended assertion level backtracking as described in Section 5.4 and selects variables based on VSIDs.

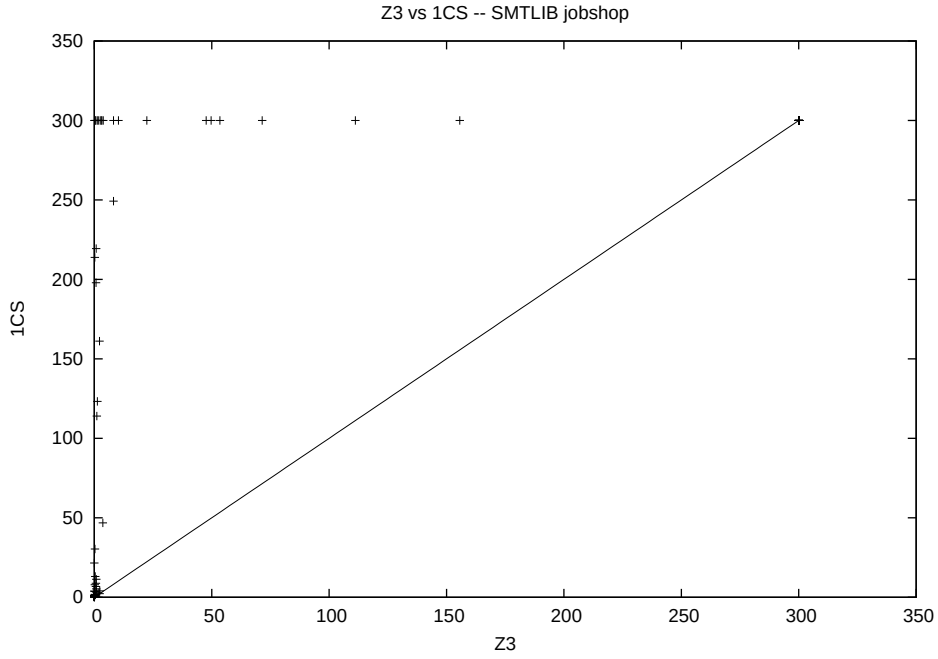
The most important configuration choice appears to be whether or not a fixed variable order is used. Apart from this, we observe that value selection plays a very important role and that the "last" configuration outperforms the "recent" configuration which in turn generally outperforms the "current" configuration. The bias flipping mechanism also appears to have a significant impact on performance. Generally, bias toggling on assignments seems to work best. But in the best overall configuration, bias toggling on assertion appears to work best. There is a large span of improvement over the space of

[1]We also implemented the structural heuristic mentioned in [MKS09], but in this case the result was worse than the variable-id based ordering.

configurations: the best configuration solves more than 4 times the problems of the worst.

Best Configuration

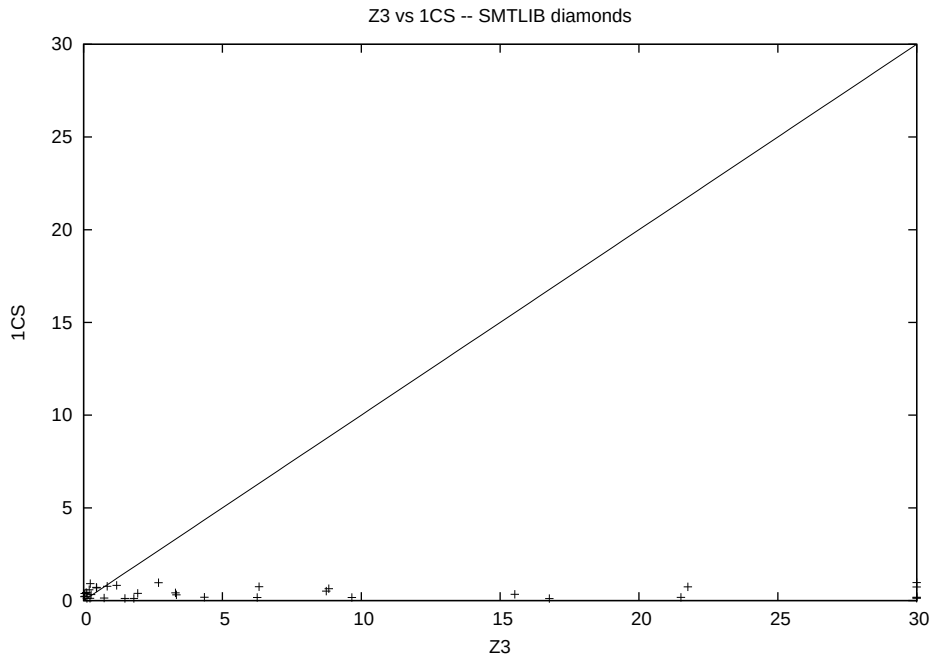
We ran the best configuration with extended asserting backtrack levels, VSIDs, and assertion bias toggling with a timeout of 300 seconds on the same set of problems and with the same machine. A side-by-side of the results against Z3 are summarized below. Z3 is vastly faster than UCS on scheduling problems.



5.6.2 Diamonds

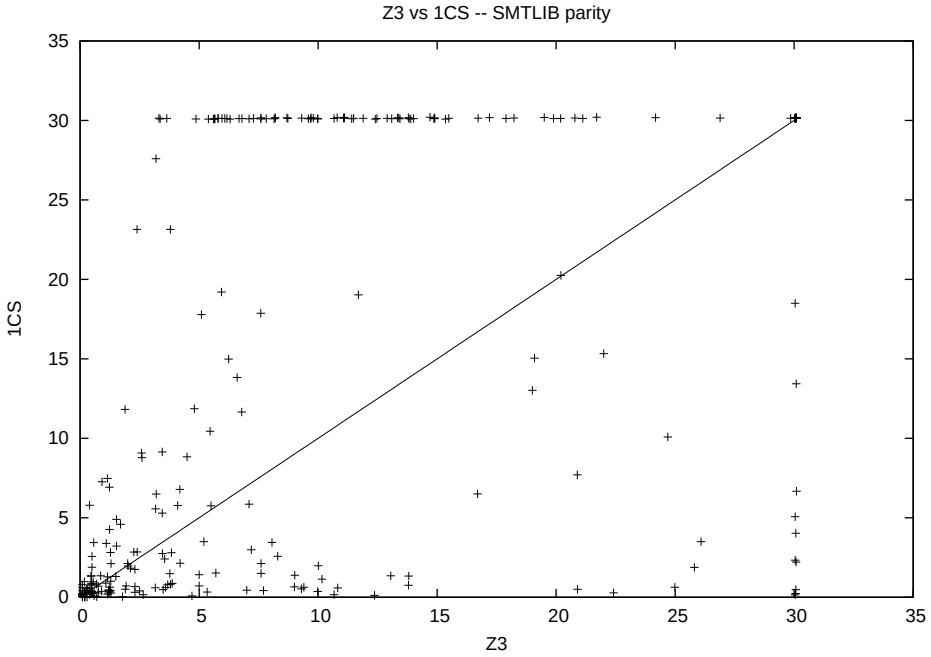
In [MKS09], it was observed that the resolution proof rule can produce exponentially shorter proofs for diamond shaped problems. We confirm this

observation for the diamond problems in SMT-LIB with a comparison of our best configuration to Z3.



5.6.3 Parity Games

Another class of difference logic problems from SMT-LIB contains a mix of propositional and integer variables. These problems are in CNF form, with only at most one difference constraint per clause. For these problems, a modification of our solver allowed coding propositional variables and performing unit propagation on them. Otherwise, our best configuration for the job shop problems was used. The largely off-diagonol results indicate that the relative strengths of UCS and Z3 are somewhat orthogonal.



5.7 Conclusion

DPLL-like direct model search poses a number of problems which do not appear in the propositional case. A careful examination reveals that many mechanisms in and properties of propositional SAT solvers have some analogues in the case of linear arithmetic with UCS; but they also tend to introduce a degree of complexity. We have identified and presented solutions for some of the problems, such as a theoretic solution to dynamic variable ordering with tree resolution, a lazy evaluation mechanism, and consistency checking. Our experimental observations suggest that for difference logic, the method has a very different performance profile than traditional SMT solvers. While the method performs well on diamond problems, the job shop problems suggest that there may exist problems with exponentially longer shortest proofs in UCS than using traditional DPLL(T) based methods. The parity problems show a highly variant performance profile in comparison to

Z3. We conjecture that this has to do with the degree of acyclicity found in the difference constraint graphs.

Much more work remains to be done. In particular, we found that pure tree resolution is impractical for a complete algorithm, hence the problem of dynamic variable ordering remains incompletely addressed. However, in the context of an incomplete algorithm with clause recording, our analysis of tree resolution aids in identifying useless clauses which can be forgotten and shows considerable improvement over fixed variable orderings for the job shop problems. For a better understanding of how the method works, an experimental examination of full linear constraints is necessary. Finally, the question of integration with a search in an extended model with propositional variables associated with each predicate has not yet been addressed.

Chapter 6

Conclusion

Satisfiability solving is fundamental problem whose potential for application grows as more efficient means are found. Some principles in satisfiability solving have emerged as fundamental and effective across a full spectrum of types of problems, such as conflict directed learning and dynamic variable ordering. At the same time, some problems, such as effective mixing of different kinds of reasoning, have remained resistant to principled solutions. In this thesis, we have contributed to several aspects of satisfiability and SMT solving.

Our first set of contributions addresses propositional satisfiability. We have presented a linear time algorithm for recursive self-subsumption based clause minimization. We have reported a direct relationship between restart frequency and VSID recency parameters, and we have presented a novel restart strategy which results in the ability to solve some hard problems which remain out of the reach of other general purpose methods.

The main focus of this thesis has been in the domain of satisfiability modulo theories. This thesis contributes to SMT on two levels, abstract and concrete. On the abstract level, we have presented a proof the Nelson-Oppen combination procedure based on a different condition. Secondly, we have formalized an extension of DPLL(T), flexible propagation, which supports decoupling and combining different degrees of theory-specific reasoning. Finally, we introduced a formal DPLL-like algorithm, UCS, with conditions and proofs of correctness and termination based on a formalization of the resulting underlying proof graph. This formal algorithm may be viewed as an extension of a family of GDPLL processes which allows forgetting clauses while retaining important notions related to correctness and termination.

On a more concrete level, our SMT contributions address two topics. First, we present a fast DPLL(T) theory solver for difference logic. The DL solver introduces a fast theory propagation algorithm which exploits consistency checks with the ideas behind flexible propagation and describes how to adapt a fast shortest paths algorithm to the problem. The ideas behind this solver have been adapted to various competitive SMT solvers and consistently proven effective in competition.

Second, this thesis studies the problem of instantiating UCS for real linear arithmetic. A comparison with the propositional case is presented, showing analogs of many mechanisms commonly found in modern DPLL propositional solvers. These analogs introduce a degree of complexity in implementation as well as restrictions in the form and efficiency of proofs required for termination. We have presented efficient methods of implementation which address some of the problems specific to UCS. Initial experimentation was presented for the case of difference logic, showing a very different performance profile than is found in traditional SMT solvers.

In sum, this thesis has addressed several problems related to satisfiability solving.

Bibliography

[ABH+08] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabour, and L. Sais. A Generalized Framework for Conflict Analysis. Technical report, Microsoft Research, 2008.

[ACG+04] A. Armando, C. Castellini, E. Giunchiglia, M. Idini, and M. Maratea. TSAT++: an Open Platform for Satisfiability Modulo Theories. In *Proceedings of the 2nd Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2004.

[AJPU07] M. Alekhnovich, J. Johanssen, T. Pitassi, and A. Urquhart. An exponential separation between regular and general resolution. *Theory of Computing*, 3:81–102, 2007.

[BCGS06] R. Bruttomesso, A. Cimatti, A. Franzn A. Griggio, and R. Sebastiani. Delayed Theory Combination vs. Nelson-Oppen for Satisfiability Modulo Theories: a Comparative Analysis. In *Technical Report DIT-06-032, Informatica e Telecomunicazioni, University of Trento.*, 2006.

[BDdM08] N. Bjørner, B. Dutertre, and L. de Moura. Accelerating Lemma Learning Using Joins – DPLL(⊔). In *In Int. Conf. Logic for Programming, Artif. Intell. and Reasoning (LPAR)*, 2008.

[BEGJ98] M. L. Bonet, J. L. Esteban, N. Galesi, and J. Johanssen. Exponential Separations between Restricted Resolution and Cutting Planes Proof Systems. In *Proc. FOCS*, pages 638–647, 1998.

[Ber01] D. Le Berre. Exploiting the Real Power of Unit Propagation Lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.

[Bie08a] A. Biere. Adaptive Restart Control for Conflict Driven SAT Solvers. In *Proc. of the 11th International Conference on Theory and Applications of Satisfiability Testing*, 2008.

[Bie08b] A. Biere. PicoSAT and PicoAigerSAT entering the SAT-Race 2008. In *System Descriptions of Solvers entering SAT-Race, 2008*, 2008.

[Bie08c] A. Biere. Picosat Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[BKS04] P. Beame, H. Kautz, and A. Sabharwal. Understanding the Power of Clause Learning. In *Journal of Artificial Intelligence Research*, 2004.

[BRST08] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[BSU97] N. S. Bjørner, M. E. Stickel, and T. E. Uribe. A Practical Integration of First-order Reasoning and Decision procedures. In *In Proc. of the 14th Intl. conference on automated deduction, volume 1249 of lncs*, pages 101–115. Springer-Verlag, 1997.

[BT03] P. Baumgartner and C. Tinelli. The Model Evolution Calculus. In Franz Baader, editor, *CADE-19 – The 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 350–364. Springer, 2003.

[BvdPTZ07] Bahareh Badban, Jaco van de Pol, Olga Tveretina, and Hans Zantema. Generalizing dpll and satisfiability for equalities. *Information and Computation*, 205, August 2007.

[CAB+02] A. Cimatti, G. Audemard, P. Bertoli, A. Kornilowicz, and R. Sebastiani. A SAT-Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *CADE*, 2002.

[CG96] B. V. Cherkassky and A. V. Goldberg. Negative-Cycle Detection Algorithms. In *ESA '96: Proceedings of the Fourth Annual European Symposium on Algorithms*, pages 349–363, London, UK, 1996. Springer-Verlag.

[CLRS90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 1990.

[CM06] S. Cotton and O. Maler. Satisfiability modulo theory chains with DPLL(T). In *Verimag Technical Report* `http://www-verimag.imag.fr/TR/TR-2006-4.pdf`, 2006.

[Coo71] S.A. Cook. The Complexity of Theorem Proving Procedures. In *Proceedings, Third Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[Cot05] S. Cotton. Satisfiability Checking with Difference Constraints. Master's thesis, Max Planck Institute, 2005.

[DdM06] B. Dutertre and L. de Moura. Integrating simplex with dpll(t). Technical report, SRI International, May 2006.

[DHN07] N. Dershowitz, Z. Hanna, and A. Nadel. Towards a better understanding of the functionality of a conflict-driven sat solver. In *In Proc. Theory and Applications of Satisfiability Testing (SAT)*, 2007.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. In *Numer. Math.*, volume 1, pages 269–271, 1959.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. In *Communications of the ACM*, volume 5(7), pages pages 394–397, 1962.

[dMB07] L. de Moura and N. Bjørner. Model Based Theory Combination. In *SMT Workshop 2007*, 2007.

[dMB08] L. de Moura and N. Bjørner. Engineering DPLL(T) + Saturation. In *Int. Joint Conf. on Automated Reasoning (IJCAR)*, 2008.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(1):201–215, 1960.

[EB05] N. Eén and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, 2005.

[EMS07] N. Eén, A. Mishchenko, and N. Sörensson. Applying Logic Synthesis for Speeding up SAT. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, 2007.

[ES03] N. Eén and N. Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[ES05] N. Eén and N. Sörensson. MiniSat: A SAT Solver with Conflict-clause Minimization. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, 2005.

[FMSN98] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully Dynamic Shortest Paths and Negative Cycles Detection on Digraphs with Arbitrary Arc Weights. In *European Symposium on Algorithms*, pages 320–331, 1998.

[Gel09] A. Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In *In Proc. Theory and Applications of Satisfiability Testing (SAT)*, 2009.

[GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *CAV'04*, pages 175–188, 2004.

[GMT03] E. Giunchiglia, M. Maratea, and A. Tacchella. (in)effectiveness of look-ahead techniques in a modern sat solver. In *CP*, pages 842–846, 2003.

[GN02] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT Solver, 2002.

[GNT01] E. Giunchiglia, M. Narizzano, and O. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. In *Artificial Intelligence*, pages 275–281, 2001.

[GNT06] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *Journal of Artificial Intelligence Research*, 26:371–416, 2006.

[Gol01] A. V. Goldberg. Shortests Path Algorithms: Engineering Aspects. In *Proceedings of the Internation Symposium of Algorithms and Computation*, 2001.

[GS09] Carla P. Gomes and Ashish Sabharwal. *Exploiting Runtime Variation in Complete Solvers*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 9, pages 271–288. IOS Press, February 2009.

[GSF08] D. Goldwasser, O. Strichman, and S. Fine. A Theory-Based Decision Heuristic for DPLL(T). In *Proc. Formal Methods in Computer Aided Design (FMCAD)*, 2008.

[Hak85] A. Haken. The Intractability of Resolution. *Theoretical Computer Science*, 1985.

[HBPG08] P. Hertel, F. Bacchus, T. Pitassi, and A. V. Gelder. Clause Learning Can Effectively P-Simulate General Propositional Resolution. In *Proc. 23rd AAAI conf. on Artificial Intelligence*, 2008.

[JC07] H. Jain and E. Clark. SAT Solver descriptions: CMUSAT-base and CMUSAT. In *System Descriptions for the SAT Competition 2007*, 2007.

[Joh77] D.B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. Assoc. Comput. Mach.*, 24:1, 1977.

[KMS97] H. Kautz, D. Mcallester, and B. Selman. Exploiting variable dependency in local search. In *In Abstracts of the Poster Sessions of IJCAI-97*, 1997.

[KV05] P. Koppensteiner and H. Veith. A Novel SAT procedure for Linear Real Arithmetic. In *PDPAR*, 2005.

[M. 93] M. Luby and A. Sinclair and D. Zuckerman. Optimal speedup of las vegas algorithms. In *Israel Symposium on Theory of Computing Systems*, pages 128–133, 1993.

[MKS09] K. L. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing DPLL to Richer Logics. In *Proc. of Computer Aided Verification (CAV)*, 2009.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC'01*, 2001.

[MR02] L. De Moura and H. Rue. Lemmas on Demand for Satisfiability Solvers. In *In Proc. Theory and Applications of Satisfiability Testing (SAT)*, pages 244–251, 2002.

[MSS96] J. P. Marquez-Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Intl. Conf. on Computer Aided Design'96*, pages 220–227, November 1996.

[MSS99] J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. In *IEEE Trans. on Computers*, volume 48, 1999.

[NMA+02] P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, O. Maler, and N. Jain. Verification of timed automata via satisfiability checking. In *Lecture Notes in Computer Science*, volume Volume 2469, pages 225 – 243, Jan 2002.

[NO79] G. Nelson and D.C. Oppen. Simplification by Cooperating Decision Procedures. In *ACM Trans. on Programming Languages and Systems*, volume 1(2), page 245257, 1979.

[NO05] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In *CAV'05*, volume 3576 of *LNCS*, pages 321–334, 2005.

[NOT04] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract dpll and abstract dpll modulo theories. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, 2004.

[NOT05] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and Abstract DPLL Modulo Theories. In *In 11h Int. Conf. Logic for Programming, Artif. Intell. and Reasoning (LPAR)*, volume LNAI 3452, 2005.

[Oli08] A. Oliveras. Homepage of barcelogictools, 2008.

[PD07] K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *Proc. of the*

10th International Conference on Theory and Applications of Satisfiability Testing, 2007.

[Rob65] J. A. Robinson. A Machine-oriented Logic based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[RS08] V. Rivchin and O. Strichman. Local Restarts. In *Theory and Applications of Satisfiability Testing*, 2008.

[RSJ95] G. Ramalingam, J. Song, and L. Joscovicz. Solving difference constraints incrementally. Technical report, IBM T.J. Watson Research Center, October 1995.

[RT03] S. Ranise and C. Tinelli. The SMT-LIB Format: An Initial Proposal. In *PDPAR*, July 2003.

[Sab05] A. Sabharwal. *Algorithmic Applications of Propositional Proof Complexity*. PhD thesis, University of Washington, Seattle, 2005.

[SAJ+08] C. Sinz, N. Amla, T. Jussila, D. Le Berre, P. Manolios, L. Zhang, H. Jain, and H. Post. Sat Race 2008. In *Presented at Conf. on Theory and Applications of Satisfiability Testing*, 2008.

[SB09] N. Sörensson and Armin Biere. Minimizing learned clauses. In *In Proc. Theory and Applications of Satisfiability Testing (SAT)*, 2009.

[Seb94] R. Sebastiani. Applying GSAT to Non-clausal Formulas. *Journal of Artificial Intelligence Research*, 1:1–309, 1994.

[Sil99] J. P. M. Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Proc. of the 9th Portuguese Conf. on Artificial Intelligence*, pages 62–74, 1999.

[SKC93] B. Selman, H. Kautz, and B. Cohen. Local Search Strategies for Satisfiability Testing. In D. S. Johnson and M. A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, October 1993.

[SLKB07] T. Schubert, M. Lewis, N. Kalinnik, and B. Becker. Miraxt – a multi-threaded SAT solver. In *System Descriptions for the SAT Competition 2007*, 2007.

[Sör08] N. Sörensson. MINISAT 2.1 and MINISAT++ 1.0 – Sat Race 2008 Editions. In *System Descriptions for the SAT Race 2008*, 2008.

[SS90] G. Stålmarck and M. Saflund. Modelling and Verifying Systems and Software in Proposional Logic. In *Ifac SAFECOMP90*, 1990.

[SS96] J. L. Marques Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *ICCAD*, pages 220–227, 1996.

[SSB02] O. Strichman, S. Seshia, and R. Bryant. Deciding separation formulas with sat. In *Proc. of Computer Aided Verification (CAV'02)*, July 2002.

[Str02] O. Strichman. On solving presburger and linear arithmetic with sat. In *In Proc. of Formal Methods in Computer-Aided Design*, pages 160–170. Springer, 2002.

[Str06] O. Strichman. Building small equality graphs for deciding equality logic with uninterpreted functions. *Information and Computation*, 204, 2006.

[Tar81] R. E. Tarjan. Shortest paths. In *AT&T Technical Reports*. AT&T Bell Laboratories, 1981.

[TH96] C. Tinelli and M. Harandi. A New Correctness Proof of the Nelson-Oppen Combination Procedure. In *Frontiers of Combining Systems, volume 3 of Applied Logic Series*, pages 103–120. Kluwer Academic Publishers, 1996.

[Tse68] G. S. Tseitin. On the Complexity of Derivations in the Propositional Calculus. In A. O. Slisenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125, 1968. Translated from Russian.

[WGG06] C. Wang, A. Gupta, and M. K. Gannai. Predicate Learning and Selective Theory Deduction. In *Design Automation Conference (DAC)*, 2006.

[WW99] Wolfman and Weld. The LPSAT system and its Application to Resource Planning. In *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1999.

[ZM03] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, page 10880, 2003.

[ZMMM01] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, 2001.

[ZS96] H. Zhang and M. Stickel. An Efficient Algorithm for Unit-propagation. In *Proc. of the Fourth International Symposium on Artificial Intelligence and Mathematics*, 1996.