Satisfiability Checking with Difference Constraints

Scott Cotton

A masters thesis supervised by
Andreas Podelski and Bernd Finkbeiner
IMPRS Computer Science
Saarbruecken

May 13, 2005

Abstract

This thesis studies the problem of determining the satisfiability of a Boolean combination of binary difference constraints of the form $x - y \leq c$ where x and y are numeric variables and c is a constant. In particular, we present an incremental and model-based interpreter for the theory of difference constraints in the context of a generic Boolean satisfiability checking procedure capable of incorporating interpreters for arbitrary theories. We show how to use the model based approach to efficiently make inferences with the option of complete inference.

Contents

List of Figures

Acknowledgements

There are several people whose aid and support made this thesis possible. In particular, Oded Maler provided the topic, the funding, and much advice. Without his support, none of this work would have been possible. Andreas Podelski welcomed me at IMPRS. Even under constraints of a different sort, he has ensured that the IMPRS masters program would be both pedagogical and feasible. Kerstin Meyer-Ross made the apparently impossible possible, performing veritable miracles from time to time. Moez Mahfoudh provided the starting point with his thesis on the topic. I would like to extend my most sincere thanks to these folks and to my family for their kind support and patience during the undertaking of this work.

Chapter 1

Introduction

1.1 Introduction

This thesis studies the problem of determining the satisfiability of a Boolean combination of binary difference constraints of the form $x - y \leq c$ where x and y are numeric variables and c is a constant. Such constraints, here called *difference constraints*, are often used in the modelling and verification of timed systems. Algorithms for the satisfiability and inclusion problem for conjunctions of difference constraints are well known and applied in a variety of tools such as Kronos [36] or Uppaal [9]. However, the development of algorithms treating arbitrary Boolean combinations of difference constraints has recently become a topic of research. This research is in part motivated by limitations in the tools for verifying timed systems and also by the successes witnessed in the algorithmics associated with Boolean functions.

1.2 Related Work

Methods for determining the satisfiability of a Boolean combination of difference constraints can be broadly categorized into two branches. In the first branch, a Boolean combination of difference constraints is translated into a diagram or data structure which is the object of various algorithms. This approach often follows the principles of binary decision diagrams [5], in which a propositional formula is given a canonical form. This approach is better suited for solving a wider array of problems than that of satisfiability checking – in particular the methods are generally designed to include efficient

conjunction, disjunction and negation. In its simplest and most often used form, the formula is maintained as a list of matrices each of which encodes a conjunction of difference constraints. The elements in the list represent disjuncts of the formula. However, there can be a great deal of redundancy in the structure, since entire conjunctions are treated atomically. This observation led to the development of difference decision diagrams [22], and clock difference diagrams [18] which eliminate the redundancy by defining and using a canonical order over the set of difference constraints.

The second branch makes use of algorithms for Boolean satisfiability and is more focused on the satisfiability problem. Work that falls in this category can make use of Boolean SAT solvers in any number of ways. One straightforward method is to translate a formula with difference constraints to an equisatisfiable formula in propositional logic, and then employ an off-the-shelf SAT solver. This method, utilized in [30], can be fast when the translation is fast. However, the translation can often create a large, even exponential, formula.

Another straightforward method, which we call the lazy method, is to use a Boolean SAT solver repeatedly on a formula which replaces each difference constraint with a Boolean variable. If at any time the SAT solver determines the problem is not satisfiable, the process is complete. Otherwise, the satisfying assignment $\mathcal{A}$ is used to check the consistency of the set of difference constraints required to satisfy the formula under the assignment $\mathcal{A}$. If this set is not consistent, a new constraint is added to the formula and the process continues. If the set is consistent, the problem is satisfiable. This approach is taken for example by MathSAT [21], and makes use of model enumeration capabilities available in many SAT solvers.

Finally, one can extend a Boolean SAT solver so that it can interpret difference constraints directly, in a fashion exactly synchronized with the Boolean SAT solver's interpretation of Boolean variables. While there are problems for which this method can be slower than either of the previous methods, there are also various advantages. In particular, this approach is more dynamic in how it adapts to a given input problem. It can find inconsistencies earlier than in the lazy method and at the same time it can avoid much of the processing required in the translation based methods by ignoring altogether many of the possible sets of difference constraints which are inconsistent. Perhaps most importantly, the direct method allows for constraint propogation based on the interpretation of the difference constraints. Constraint propogation recursively identifies and then interprets those con-

straints which are implied by a given set of interpreted constraints. Such constraint propogation can accelerate the solving process a great deal, in exactly the same fashion that Boolean constraint propogation accelerates Boolean SAT solving.

All of the methods which make use of SAT solvers have been generalized to handle constraints of various types. They are not in general restricted to difference constraints. For example, many translation based methods are in use which solve any arithmetical constraints with the finite model property [26], such as those constraints in the language of Presburger arithmetic. The lazy method can make use of any external solver capable of determining the satisfiability of a conjunction of constraints, and thus can be easily generalized to a wide class of constraints. In addition, a framework called DPLL(T) was presented in [12] which implements the direct method for any type of constraints by defining an interface for a theory specific solver which interprets the constraints and communicates with the Boolean SAT solving process. As is the case for the lazy method, all that is required of an external solver is the ability to determine the consistency of a conjunction of constraints. However, the interface between the theory solver and the SAT solver allows for some interaction, such as constraint propogation, which is not possible in the lazy approach.

1.3 Contributions

In this thesis, we present a novel implementation of the direct method for checking the satisfiability of a Boolean combination of difference constraints. In particular, we present a solver for the theory of difference constraints within the DPLL(T) framework. The solver is unique in that it maintains a stack of models that mirrors the stack of truth assignments made in the SAT solving process. Within this structure, a single model at the top of the stack provides the basis for fully incremental consistency checking as well as for constraint propogation. All the while, the stack structure as a whole allows for both efficient backtracking and a compact representation which shares information between models. This methodology is in contrast to other approaches, such as those found in [19, 27], which maintain a representation of *all* the models associated with any given truth assignment. By comparison, these approaches suffer from scalability in the number of numeric variables or from inefficient backtracking, in which the representation of all the models

must be recalculated. In addition, our approach offers more efficient consistency checking for sparse systems of constraints, which is the norm in the problems we have encountered.

One of the most developed features of our prototype solver is its treatment of constraint propogation. We give methods for complete propogation as well as for cheaper incomplete propogation. In the later case, we present an effective heuristic which experimentally outperforms complete propogation and which does not add to the asymptotic complexity of our incremental consistency checking algorithm. In both cases, we employ a method called round robin constraint propogation, which takes advantage of the DPLL(T) framework in a way that allows for semi-lazy constraint propogation. The method is in addition largely based on observations which are independent of difference constraints *per se* and thus may be applicable in other contexts.

In addition, we evaluate our methods in two experimental settings with a prototype implementation. First, we examine the classic problem of job shop scheduling and second we examine the problem of maximum stabilization time of a combinatorial circuit, phrased as a problem in the bounded model checking of timed automata.

1.4 Organization

In chapter 2, we present the DPLL procedure for propositional satisfiability checking with an emphasis on widely adopted modern optimizations. This chapter describes the framework within which we later incorporate difference constraints and forms the primary the basis for chapter 3. In chapter 3, we present a generic DPLL procedure with which the satisfiability of a Boolean combination of atoms in a given theory may be determined. In chapter 4, we study properties of difference constraints and graphical representations of systems of difference constraints under dense and integer domains. In chapter 5 we present an implementation of a theory interpreter for the theory of difference constraints within the framework of the generic DPLL procedure presented in chapter 3 and based on ideas presented in chapter 4. Chapter 6 contains descriptions of experiments, results, and the analysis of the results. We conclude in chapter 7.

Chapter 2

DPLL

The Davis-Putnam-Loveland-Logemann procedure for determining the satisfiability of a propositional formula originated in [10]. This procedure works on propositional formulae in conjunctive normal form and proceeds by branching on truth values of propositional variables, backtracking whenever an assignment to these variables falsifies a constraint. Although this overall framework is quite straightforward, there are many variations in the components which lead to widely varying performance. In fact the performance of DPLL procedures over the last decade has drastically improved at a rate well above what can be attributed to Moore's law.

In this chapter, we present the DPLL procedure with a focus on components which are common to many of the most performant solvers, and which are generally perceived as essential for a performant DPLL solver by the SAT community. While this study is not concerned with attaining a more efficient propositional satisfiability solver *per se*, we will lift the techniques presented here to a DPLL based solver capable of handling difference constraints. In doing so we will leverage some of the recent success witnessed by propositional solvers, but we will first proceed with a presentation of well exercised techniques within the DPLL framework for propositional formulae.

2.1 Overview

The DPLL procedure recursively explores a tree of truth assignments to the variables in a formula ϕ, which is required to be in conjunctive normal form. The tree is formed as follows. First an arbitrary order is given to the variables

in the formula. The root of the tree represents the first variable in this order. More generally, any node at depth i in the tree will represent the ith variable. Each node n has two children c_n^+ and c_n^-. The edge between a node n at depth i and c_n^+ represents the assignment $\{v_i \mapsto \texttt{true}\}$, and similarly the edge to c_n^- represents the assignment $\{v_i \mapsto \texttt{false}\}$. In this way every path in the tree from the root to a leaf represents a total assignment A. The algorithm, given in pseudo code below, recursively searches the tree of possible assignments by building up partial assignments as it goes deeper into the search tree.

Figure 2.1: Pseudocode for the Davis-Putnam-Loveland-Logemann procedure.

```
def DPLL(φ, A):
    stat := getStatus(φ, A)
    if (stat = SAT) then return true
    else if (stat = UNSAT) then return false
    else if A ∪ {v ↦ h} is deducible for some variable v and truth value h
        return DPLL(φ, A ∪ {v ↦ h})
    else
        let v be the next unassigned variable
        if DPLL(φ, A ∪ {v ↦ true}) return true
        else return DPLL(φ, A ∪ {v ↦ false})
```

The function `getStatus` determines whether a partial assignment renders ϕ satisfiable, unsatisfiable, or unknown. The *deducibility* of a variable's truth value under a partial assignment refers to a very simple, incomplete form of propositional deduction specific to formulae in conjunctive normal form: if ϕ simplifies [1] under assignment A to a formula in the form $x \wedge \ldots$ for some variable x, then $A \cup \{x \mapsto \texttt{true}\}$ is deducible. The case of $\neg x \wedge \ldots$ is treated similarly.

This simple backtracking framework is the theoretically fastest known algorithm for determining the satisfiability of a propositional formula. However, vast performance improvements can result from careful attention to

[1] The notion of simplification is described in more detail in the next section covering conjunctive normal form.

implementation details and heuristics. For example, the order given to the variables turns out to be quite important, and has been the subject of various studies. In addition, the process of identifying deducibility of variable's truth values and transitively assigning deducible variables from other assignments has been observed to occupy the majority of DPLL solver's time [37]. Consequently, this process has been highly optimized in various solvers [23, 14], also leading to significant performance enhancements.

Perhaps the most important development is the addition of a completely new component to the algorithm: learning. As stated above, when the algorithm detects that the formula is unsatisfiable under an assignment, it simply backtracks to the next available assignment. With learning, the algorithm first identifies a small subset of the assignment which is sufficient to lead to unsatisfiability of the formula. It encodes the negation of this assignment as a constraint that must be satisfied if ϕ is to be satisfied, and remembers this constraint as the remainder of the assignments are explored. The effect of the added clause on pruning the assignment tree can be dramatic.

2.2 Conjunctive Normal Form

The DPLL procedure requires formulae in conjunction normal form (CNF). To determine the satisfiability of an arbitrary formula, a translation to conjunctive normal form is necessary. In this section we define CNF, present some translations for arbitrary formulae to CNF and discuss how formula simplification works in CNF.

Definition 2.2.1. Conjunctive Normal Form (CNF)

- A *literal* is a propositional variable v or its negation $\neg v$.
- A *clause* is a disjunction of literals. Moreover a clause must contain no duplicate literals and must not contain both a variable and its negation.
- A propositional formula is in *conjunctive normal form* if it is a conjunction of clauses.

DPLL thus operates over formulae of the form $\bigwedge_{i \in I}(\bigvee_{j \in J_i} l_j)$ where each literal l_j is a propositional variable or its negation, i indexes clauses and each $j \in J_i$ indexes disjuncts within clause i.

This form allows the DPLL algorithm to easily simplify a formula under an assignment. If a variable x maps to `true`, then any clause $\ldots \vee x \vee \ldots$ becomes solved, and any clause $l_1 \vee \ldots \vee \neg x \vee \ldots \vee l_n$ simplifies to $l_1 \vee \ldots \vee l_n$. Additionally, a clause with no disjuncts indicates an unsatisfiable assignment and a clause with exactly one disjunct x or $\neg x$ (also called a *unit clause*) indicates a deducible assignment for x. Finally, if all the clauses are solved under a partial assignment, then the formula is clearly satisfiable under any extension of that assignment.

The procedure transitively deduces assignments in unit clauses. This process, called *unit propogation* was introduced in [10]. Most importantly, unit propogation serves as *lookahead* in the search of the assignment space. The search space is reduced whenever an assignment x or $\neg x$ is deduced via a unit clause, simply because there is no need to guess the truth value of x.

An important side-effect of unit propagation is the introduction of a consistency invariant to the solution process. Since the solver does not guess the truth value of a variable unless there are no unit clauses, it is guaranteed that at every point immediately prior to guessing a variable, the problem is 1-consistent. In other words, the partial assignment prior to a decision can always be extended to include any one unassigned variable with the guarantee that the resulting assignment will not produce an empty clause. We call this property *decision time* 1-*consistency*. Some extensions of DPLL lose this property and consequently do not implement techniques which rely on it.

2.2.1 Simple Translation

One can translate an arbitrary propositional formula into CNF using standard Boolean equivalences in two phases. First the negation operator is pushed down to subformulae:

$$\begin{aligned} \neg(\phi \wedge \psi) &\rightarrow \neg\phi \vee \neg\psi \\ \neg(\phi \vee \psi) &\rightarrow \neg\phi \wedge \neg\psi \\ \neg\neg\phi &\rightarrow \phi \end{aligned} \tag{2.1}$$

Second, as long as there is a disjunct over a conjunct the following transformation is applied:

$$\phi \vee (\psi \wedge \psi') \quad \rightarrow \quad (\phi \vee \psi) \wedge (\phi \vee \psi')$$

The major drawback of this translation is that it can yield an exponential formula. Consider formulae of the form $\bigvee_{0 \leq i \leq n} (x_i \wedge y_i)$. Translation of these formulae by simple Boolean equivalences yields

$$\bigwedge_{w \subseteq \{0,1,\ldots,n\}} \phi_w \text{ where } \phi_w = \bigvee_{i \in w} x_i \vee \bigvee_{i \notin w} y_i$$

which is clearly exponential in the size of the original. A more compact translation is needed.

2.2.2 Translations with extra variables

Numerous efficient translations to conjunctive normal form exist, such as those of Tseitin [33] or Wilson [34], that make use of extra variable. All of them are based on the idea of adding variables that are equivalent to subformulae of the formula to be translated. Here we present a minor variation of these translations.

As before, we first push the negation operator down to the propositional variables and eliminate double negations using the mapping as presented in equation 2.1. Second, we translate any disjunct $\varphi \vee \psi$ that is not already in the form of a clause. To do so, we recursively build an equisatisfiable formula $C_\varphi^x \wedge C_\psi^{\neg x}$ constructed with the aid of a new variable x. We first translate φ to a set of clauses C_φ, then likewise translate ψ to a set of clauses C_ψ. We then append x as a disjunct to each clause in C_φ, resulting in a set of clauses C_φ^x, and similarly we append $\neg x$ as a disjunct to each clause in C_ψ, resulting in $C_\psi^{\neg x}$. Now if φ is satisfiable, C_φ is satisfiable and so is C_φ^x. Moreover, we can set $x \mapsto \texttt{false}$ without affecting the satisfiability of C_φ^x, but satisfying $C_\psi^{\neg x}$. The case of ψ being satisfiable follows symmetrically.

If we consider what happens if neither φ nor ψ are satisfiable, then under any assignment over φ, at least one clause in C_φ will not be satisfied. This however implies $x \mapsto \texttt{true}$ in C_φ^x. Similarly, if ψ is not satisfiable then under any assignment over ψ atleast one clause in C_ψ is not satisfied. Hence under any assignment for ψ, $C_\psi^{\neg x}$ implies $x \mapsto \texttt{false}$. Thus any assignment satisfying neither φ nor ψ will imply both $x \mapsto \texttt{true}$ and $x \mapsto \texttt{false}$, making

the resulting translation unsatisfiable as well. We can then translate any $\varphi \vee \psi$ that is not a clause to $C_{\varphi}^{x} \wedge C_{\psi}^{\neg x}$ and the resulting formula is equisatisfiable.

This translation is slightly more compact than Wilson's [34] and more compact than Tseitin's [33]. For every disjunct of size n, it uses at most $n-1$ variables and only adds clauses via conjunction. Additionally, we further compact the translation as follows. Let D be a set of formula, and $D_l = \{d \in D \mid d \text{ is a literal } \}$. We can rearrange generalized disjuncts $\bigvee D$ into the form $(\bigvee D_l) \vee (\bigvee (D \setminus D_l))$. In this form, $\bigvee D_l$ will be translated to a single clause and only $|D \setminus D_l|$ new variables are introduced. This last technique can greatly reduce the number of variables introduced in the translation.

2.3 Variable Ordering Heuristics

An important component of the performance of a DPLL based procedure is the order which is given to the variables in a formula. Since most problems will be satisfiable or provably unsatisfiable based on a subset of the variables, it would be advantageous to consider these variables independently of the rest and hence to process them first. Unfortunately, identifying a minimal set of variables for proving or disproving satisfiability is at least as hard as determining satisfiability. Hence heuristics are used to determine the most important variables and these variables are considered first in the assignment search.

2.3.1 MOMs Heuristics

One common class of heuristics is the MOMs heuristic (Maximum Occurrences in Minimum length Clauses) [25]. The intuition behind MOMs heuristics is to branch on the most constrained variables, and to determine a degree of constainedness by occurrence in small clauses. There are many variations on MOMs based heuristics and this class of heuristic is very often used in DPLL solvers. In simple form, this heuristic maintains a rank for each variable based on the tuple $(m_c(v), ct(v))$ where $m_c(v)$ is the size of the minimum sized clause of a variable v and ct is the number of occurrences of a variable within clauses of size $(m_c(v))$. A notable enhanced variation by Jeroslow and Wang [16] ranks each variable v with

$$r(v) = \Sigma_{c \in C_v} 2^{-|c|}$$

Where C_v is the set of clauses containing v.

The simpler MOMs heuristics can be efficiently maintained during the solving process but tend to suffer from overemphasis on short clauses.

2.3.2 Literal Counting Heuristics

The maintenance of heuristics based on clause size can become unduly expensive during the solving process. A more light weight approach is simply to maintain a count of the clauses in which a variable or literal occur. In this framework the truth value of a variable may be chosen at random or determined based on the phase with maximum count.

An important class of literal counting heuristics are those that

1. Only increase variables or literals scores.

2. Periodically divide all scores by a constant factor.

These heuristics are very light weight and can be effective when the condition for increasing a variables' score reflects the difficulty of finding a satisfying assignment involving the variable. Two of the most competitive current solvers use this approach, calling it VSID (Variable State Independent Decay). In [23], the condition for increasing scores of a literal is its presence in a learned clause. In [14], the condition is generalized to involvement in a proof of unsatisfiability under a given assignment.

2.3.3 Clause Based Heuristics

A completely different heuristic introduced in [14] enjoys a good deal of success in SAT solvers which learn clauses. The idea is to keep learned clauses in a stack, and to choose variables on which to branch from the most recently learned clause which is not satisfied. When there are multiple variables to choose from, counting based heuristics described above are used as a backdrop. This form of heuristic tends to pick variables which are more strongly interrelated under the current assignment and allows the solver to focus on hard subproblems first.

2.4 Efficient Constraint Propogation

The recursive process of keeping track of unit clauses and simplifying based on their assignments has been observed by Zhang to occupy the majority of DPLL based solvers' time [37]. This constraint propogation suffers from two bottlenecks. First, the clause simplification can involve a scan of the variables in any clause containing the negation of the last assignment. The purpose of this scan is determine whether or not the clause has become unit or empty as a result. The second bottleneck arises from backtracking, during which the status of each clause needs to be determined. In both cases, a naive implementation will scan a clause in order to maintain clause status. Note that there is no bottleneck if a clause is solved by an assignment, since no scan of the clause is necessary.

In this section, we present the method dubbed *two literal watching*, originating from [37]. The effect of the method is twofold:

1. Backtracking can be performed without any work involving the clauses in a problem.

2. Clause simplification is unlikely to require a scan.

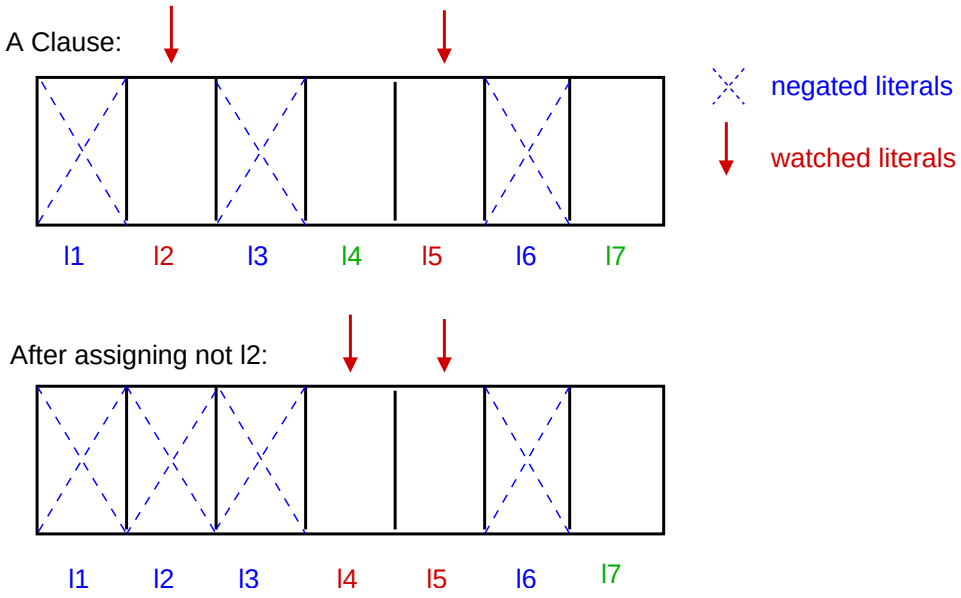
The central idea of this method is to keep track of two distinguished or "watched" literals in a clause, maintaining for as long as possible the invariant that the variables associated with the literals are unassigned. With this invariant, it is easy to see that a clause is unit when exactly one of the two watched literals is unassigned and empty when both watched literals are assigned. Hence the status of clause can be determined in constant time.

Many variations of this technique are in use [23, 14, 17], often involving a extra conditions for determining a new watched literal. While these variations can induce better interaction with variable ordering heuristics, the major benefits remain the same with or without the extra conditions. However simple the method, it is difficult to underestimate its benefits in comparison to more naive implementations.

2.5 Conflict Directed Learning

Learning is in the author's view the most interesting development in DPLL based solvers. While there is a long history in constraint programming of

Figure 2.2: Two literal watching assignment update requiring a scan of the clause.



similar procedures of augmenting the set of constraints to make a problem easier to solve, conflict directed learning is a relatively new phenomenon for propositional satisfiability solving. The intuition behind conflict directed learning is simple: "to learn from mistakes". Whenever the solver finds an assignment that violates a constraint, a.k.a. a conflict, it figures out a succint and sufficient *cause* of the conflict, which comes in the form of a condition $c(\bar{x})$ over some of the variables in the problem. It so happens that the negation of this condition is invariably a single clause and can be *learned* simply by adding $\neg c(\bar{x})$ to the set of clauses to be satisfied. This in turn reduces the space the of assignments that the solver must explore, and thus accelerates the solution process.

Described at this high level, learning makes a lot of sense. However, there are many different criteria for determinig a "succinct and sufficient cause of a conflict" and it is not well-understood why some criteria work better than others. In this section we present the mechanisms behind conflict directed learning and discuss briefly the various ways of determing the cause of a conflict.

2.5.1 Implication Graph

The analysis of a conflict begins by the construction of an *implication graph*. The graph is built by undoing the constraint propogation, starting at the place of conflict, which is an empty clause. The graph is built with vertices consisting of assigned variables in the form of literals. Whenever a literal was deduced because it was the last literal in a unit clause, an edge is added from the negation of the other literals in the clause to the deduced literal. For example, under the assignment $\{x \mapsto \texttt{false}, y \mapsto \texttt{true}\}$, the clause $x \vee \neg y \vee z$ would produce the implication graph

$$(V = \{\neg x, y, z\}, E = \{\langle \neg x, z\rangle, \langle y, z\rangle\})$$

Implications are followed backwards from the literals in the empty clause until guessed literals are found. Since guessed literals are not deduced, no implications may be followed from guessed literals.

Example 2.5.1. Suppose a formula contains the clauses below. The variables x_1, x_2 and x_3 are guessed. The variables y_{1-8} are deduced. The clause in the upper left is an empty clause. The corresponding implication graph is given in figure 2.3.

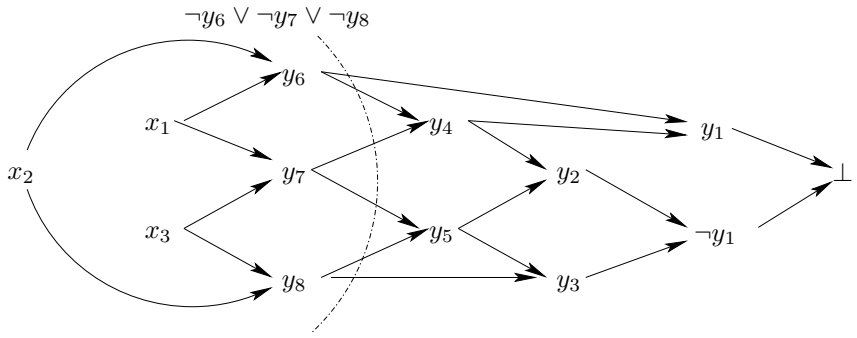
$$\begin{array}{lll}
\neg y_1 \vee \neg y_2 \vee \neg y_3 & \neg y_8 \vee \neg y_5 \vee y_3 & \neg y_7 \vee \neg y_8 \vee y_5 \\
\neg y_4 \vee \neg y_6 \vee y_1 & \neg y_6 \vee \neg y_7 \vee y_4 & \neg x_1 \vee \neg x_2 \vee y_6 \\
\neg y_4 \vee \neg y_5 \vee y_2 & \neg x_2 \vee \neg x_3 \vee y_8 & \neg x_1 \vee \neg x_3 \vee y_7
\end{array}$$

2.5.2 The Various Cuts

A *cut* in the implication graph is a set of edges separating the negations of guessed literals from the point of conflict. Each cut C results in a learned clause consisting of the negations of the literals from which there is an edge in C. In the figure 2.3, the cut resulting in the learned clause $\neg x_6 \vee \neg x_7 \vee \neg x_8$ is indicated. There are many different possible cuts of an implication graph, and different cuts have been defined and studied experimentally. It is not well understood why some cuts perform better than others.

One of the first cuts used was the one consisting of all the edges emanating from negations of guessed literals. This cut is called a *decision cut* because

Figure 2.3: Implication graph of a conflict.



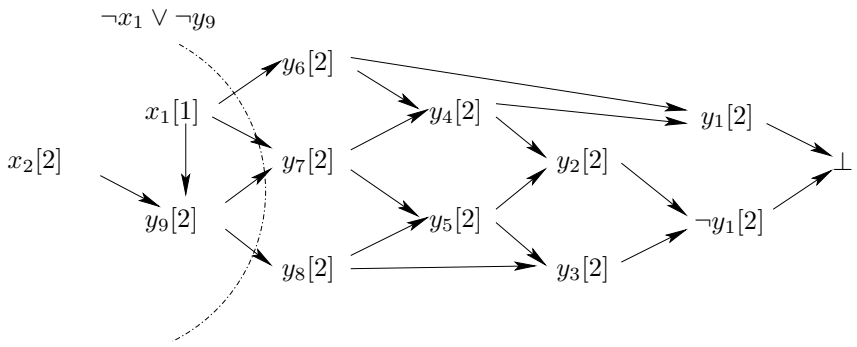
it learns clauses involving decision variables. This cut was first introduced in [4]. Perhaps the most often used cut is called a *first unique implication point* (1UIP) cut [28]. To describe it, we introduce a few more terms. The *decision level* of an assigned literal l is the number of guessed literals at the time of the assignment of l. The literal of a decision level d is the dth guessed literal. A literal l in a constraint graph is called a *unique implication point* if every path from the literal of the decision level of l to the point of conflict passes through l. Clearly the literal of every decision level is a unique implication point. The 1UIP cut of of an implication graph is the cut generated by the unique implication point closest to the place of conflict.

Example 2.5.2. We follow a modification of example 2.5.1. The variables x_1 and x_2 are guessed and the rest are deduced. The clause in the upper left is an empty clause. The corresponding implication graph with the 1UIP cut is given in the figure 2.4. The decision cut is $\neg x_1 \vee \neg x_2$.

$$\begin{array}{ccc} \neg y_1 \vee \neg y_2 \vee \neg y_3 & \neg y_8 \vee \neg y_5 \vee y_3 & \neg y_7 \vee \neg y_8 \vee y_5 \\ \neg y_4 \vee \neg y_6 \vee y_1 & \neg y_6 \vee \neg y_7 \vee y_4 & \neg x_1 \vee y_6 \\ \neg y_4 \vee \neg y_5 \vee y_2 & \neg x_1 \vee \neg x_2 \vee y_9 & \neg x_1 \vee \neg y_9 \vee y_7 \\ & \neg y_9 \vee y_8 & \end{array}$$

The 1UIP cut works much better in practice than the decision cut. Some have suggested that this is because the decision cut constrains variables which

Figure 2.4: Implication graph of a conflict with 1UIP cut.



are already somewhat constrained by backtracking, whereas the 1UIP cut tries to involve non decision variables. Multiple variations have been studied. For example, the zChaff solver [23] contains code to learn all uip cuts, find the minimum cut, etc. None of these other methods have performed well enough to receive published consideration as a competitive learning scheme.

2.5.3 Non Chronological Backtracking

After learning a clause which describes the reason for a conflict, the solver must backtrack. The learned clause gives additional information which allows the solver to backtrack closer to the root of the search tree than otherwise. In particular, the solver can backtrack to the deepest point in the search tree that

1. includes a literal in the learned clause
2. is less deep than the last decision leading to the conflict.

Such non chronological backtracking helps to keep the solver close to the root of the search tree. This technique also interacts with the cut used to learn a clause. In particular, some cuts will always produce clauses which will be unit after backtracking, so the solver can continue with its constraint propogation before guessing another variable. The 1UIP and decisions cut have this property.

2.6 Conclusion

We have presented the DPLL procedure with an emphasis on well accepted modern optimizations and heuristics. This procedure provides the basis of chapter 3, in which we present a generic DPLL framework which can handle arbitrary constraints in addition to binary propositional variables.

Chapter 3

Parametric DPLL

In this chapter we present the structure and algorithms involved in a parametric DPLL based satisfiability solver for a blend of propositional variables and atoms interpreted under a given theory. In order to accomplish this, we will first present a few more implementation details for DPLL than were presented in chapter 2. We will then present an interface for an arbitrary theory to be plugged into DPLL based on work in [32, 12]. Finally, we will review the necessary changes to the DPLL procedure to incorporate a pluggable theory.

3.1 DPLL Implementation

A modern DPLL solver does not use the recursive implementation presented in chapter 2. In the recursive implementation, there is an implicit stack of truth assignments and an implicit way of exhaustively traversing the search tree. In a modern implementation, this stack is maintained explicitly and, due to learning, there is no need to explicitly ensure an exhaustive traversal of the search tree. Below is pseudocode for the overall non recursive algorithm.

```
while(decide()):
    while(¬simplify()):
        if (¬resolveConflict):
            return not SAT
return SAT
```

The function `decide` finds an unassigned literal with highest priority

according to the variable ordering heuristic. It pushes the literal on a stack of decisions. If there are no more literals to assign, it returns `false`, indicating the problem is satisfiable.

The function `simplify` maintains a queue of implications. Initially, it takes the most recently assigned literal and pushes it on the queue. It then pops the assigned literal from the queue and finds unit or empty clauses in which the negation of the literal appears. For any unit clause it finds, it pushes the implied literal onto the queue together with a pointer to the unit clause, and pushes the literal onto a stack of assigned variables associated with the decision variable. If it encounters an empty clause, it empties the queue and then returns `false`. If it does not encounter an empty clause it simply continues, ultimately returning `true` when there are no more implications in the queue. The function `resolveConflict` starts with the empty clause and builds the implication graph as described in chapter 2. It learns a new clause based on the implication graph, adds the clause to set of clauses to be solved, and then backtracks, popping elements from the decision stack until it finds an unexhausted literal. If there is no such literal, it cannot backtrack, so it returns `false` indicating the problem is not satisfiable.

Other major data structures not previously mentioned include the clause database, the priority queue holding literals at their heuristic priority, a structure for efficiently relating literals with their negations, with their respective variables, with the clauses implying them, etc.

3.2 A Generic Theory Interface

In this section we present an interface between an interpreter for terms in a theory and a generic DPLL procedure. This presentation closely follows that given in [12]. The only differences is that the interface is here widened a little bit to allow for more flexibility in the implementation of an interpreter for a theory. Where there are differences, we give a justification for the additional options given to the theory.

In short, a theory interpreter should

1. **Provide a means to interpret an atom in the theory**.
   The DPLL procedure will ask the theory to interpret the atom and indicate whether the resulting interpretation is consistent. This may occur as a decision point or as a deduction. The DPLL procedure will indicate whether the interpretation request is the result of a decision

or a deduction so that the theory may use this information to more efficiently backtrack. Whenever this is done, the DPLL procedure will pass a queue of implied literals to the theory interpreter so that the interpreter may add implications to the queue.

2. **Provide a list of implied atoms in the theory under a given assignment**.
   At the end of unit propogation, before returning from the simplification procedure, the DPLL procedure will ask the theory for a set of atoms implied under the current assignment. The theory implementation thus has the option of doing implications in a fine grained manner via the queue during the interpretation of an atom or in batch via this communication point.

3. **Provide explanations for literals implied by the interpretation**.
   If an implied literal contributes to the derivation of a conflict, then the conflict analysis mechanism may ask the implication for a set of reasons for the literals' assignment. This may occur long after the literal is first assigned, but never after the literal is unassigned. [1] The reasons for an implication must be taken from the set of assigned atoms *at the time the implication was made*.

4. **Provide a means to backtrack**.
   The DPLL procedure will communicate backtracking to the theory implementation in two ways. First, every assigned literal which has an interpretation in the the theory will be unassigned, in the reverse order of assignment. Each unassignment is communicated to the theory. Second, the DPLL procedure will indicate to the theory the decision level to which it backtracks, after it is done backtracking. The latter point of communication may allow for more efficient batch backtracking while the former may be easier to implement since there would be no need to keep track of decision levels.

3.3 Some Changes to DPLL

In this section we outline some changes to the dpll procedure that accomodate such a theory.

[1] For 1UIP, the duration is further limited to the current decision level.

Figure 3.1: DPLL modified to accept decisions inconsistencies.

```
while (true):
    stat = decide()
    if(stat = OK)
        if (¬propogate()) return not SAT
    else if(stat = INCONSISTENT):
        if (¬resolveDecisionConflict):
            return not SAT
        if (¬propogate()) return not SAT
    else
        return SAT

def propogate():
    while(¬simplify()):
        if (¬resolveConflict):
            return not SAT
    return OK
```

3.3.1 Decision Time Inconsistency

When the literals represent a term in a theory, more work needs to be done in a DPLL based solver. In particular, assignments to an atom a may negate atoms other than $\neg a$ even if not otherwise related in a clause. Hence when the assignment resulting from a decision or an implication negates an atom that is already assigned, an inconsistency may arise. In the propositional case, an inconsistency may only arise from an implication, since their are no unit clauses present whenever a decision is made. In addition, the treatment of inconsistencies resulting from decisions is different than the treatment of inconsistencies arising from implications. In particular, in the former case, there is no need to do 1UIP learning, since the negation of the inconsistency is already 1UIP. Regarding control flow, after the resolution of a conflict, the solver immediately continues simplifying since the associated learned clause may be unit after backtracking [2]. This property should hold for decision induced inconsistencies as well. A modification of DPLL for decision induced inconsistencies implementing these features is outlined in figure 3.1.

[2]In fact the learned clause is always unit after backtracking in a decisions or 1UIP cut

3.3.2 Pure Literals

A common simplification rule in SAT solvers is the *pure literal* rule, which assigns any literals whose negations do not occur in the CNF formula. This rule is not in general valid in the presence of a theory [32, 3, 12]. However, it can be applied to literals which are not interpreted by a theory.

3.3.3 Constraint Propogation

In the propositional case, constraint propogation consists of finding unit clauses which result from truth assignments. With the introduction of a theory which interprets the literals, each assignment may negate other atoms and so may lead to more unit clauses or earlier detection of conflicts. If more unit clauses are identified, then the Boolean constraint propogation should continue. It is then clear that constraint propogation should alternate between Boolean constraint propogation and constraint propogation specific to a theory. A remaining question is how frequently the alternation occurs. It is entirely possible to perform theory specific constraint propogation on every assignment to a term in the theory, or equally possible to alternate more coarsely, finishing all the Boolean propogation before starting theory specific propogation and vice versa. In both cases, the same set of implications will be made, only in a different order.

In the former case the theory in question may benefit from incremental implications due to the availability of a fast algorithm for managing incremental differences to the interpretation. In the latter case, the theory may do less overall work because terms which are implied by unit propogation and by the theory's interpretation of the truth assignment will be handled by the unit propogation. However, in this batch form some optimizations will not be possible. For example, the ZChaff solver [23] minimizes the size of the clause which implies a given literal by choosing a clause of smallest size amongst all that imply the literal. This is done by keeping track of multiply implied literals and would not be applicable if the theory implementation only informed DPLL of the implication of unassigned literals.

3.4 Conclusion

We have presented a generic interface for satisfiability modulo a theory, given initially in [12], which in turn was derived from [32]. Our presentation differs

from that in [12] in that we provide a wider interface which allows for a greater deal of flexibility in the implementation of a theory interpreter. We have discussed some of the consequences of making use of the additional flexibility.

Chapter 4

Difference Constraints

In this chapter we present basic properties of difference constraints taken individually and in Boolean combinations.

Definition 4.0.1. A *difference constraint* is a formula taking the form $x - y \leq c$ or $x - y < c$ for numeric variables x and y, and constants c.

Notation 4.0.1. Given a Boolean combination of difference constraints and propositional variables φ, we refer to the set of difference constraints contained in φ as D_φ. We refer to the truth value of a difference constraint $x - y \leq c$ by d^c_{xy}. We refer to the formula which results from replacing every difference constraint $x - y \leq c$ in φ by a new propositional variable d^c_{xy} representing the truth value of $x - y \leq c$ as the *Boolean abstraction of* φ, written $\alpha(\varphi)$. Given an assignment $\mathcal{A}$ to the truth values of difference constraints in φ, we refer to the *system of constraints induced by* $\mathcal{A}$ as the system of difference constraints given by $\{x - y \leq c \mid \mathcal{A}(d^c_{xy}) = \texttt{true}\}$.

4.1 Conjunctions and Constraint Graphs

Below is a system of difference constraints in matrix form.

$$\begin{array}{cccccccccc}
x_0 & -x_1 & & & & & & \leq & c_1 \\
 & +x_1 & -x_2 & & & & & \leq & c_2 \\
 & & +x_2 & -x_3 & & & & \leq & c_3 \\
 & & & \vdots & & \cdots & & & \vdots \\
 & & & & +x_{n-2} & -x_{n-1} & & \leq & c_{n-1} \\
 & & & & & +x_{n-1} & -x_n & \leq & c_n
\end{array}$$

In this form, it is easy to see that if all the equations are added together, one ends up with $x_0 - x_n \leq \Sigma_{i=1}^{n} c_i$. This is the fundamental property of systems of difference constraints, namely transitivity. In this example, it is easy to see that if $x_0 = x_n$ and $\Sigma_{i=1}^{n} c_i < 0$, we end up with the contradiction $x_0 - x_0 \leq c$ where c is negative. We will soon show that if there are no such cycles, a system of difference constraints is satisfiable.

More generally, systems of difference constraints are matrices in which every coefficient is taken from $\{-1, 0, 1\}$ and in which every constraint (row) has at exactly one coefficient with value 1 and exactly one coefficient with value -1. This restricted form linear constraints has a convenient graphical representation. Given a system of difference constraints such as above, we create its *constraint graph* by representing every variable as a vertex, and constraint $x - y \leq c$ by a c-weighted edge from x to y. In this graph the weight of a path from x_0 to x_n corresponds to adding up the constraints in the path just as we did in matrix form above. Thus every path in the graph from x to y represents an implied constraint $x - y \leq c$ where c is the weight of the path. Since $x - y \leq w$ implies $x - y \leq w'$ for any $w' \geq w$, the *shortest* path between any two two vertices in the graph corresponds to the strongest constraint between the corresponding numeric variables.

Notation 4.1.1. The graphical representation is sufficiently faithful that we often use the term "variables" and "vertices" interchangeably and also interchange the terms "edges" and "constraints". We even interchange the terms *constraint graph* and *system of difference constraints*. Furthermore, given an assignment $\mathcal{A}$ to the truth values of difference constraints, we denote the constraint graph of the system of constraints induced by $\mathcal{A}$ as $G_{\mathcal{A}}$. Similarly, we say $G_{\mathcal{A}}$ is satisfiable if the system of constraints induced by $\mathcal{A}$ is satisfiable.

The analogy between the graphical representation and systems of difference constraints applies between potential functions and satisfying assignments, but with an interesting twist. A *potential function* is a function $p : V \to D$, where D is the appropriate numeric domain. A potential function p must satisfy

$$p(u) + W_{uv} \geq p(v)$$

for every edge (u, v) with weight W_{uv}. With this property we can derive that

$$\begin{aligned} p(u) + W_{uv} &\geq p(v) \\ -p(u) - W_{uv} &\leq -p(v) \\ -p(u) &\leq W_{uv} + -p(v) \\ -p(u) - -p(v) &\leq W_{uv} \end{aligned}$$

Hence, given a potential function π of a constraint graph, the function $-\pi = v \mapsto d \iff \pi(v) = -d$ is a satisfying assignment of the corresponding system of difference constraints. We thus have that a system of difference constraints is satisfiable if and only if there exists a potential function in the corresponding constraint graph.

Given a vertex s which can reach every vertex in V, the shortest paths distances from s to every vertex in V is a potential function. To see why, consider any shortest path distance function $\delta : V \to D$ which is not a potential function. Then there is an edge (u, v) such that $\delta(u) + W_{uv} < \delta(v)$. If δ describes the shortest path distances from s, then there exists a shorter path from s to v, namely the shortest path from s to u followed by the edge (u, v), a contradiction. Hence, given a constraint graph $G = (V, E)$ and a vertex s which reaches all vertices in V, if the shortest paths from s to every $v \in V$ are well defined, then there exists a satisfying assignment for the system of difference constraints corresponding to G.

If the shortest paths from s are *not* well defined, then there exists a vertex v such that for any path from s to v, there exists a shorter path from s to v. Hence there is an infinite sequence of paths from s to v $p_1, p_2, p_3, \ldots$ with the property that $w(p_1) > w(p_2) > w(p_3) > \ldots$. As every path from s to v with weight w implies a constraint $s - v \leq w$, for any assignment $A : V \to D$, there exists an implied constraint $s - v \leq w$ such that $A(s) - A(v) > w$. Hence the system of difference constraints is not satisfiable if the shortests paths are not well defined. Taken together with the above result, we have that a system of difference constraints is satisfiable if and only if the shortest paths are well defined.

For any vertex s reaching every vertex in V, the shortests paths are well defined if and only if there does not exist a negative cycle in G. To see why, consider the set of shortest simple paths from s to every $v \in V$. If there is no negative cycle, then the shortest simple paths are shortests paths. If there is a negative cycle c, then let v be a vertex on the negative cycle. A shorter path than the shortest simple path from s to v may be obtained by

concatenating the shortest simple path from s to v with the negative cycle involving v, starting from v. More generally, let p_i be the shortest simple path from s to v concatenated with i traversals of the negative cycle. Then $w(p_i) < w(p_j)$ whenever $i > j$. So the sequence $p_0, p_1, p_2, \ldots$ is an infinite sequence of paths with descending weights. Hence the shortests paths from s are not well defined.

In summary, we have that

1. A system of difference constraints is satisfiable if and only if there exists a potential function in the corresponding constraint graph.

2. A system of difference constraints is satisfiable if and only if the shortests paths from any vertex s reaching every vertex in its constraint graph are well defined.

3. The shortests paths from any vertex s reaching every vertex in a constraint graph are well defined if and only if the graph contains no negative cycle.

which leads immediately to the following

Proposition 4.1.1. A system of difference constraints is satisfiable if and only if its constraint graph contains no negative cycles.

4.1.1 Negative Cycle Detection

In this section we present negative cycle detection algorithms. In the broader context of Boolean combinations of difference constraints, the problem of negative cycle detection is very important. It will be necessary to determine the existence of lack thereof of a negative cycle in many constraint graphs, in the worst case exponentially many times. Consequently, a procedure for negative cycle detection often dominates the running time of DPLL with difference constraints, and so we wish to perform this detection procedure as efficiently as possible.

Negative cycle detection for a constraint graph $G = (V, E)$ is generally accomplished by augmenting G with a new source vertex s, and adding edges $\{(s, v) \mid v \in V\}$ with weight 0. In the augmented graph, we solve the single source shortest path (SSSP) problem, which is to say we find the shortest path from s to each vertex $v \in V$. The SSSP must fail to find a shortest path if there is a negative cycle.

There are many different SSSP algorithms, but the most widely used variants are based on *edge relaxations*, in which a conservative estimate of the shortest path distances $\pi : V \to D$ is refined by repeatedly taking any edge (u, v) with weight W_{uv} such that $\pi(v) > \pi(u) + W_{uv}$ and relaxing the edge by setting $\pi(v) = \pi(u) + W_{uv}$. Once this occurs, it may propogate, creating a circumstance in which v has an edge (v, w) which may be relaxed. If there is no negative cycle, the total number of edge relaxations required to find the shortest paths distances π^* is $|V| \cdot |E|$, since no single relaxation can propogate to more than $|V|$ vertices.

The common Bellman-Ford ford algorithm iterates over all the edges $|V|$ times, relaxing any edge (u, v) such that $\pi(v) > \pi(u) + W_{uv}$. Finally, one further pass over the edges is made to check to see if any more relaxations are possible. If an edge is relaxable, then there is a negative cycle. This algorithm always runs in the worse case time $|V| \cdot |E|$. The problem with this algorithm is that although it is possible to detect a negative cycle well before completion, no attempt to do so is made.

A variant of the algorithm above is the Bellman-Ford-Moore algorithm, which uses the push-relabel method. With this method, a queue is maintained containing all the vertices which have been relabelled with a new distance. One vertex v at a time is removed from the queue and relaxations are performed on qualified outgoing edges (v, w). Each such w is then pushed onto the queue. This algorithm does not terminate when there is a negative cycle. However, one can check for a negative cycle if the shortest paths tree is maintained as well as the distances. A check simply involves following the parent of a vertex in the shortest paths tree until either the root is found or the some vertex is encountered twice. This operation takes $\mathcal{O}(|V|)$ time. If it is performed every $|V|$ times a vertex is remove from the queue, the check becomes amortized over the queue removals and only costs a constant factor of extra run time. This algorithm does better than the basic Bellman-Ford algorithm at negative cycle detection, but still is unable to identify negative cycles exactly when they occur. Moreover, it was shown in [6] that cycles can appear and then later disappear from the shortest path tree, making it possible for this algorithm to run long beyond the appearance of the first negative cycle.

There are two SSSP algorithms which identify negative cycles immediately, and they are in some sense duals of each other. Both maintain a queue just as in the push-relabel method. The first is due to Tarjan [31]. This algorithm maintains the shortest path tree, and whenever the distance to

Figure 4.1: Pseudocode for Bellman-Ford-Moore Algorithm with negative cycle detection.

```
n:=0
Q:={s}
while Q ≠ ∅:
    u:=pop(Q)
    n:=n + 1
    if n mod |V| = 0 then cycleCheck(u)
    for each edge (u, v) with weight W_uv:
        if δ(v) < δ(u) + W_uv then
            δ(v):=δ(u) + W_uv
            parent(v) := u
            push(v, Q)
```

vertex v is updated via an edge (u, v), it disassembles the subtree rooted at v, removing the vertices in the subtree from the queue. While disassembling the subtree, a check for a negative cycle can be done simply by whether u is in the subtree rooted at v. The subtree disassembly is amortized over the construction of the shortest path tree: there can be only as many vertices visited in the disassembly as there are vertices added to the tree. In this way, immediate negative cycle detection is possible without affecting the overall $|V| \cdot |E|$ runtime.

A dual of this algorithm is due to Goldberg and Radzik [13]. This algorithm performs depth first search on each vertex v with an updated distance. The depth first search propogates the distance updates to any vertex reachable from v in the *admissible subgraph* [1]. With this algorithm, the queue is kept full, whereas with the subtree disassembly method, the queue is restricted to most recently updated vertices. A cycle in the admissible subgraph with a relaxable edge is a negative cycle and is easily detected with depth first search. This method also offers immediate negative cycle detection, and in some sense is better than the subtree disassembly method because it checks for cycles in the whole admissible subgraph.

An experimental analysis of negative cycle detection algorithms was given

[1]The admissible subgraph is the subgraph consisting of all edges (u, v) satisfying $\texttt{distance}(v) \leq \texttt{distance}(u) + W_{uv}$

Figure 4.2: Pseudocode for Tarjan's subtree disassembly algorithm.

```
Q:={s}
while Q ≠ ∅:
    u:=pop(Q)
    for each edge (u, v) with weight W_uv:
        if δ(v) < δ(u) + W_uv then
            for every w a descendent of v in the shortest path tree:
                detach w from the shortest path tree
                remove w from Q
                if w = u then there is a negative cycle
            δ(v):=δ(u) + W_uv
            parent(v) := u
            push(v, Q)
```

Figure 4.3: Pseudocode for the Goldberg-Radzik algorithm.

```
Q:={s}
while Q ≠ ∅:
    u:=pop(Q)
    dfsScan(u, Q)

def dfsScan(u, Q)
    visiting(u):=true
    for each edge (u, v) with weight W_uv:
        if δ(v) < δ(u) + W_uv then
            δ(v):=δ(u) + W_uv
            parent(v) := u
            push(v, Q)
            if visiting(v) then  there is a negative cycle
        if δ(v) ≤ δ(u) + W_uv and v is not yet scanned then
            dfsScan(v, Q)
    visiting(u):=false
```

in [6]. On the whole, the Goldberg-Radzik algorithm and Tarjan's subtree disassembly method proved most efficient and robust. Tarjan's subtree disassembly method was often slightly faster than the Goldberg-Radzik algorithm, but by a small margin and there were some cases where the Goldberg-Radzik algorithm outperformed the subtree disassembly method by a wider margin.

4.1.2 Incremental Negative Cycle Detection

When an edge (u, v) is added to a constraint graph G with a potential function δ, there is a faster way to determine whether the resulting graph G' contains a negative cycle than running any of the above algorithms on G'. In this section, we present such an algorithm based largely on ideas given in [11]. We assume that for G we have computed a shortests paths tree π and a potential function δ. Moreover, we assume the weight of (u, v), denoted W_{uv}, is such that $\delta(u) + W_{uv} < \delta(v)$. If this is not the case, then δ and π are valid for G' since δ is a potential function for G' and π is consistent with δ. We wish to compute δ' and π', the distance function and a shortest paths tree for G'.

To perform the dynamic update, we will maintain a priority queue over the vertices in a Fibonacci heap with priorities Δ. We initialize the priority queue with $\Delta(v) = \delta(v) - (\delta(u) + W_{uv})$ and $\Delta(w) = 0$ for all $w \neq v$. We will initialize the new shortests path tree with $\pi'(v) = u$ and $\pi'(w) = \pi(w)$ if $w \neq v$. We first scan the descendents d_v of v in the shortests path tree π' to see if u is a descendant, in which case there is a negative cycle. As is done in Tarjan's subtree disassembly algorithm, we disassemble the subtree rooted at v as this is done. We then pop vertices from the priority queue, one at a time, and scan its outgoing edges as shown in figure 4.4.

The key idea behind this scheme is that $\Delta(v)$ measures the maximum change in potential via all edges (v, w), since $\delta(v)$ is a valid potential function. In other words, whenever v maximizes Δ with priority p_v, there does not exist an edge (u, v) such that $(\delta(u) - p_u) + W_{uv} < \delta(v) - p_v$, otherwise $\Delta(u) > \Delta(v)$ or δ is not a valid potential function. Hence the single source shortest path distance $\delta'(v) = \delta(v) - p_v$ is minimal. The algorithm runs in $\mathcal{O}(|V| \log |V| + |E|)$ time. Initiallization is $\mathcal{O}(|V|)$, each pop of the priority queue is $\mathcal{O}(\log |V|)$ and there are $\mathcal{O}(|V|)$ pops. In addition, every edge E is visited once. The subtree disassembly is amortized over the construction of the tree as in Tarjan's subtree disassembly algorithm. Finally, with a Fibonacci heap, the `increasePriority` function takes constant time.

Figure 4.4: Pseudocode for incremental negative cycle detection.

```
while Q ≠ ∅:
    (u, d):=pop(Q)
    δ'(u):=δ(u) − d
    for each edge (u, v) with weight W_uv:
        if v ∈ Q with priority p_v then
            if (δ(v) − p_v) < δ'(u) + W_uv then
                π'(v) := u
                for every w in the subtree rooted at v:
                    detach w from the shortest path tree if w ≠ v
                    if w = u then there is a negative cycle
                increasePriority(v, Q, δ(v) − (δ'(u) + W_uv))
```

A subtle difference between Tarjan's subtree disassembly algorithm and this one is that the check for a negative cycle must include the root of the subtree. Other nuances include whether to initialize the queue with the entire set of vertices or with the target of the new edge. In the latter case, a check for the presence of a vertex in the queue can be done prior to increasing its priority. If the vertex is not in the queue, then it is inserted, if it is in the queue, then its priority is increased. This variation can reduce the overhead of popping elements from the queue, since it is kept smaller. In addition, it allows for a sub $\mathcal{O}(|V|)$ lower bound, since no initialization is necessary.

4.2 Real and Integer Domains

When variables are taken over the integers, there is no need for for both strict ($<$) and non strict ($\leq$) constraints. One can simply express $x - y < c$ as $x - y \leq c - 1$. For dense numeric domains, this is of course no longer the case. However, it is notationally cumbersome to unilaterally make the distinction between strict and non strict constraints, and we have dropped the distinction at times [2] Here we provide a justification of this notation and at the same time present a means to encode dense domain systems of difference constraints as shortest paths problems.

[2]For example, when defining the constraint graph for a system of difference constraints.

Recall that we showed that systems of difference constraints are satisfiable if and only if shortest paths are well defined in the corresponding constraint graph. We will now use this equivalence in reverse. In particular, we'll show how to define shortest paths problems where the weights are in fact so called *bounds* or intervals of the form $(-\infty, c)$ and $(-\infty, c]$. We will refer to bounds such as these as $\mathbf{i}$ or $\mathbf{i}'$. We will also refer to the upper bound of the interval $\mathbf{i}$ as $c_{\mathbf{i}}$. The shortests paths problems are are definable and computable by two operations `min` and $+$. We define these operations over bounds as follows

1. $\mathbf{i} + \mathbf{i}' = \{x + y \mid x \in \mathbf{i}, y \in \mathbf{i}'\}$.
2. $\mathtt{min}(\mathbf{i}, \mathbf{i}') = \mathbf{i} \cap \mathbf{i}'$.

With these definitions, shortests paths can be computed when the edges are weighted with bounds. Now if we rewrite $x - y \leq c$ as $x - y \in (-\infty, c]$ and $x - y < c$ as $x - y \in (-\infty, c)$, we preserve the necessary properties for shortests paths and difference constraints. In particular, $x - y \in \mathbf{i} \wedge y - z \in \mathbf{i}'$ implies $x - z \in \mathbf{i} + \mathbf{i}'$, and every constraint $x - y \in \mathbf{i}$, explicit or implied, must be satisfied by the definition of `min`. In this context, we say the weight $\mathbf{i}$ of a path from x to y is negative if $\mathbf{i} = (-\infty, 0)$ or $c_{\mathbf{i}}$ is negative.

Thus when we previously defined constraint graphs by taking every constraint $x - y \leq c$ as a c-weighted edge from x to y, we implicitly assumed that all the constraints were non strict, and this assumption is only valid for integer variables. For real or rational variables, we rather take any constraint $x - y \in \mathbf{i}$ as an $\mathbf{i}$-weighted edge between x and y. Similarly, when we defined the notation d^c_{xy} for the truth value of a difference constraint $x - y \leq c$, we neglected to consider the possibility of strict constraints. If the variables are over a dense domain, we would instead write $d^{\mathbf{i}}_{xy}$ for the truth value of the constraint $x - y \in \mathbf{i}$. Instead of propogating these notational variants throughout this thesis, we will instead assume that $x - y \leq c$ refers to just that for integer domained problems and instead refers to a constraint $x - y \in \mathbf{c}$ for some bound $\mathbf{c}$ if the variables are taken over a dense domain.

The obvious data structure for bounds is a pair $\langle s, c \rangle$ where s is a Boolean variable denoting the strictness of the constraint and c is a number. In this form, we define $\langle s, c \rangle + \langle s', c' \rangle = \langle s \vee s', c + c' \rangle$ and $\mathtt{min}(\langle s, c \rangle, \langle s', c' \rangle) = \langle s, c \rangle$ if $c < c'$, $\langle s', c' \rangle$ if $c' < c$, and $\langle s \vee s', c \rangle$ if $c = c'$.

This presentation of bounds based shortests paths is a minor simplification of the method used in *difference bound matrices* used for example in tools

such as Kronos [36] and Uppaal [9], where bounds are extended to include the interval $(-\infty, \infty)$.

4.3 Diffference Constraints in CNF

Conjunctions of difference constraints are well understood and handled efficiently with relatively ease. In this section we present some basic properties of difference constraints in conjunctive normal form, defined in chapter 2 and discussed there for the case of propositional formulae.

4.3.1 Negation Closure

Since $\neg(x - y \leq c)$ is equivalent to $y - x < -c$, and $\neg(x - y < c)$ is equivalent to $y - x \leq -c$, the set of all difference constraints are closed under negation. Thus any CNF formula containing difference constraints may be written as a CNF formula containing only positive difference constraints.

4.3.2 Satisfying Partial Assignments

Another notable departure from the propositional case is that for a propositional formula, if a partial assignment solves all the clauses, the assignment can be extended to the unassigned propositional variables in any which way and the resulting assignment is a satisfying one. With difference constraints, this is no longer the case. But we do have a weaker property. In particular, if a satisfiable conjunction of difference constraints covers the clauses in a CNF formula, then the formula is satisfiable. In fact, a satisfying assignment may be constructed as follows. Let G be the constraint graph of the satisfiable conjunction of constraints covering all the clauses. Since G is satisfiable, it contains no negative cycles. Let $x - y \leq c$ be a difference constraint whose truth value is not known, *i.e.* it is not in the satisfiable conjunction of difference constraints. Then either the constraint $x - y \leq c$ or $y - x < -c = \neg(x - y \leq c)$ can be added to G without introducing a negative cycle. Otherwise, there would be a path p_{yx} from y to x in G with weight $w(p_{yx}) < -c$ and a path p_{xy} from x to y with weight $w(p_{xy}) \leq c$, and hence G would have a negative cycle, a contradiction.

Using this property, we now give a constructive argument. Given a Boolean combination of difference constraints φ in CNF and a partial truth

assignment $\mathcal{A}$ over the difference constraints D_φ, we have the following property. If $G_\mathcal{A}$ contains no negative cycles, then we can extend $\mathcal{A}$ to include a truth value for an arbitrary difference constraint $x - y \leq c$, resulting in an assignment $\mathcal{A}'$ such that $G_{\mathcal{A}'}$ contains no negative cycle. We simply let $\mathcal{A}' = \mathcal{A} \cup \{d^c_{xy} \mapsto \texttt{true}\}$ in case adding $x - y \leq c$ to $G_\mathcal{A}$ does not induce a negative cycle, and let $\mathcal{A}' = \mathcal{A} \cup \{d^c_{xy} \mapsto \texttt{false}\}$ otherwise. In this way, $\mathcal{A}$ may be extended until it is total. We call the resulting total truth assignment $\mathcal{A}^*$. Given $\mathcal{A}^*$, a satisfying assignment over the numeric variables in φ may be derived by finding a potential function π for $G_{\mathcal{A}^*}$, or equivalently solving the augmented SSSP problem for $G_{\mathcal{A}^*}$.

4.4 Conclusion

We have presented basic properties of Boolean combinations of difference constraints. Conjunctions of difference constraints are characterized by shortests paths in the corresponding constraint graph. A satisfying assignment for a system of difference constraints may be obtained by finding a potential function for the constraint graph. Difference constraints are closed under negation. Although satisfiablity of a Boolean combination of difference constraints is determinable by satisfying all the clauses, a partial truth assignment satisfying all the clauses is not so easily extendible as in the case for propositional logic. Nevertheless, a satisfying assignment can be readily constructed once all the clauses are satisfied.

Chapter 5

Model Based Interpretation

In this chapter we present an incremental model based implementation of a theory interpreter for the theory of difference constraints in the context of a generic DPLL solver as presented in chapter 3.

5.1 Overview

We first briefly recall the interface for a theory introduced and discussed in section 3.2. In short, an interpreter for a theory should support extension by interpretation of a single atom, backtracking, constraint propogation, and explanation of implied constraints.

The guiding principle we will use is a *model based* approach. In particular, we will maintain one model for each set of interpreted difference constraints. Each model consists of an assignment to the numeric variables which satisfies the set of interpreted difference constraints. The main motivations for using a model based approach are that

1. It uses less storage than previous approaches such as [19], which store one adjacency matrix representation of the constraint graph for each assignment.

2. It supports incremental extension that is far more efficient than the non incremental counterparts.

3. It provides a basis on which constraint propogation may be implemented efficiently.

5.2 Architecture

Globally speaking, the theory interface is driven by DPLL which maintains a stack of assigned literals, some of which may be difference constraints. The theory interface is asked to interpret difference constraints in the same order as they appear in the literal stack. Thus the difference constraints implicitly form a stack as well. In turn, each point in the stack of difference constraints induces a constraint graph consisting of all the constraints at that point or at a lower point in the stack. In this way, each point in the stack of difference constraints defines a constraint graph which is a subgraph of the constraint graph associated with any higher point in the stack.

We will maintain instead a stack of models. The first model M_0 will satisfy the first difference constraint c_0 and any subsequent interpreted difference constraints $c_1 \ldots c_n$ which happen to be satisfied by M_0 will not generate a new model. However, as soon as a difference constraint c is interpreted which is not satisfied by the model at the top of the model stack, a new model will be generated and pushed onto the model stack.

5.2.1 Stack Threaded Constraint Graphs

In general, we will often work with the constraint graph induced by some point in the model stack. This point will often, but not always, be the top of the stack. To support efficient extension, backtracking, and extraction of a constraint graph from an arbitrary point in the stack, we will thread the constraint graph through the model stack. In this section we detail how this is accomplished.

Each model M contains an assignment to the numeric variables present in its interpreted difference constraints M_C. The numeric variables are associated in one to one correspondence with the vertices of the constraint graph induced by M_C. We associate with each vertex a pointer into an adjacency list representation of the edges from the top of the stack.

As an example, we consider the case where the model stack is (M_0, M_1), the stack of interpreted constraints associated with M_1 coming from a vertex x is $M_1^x = ((x - y_0 \leq c_0), (x - y_1 \leq c_1), (x - y_2 \leq c_2))$, and the stack of constraints associated with M_0 coming from x is $M_0^x = ((x - y_0 \leq c_0), (x - y_1 \leq c_1))$. We keep the edge stack in a linked list of edges projected onto the positive variable in the constraint. The list as a whole is an adjacency list for the vertex associated with x in the model at the top of the stack. If we

associate a pointer to the list cell containing $(x - y_1 \leq c_1)$ with x in M_0, then we have direct access to an adjacency list representation of the constraint graph for x in M_0 as well. In this fashion, an adjacency list representation *of the edges* in the constraint graph associated with a given model can be accessed from the model directly as an edge adjacency list with zero copying of the constraints.

To complete the threading of the constraint graph through the stack, we will not follow the convention that edges store direct reference to their vertices. Instead, each edge will keep an index to its source and target vertices which is valid in all models. Thus in the context of a given model, one may retrieve the vertices associated with an edge efficiently. An arbitrary static ordering of the vertices shared by all the models can be used to provide an index which allows constant time access.

5.2.2 Uninterpreted Constraint Graph

In addition to the interpreted constraints, the uninterpreted constraints form a pool of constraints which might be candidates for implication and from which the DPLL procedure may select a truth value on which to branch. It turns out that a graphical treatment of these constraints also facilitates the inference process. Thus we will maintain a dual graph, consisting of vertices corresponding to all the numeric variables in the difference constraints and edges corresponding to all the uninterpreted difference constraints. We call this graph the uninterpreted constraint graph.

5.3 Interpretation and Backtracking

The interpretation of a new difference constraint consists of two cases. First if a new constraint is satisfied by the current model at the top of the model stack, then we simply add the constraint to the model as described above. If the new constraint is not satisfied by the current model, a new model is created or detection of unsatisfiability of the currently interpreted constraints with the new constraint is accomplished with a straightforward application of the incremental negative cycle detection algorithm presented in section 4.1.2. This algorithm requires that we store with each model the shortest path tree in the form of parent pointers together with a doubly linked list containing the vertices of the tree in a preorder traversal, as well as the degree of a

vertex in the shortest path tree.

Backtracking can be accomplished simply by removing edges from the stack threaded graph and putting them in the pool of unassigned constraints. If the removed edge is the one which triggered the generation of a new model, then we pop the model from the model stack. Since the unassignments happen in reverse order of the assignments, there is no need to keep track of decision levels in the interpretation stack and we are guaranteed to always remove constraints from the top of the constraint stack associated with its positive variable.

5.4 Constraint Propogation

In section 4.1, we established the characterization of systems of difference constraints by shortests paths in constraint graphs. It follows immediately from that discussion that an unassigned constraint $x - y \leq c$ is implied by a set of interpreted constraints if and only if the shortest path from x to y in the constraint graph associated with the interpreted constraints has weight less than or equal to c. However, computing the shortest paths for all the uninterpreted constraints for every assignment is prohibitively expensive. In this section we present a semi-lazy constraint propogation method whose goal is to minimize the overhead of numeric constraint propogation.

5.4.1 Round Robin Constraint Propogation

In a standard DPLL implementation, Boolean constraint propogation (BCP) maintains a queue of implied literals. Some literals in the queue may be implied multiple times, and the queue may not be empty when a conflict is found. From these two properties of the constraint propogation it follows that as long as some implication is queued for BCP to process, there is no need to find more numerical implications. In fact if work is done to find numerical implications, it may simply be wasted effort.

An alternative is to perform numerical constraint propogation incrementally, sending only a few implications for BCP at a time. Note that it is possible to do this without any loss of completeness in the constraint propogation, so long as the procedure always checks with the theory interpreter for more implications before deferring to a guess. However, in order to implement this alternative it is necessary to be able to stop and restart the

process of finding implied difference constraints while the underlying model is changing. This is because BCP may imply a constraint which changes the model at any time.

Example 5.4.1. Suppose that a problem contains the clauses

$$\begin{array}{ccc} (z-w \leq 2) & & \\ (y-z \leq 10) & & \\ (y-x \leq -2) & \vee & (y-z \leq 9) \\ (x-z \leq 9) & \vee & (z-w \leq 0) \\ (z-y \leq -8) & \vee & (w-z \leq -1) \\ (w-y \leq -12) & \vee & \ldots \end{array}$$

Suppose that the model at the top of the model stack has interpreted $(z-w \leq 2)$, $(y-z \leq 10)$, $(x-z \leq 9)$ and $(z-y \leq -8)$, and moreover that we queue implications of the form $(x-\_ \leq \_)$. Since $(x-z \leq 9) \wedge (z-y \leq -8)$ implies that $(x-y \leq 1)$, $(y-x \leq -2)$ is falsified. Then $(y-z \leq 9)$ is in a unit clause and so becomes interpreted. Suppose that we then queue implications of the form $(y-\_ \leq \_)$. Since $(z-w \leq 2) \wedge (y-z \leq 9)$ implies $(y-w \leq 11)$, $(w-y \leq -12)$ becomes falsified. Note that prior to the first unit propogation, we knew that $(y-z \leq 10)$ but afterwards there is a tighter bound on $y-z$ and this tighter bound falsified $(w-y \leq -12)$.

To implement this kind of constraint propogation for both complete and incomplete propogation, we use a variant of Johnson's all pairs shortest path algorithm [7]. The algorithm normally works in two stages, the first stage runs an augmented SSSP algorithm, some of which are described in chapter 4, and establishes a potential function. The second stage uses the potential function to run one instance of Dijkstra's algorithm per vertex on a modified graph with positive edge weights in which shortests paths are preserved[1]. In our case, since we maintain a stack of models incrementally, and models which satisfy the interpreted constraints also define potential functions, we can proceed directly with the second step.

The second step conveniently partitions the shortests paths into those originating from one vertex, or variable in a difference constraint, at a time.

[1]Given a potential function π on a graph with weighted edges $G = (V, E, W)$, the graph $G' = (V, E, W')$ with $W'(x, y) = \pi(x) + W(x, y) - \pi(y)$ has positive edge weights and the same set of shortests paths as G. In particular, the weight of a path from $(v_0 v_1 \ldots v_n)$ in G' is $\Sigma_i \pi(v_i) + W(v_i, v_{i+1}) - \pi(v_{i+1})$, or equivalently $\pi(v_0) - \pi(v_n) + \Sigma_i W(v_i, v_{i+1})$.

Thus we can easily process implications one variable at a time. In addition, we can continue to process different variables even when the underlying model changes, since the second step only requires that *some* potential function is established, and doesn't rely on the particular value of the potential function.

In both complete and incomplete forms of propogation, we will process the variables with Dijkstra's algorithm in a round-robin manner. In particular, we'll keep a list of *candidate* variables x which may occur in some implied constraint $x - y \leq c$. Whenever the coarse grained constraint propogation occurs we go through the list running Dijkstra's algorithm, treating each variable as the source vertex from which shortests paths are computed. This continues until either there is an implied constraint or no more variables remain. If there is an implied constraint, we remember the location in the list and let some BCP occur, possibly removing and adding candidate variables from the list. Once BCP is done, we resume processing the variables from the place we left off. The only differences between complete propogation and incomplete propogation are in the management of the candidate list and in how many variables in the list are processed in search of an implied constraint.

5.4.2 Complete Propogation

For complete propogation, we need to establish for every uninterpreted constraint $x - y \leq c$ whether or not there is path in the topmost interpreted constraint graph with weight less than or equal to c. If there is a such a path, the constraint is implied, otherwise the constraint is not implied. To accomplish this, we maintain in the candidate list all the variables x for which the following conditions hold:

1. The corresponding vertex in the uninterpreted constraint graph has positive degree.

2. The corresponding vertex in the interpreted constraint graph has positive degree.

3. Given the potential function π associated with the current model, there exists a constraint $x - y \leq c$ with $c \geq \pi(y) - \pi(x)$.

The list maintenance is done lazily over the complete set of variables: each variable is tested for these three conditions until a variable satisfying the

conditions is found. If a variable does satisfy these conditions, Dijkstra's algorithm is run in search of implied constraints. If the variable x does not satisfy these conditions, then there can be no implied constraint $x - y \leq c$. In particular, if x violates conditions 1 or 2, then either there are no uninterpreted constraints which might be implied or there is no path originating from x in the interpreted constraint graph which might imply some uninterpreted constraint. If the third condition is violated, then there are no constraints $x - y \leq c$ such that $c \geq \pi(y) - \pi(x)$. In this case, there cannot be a path from x to y with weight less than or equal to c since π is a valid potential function. Finally to ensure completeness, *every* variable may be checked in search of an implied constraint.

5.4.3 Incomplete Propogation

In the case of incomplete propogation, the choice of which variables are included in the search for an implied constraints becomes important: we'd like to include variables x which will likely lead to implied constraints $x - y \leq c$. We present here a heuristic for identifying such variables based on the evolution of models as the model stack grows. We begin with the observation that the potential function π associated with a model acts as an approximate measure of the shortest path distances in that there is no path from x to y shorter than $\pi(y) - \pi(x)$. Now as the model stack grows, the value of the potential function for some vertices decrease and no values increase. Given the set of vertices V^- whose potential decreases when a new model is generated and its pre-image $\text{pre}(V^-)$ in the uninterpreted constraint graph, the the estimated distance originating from any vertex in the pre-image $\text{pre}(V^-)$ to any vertex in V^- may decrease, and will certainly decrease for vertices $v \in \text{pre}(V^-) \setminus V^-$. We will focus on vertices taken from $\text{pre}(V^-)$.

Note that the set $\text{pre}(V^-)$ does not necessarily contain every variable from which implications may be found, since the addition of a constraint $x - y \leq c$ to a model M may reduce the length of many paths without reducing the length of any path in the shortest path tree. However, potential functions do estimate shortest path distances between arbitrary vertices in the sense that they place a lower bound on such paths.

To implement this strategy, we add a hook function to the incremental negative cycle detection algorithm. Whenever an edge is relaxed, reducing the potential of some vertex, we mark the vertex and add it to a queue if it was not already marked. Upon constraint propogation, we unmark

the marked vertices, find their pre-image in the uninterpreted constraint graph, and add these variables to the candidate list if they are not already there. The candidate list is then filtered by the same conditions used in complete propogation, described above. Since the propogation is incomplete, a configurable number of variables may be checked in search of an implied constraint, for both fine and coarse grained propogation. However, variables are removed from the candidate list as they are checked, making the list effectively a queue.

5.4.4 Complexity

Constraint propogation depends on the generation of a new model, which invokes the incremental negative cycle detection algorithm described in 4. This algorithm runs in $\mathcal{O}(|V| \log |V| + |E|)$ time on the interpreted constraint graph. For incomplete propogation, the hook function in the uninterpreted constraint graph embedded in this algorithm takes constant time. The pre-image of affected sets takes $\mathcal{O}(|V| + |E|)$ time in the worst case and is proportional to the subgraph induced by the vertices whose potential has changed in the model generation. Dijkstra's algorithm is of course $\mathcal{O}(|V| \log |V| + |E|)$ when used with a Fibonacci heap. Our strategy runs Dijkstra's algorithm once for every model generation and a configurable number of times when the BCP queue is empty. In the former case, we get negative cycle detection *and* constraint propogation in $\mathcal{O}(|V| \log |V| + |E|)$ time in the size of the interpreted constraint graph. In the later case, the complexity does not change if the paramater is small enough in comparison $|V|$, but complete propogation takes $\mathcal{O}(|V|(|V| \log |V| + |E|))$ time since every vertex must be processed.

5.5 Conclusion

We have presented an incremental, model based implementation of a theory interpreter for determining the satisfiability of a Boolean combination of difference constraints. We presented a round robbin approach to constraint propogation for the theory of difference constraints, which can be implemented for complete or incomplete propogation and allows for negative cycle detection together with a substantial amount of incomplete inference in $\mathcal{O}(|V| \log |V| + |E|)$ time. This is a substantial improvement over the oft

used method of a single source shortest path algorithms for negative cycle detection without inference, which takes $\mathcal{O}(|V||E|)$ time. It also provides an alternative to methods that use Floyd and Warshall's $\mathcal{O}(|V|^3)$ all pairs shortests paths or its $\mathcal{O}(|V|^2)$ incremental variation for constraint propogation. In particular, the model based approach presented utilizes far less space and has better asymptotic time bounds for sparse graphs.

Chapter 6

Experiments

In this chapter we present the results of some experiments.

6.1 Job Shop Scheduling

Job shop scheduling problems present a class problems which strongly exercise the constraint propogation for difference constraints and at the same time offer an easily generated spectrum in the degree of constrainedness, ranging from the far from satisfiable to the very easily satisfiable. Although the problems are highly disjunctive, the constrainedness is almost entirely numerical. For this reason, we examine job problems with respect to different configurations for numerical constraint propogation.

6.1.1 Problem Description

The classical problem of job shop scheduling can be solved with the aid of a satisfiability solver for difference logic. A job shop problem consists of a set of jobs J and a set of machines M. Each job is a sequence of pairs (m, d) with $m \in M$ and $d \in \mathbb{R}$. Each pair denotes a task t, indicating that machine m is used exclusively for a total of d time units. The *makespan* of a schedule is the total time to complete all the jobs. The scheduling problem is to find a schedule for the optimal makespan of the problem.

To model a job shop problem with a Boolean combination of difference constraints, we create a variable e indicating the earliest time at which any task may start and a variable $s(t)$ for each task t in the problem, indicating

the time at which t starts. We then encode the ordering constraints of each job j with $s(t_{i+1}) - s(t_i) \geq d(t_i)$ where t_i indicates the ith task in j and $d(t)$ indicates the duration associated with the task. For the first task of each job, we encode that it must start after e with $s(t_1) - e \geq 0$. These constraints may readily be translated to proper difference constraints with the equivalence $x - y \geq c \leftrightarrow y - x \leq -c$. We call the conjunction of these constraint for each job S.

We then encode exclusive access to a given machine m at a time as follows. Let $T(m)$ be the set of tasks using machine m. For each pair of tasks t, t' taken from $T(m)$, we require that

$$(s(t) - s(t') \geq d(t')) \vee (s(t') - s(t) \geq d(t))$$

indicating that only one task in $T(m)$ can be executed at a time. We refer to the conjunction of all such disjuncts as X. These constraints can readily be translated to proper difference constraints as before.

Finally, we encode a given makespan D as follows. For the last task t_j of each job j, we encode finishing within the makespan as

$$s(t_j) - e \leq D - d(t_j)$$

We refer to the conjunction of these formulas as F.

It is clear that $S \wedge X \wedge F$ is satisfiable if and only if there is a schedule of length D for the problem. To find an optimal schedule, we can perform binary search over an appropriate interval for D. At each point in the search, a SAT problem for difference constraints will direct the search to a longer or shorter value for D.

6.1.2 Experimental Analysis

We performed experiments on two job shop scheduling problems taken from a suite of operations research benchmark problems [15] for which the optimum is known. For each problem, we give run times for several individual SAT instances where D varies around and including the optimum. In addition, we replicate these experiments under different configurations of our prototype implementation. In particular, we run the experiments without constraint propogation, and with both incomplete and complete constraint propogation as described in chapter 5. Each experiment was run on a Dell laptop with a 3 Gigahertz P4 processor and a gigabyte of RAM. For these experiments,

all problems occupied less than 128 megs of RAM and it is clear that CPU is the bottleneck.

For each SAT problem we give the number of decisions (guessed literals), the total run time [1], the number of times an implied difference constraint was found by numerical constraint propogation, and the result.

The first problem, FT06, is handled easily by our prototype solver. Nonetheless, it is evident that more work is required around the optimum. The solver without constraint propogation makes many more decisions than those with constraint propogation. While those with constraint propogation run a bit faster, the difference is negligable. However, these problems do not exercise the DPLL procedure enough for differences in configuration to become clear.

The second problem, ABZ5 [1], is a 10 job, 10 machine problem. To our knowledge, this problem has never before been solved by means of SAT, although efforts have been made [19, 8]. In this problem, the increased difficulty around the optimum is more evident. The role of numerical constraint propogation is also more evident.

Comparing the configurations without constraint propogation and with incomplete constraint propogation, we find that the incomplete constraint propogation is unilaterally faster, and that the difference between it and the configuration without propogation becomes markedly more pronounced on the harder problems closer to the optimum.

On the other hand, in this problem it becomes clear that our method for complete propogation is inferior to our method for incomplete propogation and even sometimes inferior to our method without propogation. With complete propogation, the rate at which the solver makes decision decreases drastically while the average number of implications the solver makes per decision is often only slightly greater than that for incomplete propogation, and sometimes even less. The fact that complete propogation can produce fewer implications per decision than incomplete propogation is counterintuitive, but differing sets of implications lead to different sets of decisions making this phenomenon entirely possible.

Despite the fact that these problems are more difficult at the optimum, they do not display a clear curve indicating the run time as a function of

[1] We report run times of the DPLL procedure, rather than the total program run time which would also including parsing and CNF translation, as well as initialization of a JVM (The prototype is in Java). These later components occupied negligable amount of wall clock time in all cases: the DPLL run time is always with in .2 seconds of the total program run time.

Figure 6.1: FT06 without numerical constraint propogation.

makespan	Time (Seconds)	Decisions	Implications	Result
50	0.08	98	0	unsat
51	0.05	157	0	unsat
52	0.11	659	0	unsat
53	0.13	1341	0	unsat
54	0.11	1629	0	unsat
55	0.21	3257	0	sat
56	0.23	3326	0	sat
57	0.12	2171	0	sat
58	0.13	2608	0	sat
59	0.03	485	0	sat
60	0.04	662	0	sat

distance from the optimum. In general, SAT solvers' time can vary wildly as a result of variable ordering and also the order in which constraint propogation proceeds. This high variance makes analysis of the results more difficult as one can only make conclusions based on tendencies in the results which can easily be obfuscated by deviations in performance arising from heuristics or lack of thorough testing.

Figure 6.2: FT06 with incomplete numerical constraint propogation.

makespan	Time (Seconds)	Decisions	Implications	Result
50	0.04	0	54	unsat
51	0.05	14	98	unsat
52	0.04	28	139	unsat
53	0.08	34	225	unsat
54	0.25	193	1287	unsat
55	0.04	100	49	sat
56	0.21	310	1613	sat
57	0.03	95	289	sat
58	0.02	46	168	sat
59	0.01	60	108	sat
60	0.03	107	52	sat

Figure 6.3: FT06 with complete numerical constraint propogation.

makespan	Time (Seconds)	Decisions	Implications	Result
50	0.05	0	80	unsat
51	0.06	2	94	unsat
52	0.06	3	109	unsat
53	0.08	7	156	unsat
54	0.26	28	381	unsat
55	0.12	30	205	sat
56	0.06	12	178	sat
57	0.05	10	118	sat
58	0.05	12	113	sat
59	0.05	15	114	sat
60	0.05	17	116	sat

Figure 6.4: ABZ5 with no numerical constraint propogation.

makespan	Time (Seconds)	Decisions	Implications	Result
900	0.03	69	0	unsat
1000	0.75	2040	0	unsat
1100	6.43	23328	0	unsat
1150	88.44	96590	0	unsat
1200	550.58	305078	0	unsat
1232	>1000.00	474697	0	-
1233	>1000.00	329344	0	-
1234	>1000.00	480566	0	-
1235	510.67	318778	0	sat
1250	421.38	306402	0	sat
1300	229.01	218456	0	sat
1350	191.29	234492	0	sat
1400	40.68	72015	0	sat

Figure 6.5: ABZ5 with incomplete numerical constraint propogation.

makespan	Time (Seconds)	Decisions	Implications	Result
900	0.01	0	89	unsat
1000	0.27	30	594	unsat
1100	1.99	540	6932	unsat
1150	4.23	1294	875	unsat
1200	80.34	16471	11983	unsat
1232	300.67	44738	31297	unsat
1233	378.24	49883	35408	unsat
1234	120.80	17041	11633	sat
1235	110.16	16015	10970	sat
1250	154.27	22334	15211	sat
1300	21.04	3757	2435	sat
1350	4.86	990	12538	sat
1400	3.56	1342	347	sat

Figure 6.6: ABZ5 with complete numerical constraint propogation.

makespan	Time (Seconds)	Decisions	Implications	Result
900	0.03	0	652	unsat
1000	1.52	16	816	unsat
1100	9.63	99	6609	unsat
1150	46.50	517	31551	unsat
1200	562.73	3603	992	unsat
1232	>1000	6410	1846	-
1233	>1000	5810	1730	-
1234	>1000	6035	1984	-
1235	>1000	5755	1788	-
1250	335.33	1946	524	sat
1300	37.35	318	9243	sat
1350	12.95	130	2241	sat
1400	2.00	40	672	sat

6.2 Bounded Model Checking of Timed Automata

Bounded Model Checking (BMC) is the process of model checking a transition system unwound to some finite bound. For a given bound, the process may not be complete; however, a bug may be found. In this way, BMC can be used to find bugs in systems which cannot otherwise be verified. BMC has enjoyed some industrial success for discrete transition systems. As a result, BMC for timed and hybrid systems has been the object of much research [24, 29, 35]. In this section we will present some experiments implementing bounded model checking for a class of timed automata [2].

6.2.1 Definitions

In this section we give the formal definitions required to describe a problem on circuit timing analysis.

Definition 6.2.1. Timed Automaton
Let C be a set of variables ranging over integers, also called clocks. Let $L(C)$

be the set of constraints of the form $x \leq c$, $x \geq c$ for $x \in c$ and $c \in \mathbb{Z}$, closed under conjunction. For our purposes, a *timed automaton* is a tuple $\mathcal{A} = (Q, S, T, C)$ where Q is a finite set of states, $S : Q \to L(C)$ maps states to staying conditions, and $T \subseteq Q \times Q \times 2^{L(C)} \times 2^C$ is a finite discrete transition relation.

A state of a timed automaton $\mathcal{A}$ is a pair (q, v) where $q \in Q$ and $v : C \to \mathbb{Z}$ is a valuation of the clock variables. A discrete transition $(q, q', g, r) \in T$ represents a transition from (q, v) to (q', v') where $v(C) \models g$ and $v'(c) = v(c)$ whenever $c \notin r$ and $v'(c) = 0$ otherwise. We sometimes write $r(v)$ for v' defined in this way.

Definition 6.2.2. Run
A *run* of a timed automaton $\mathcal{A}$ is a sequence of states $\langle (q_0, v_0), (q_1, v_1), \ldots \rangle$ such that for every i, $v_i(C) \models S(q_i)$, and such that every pair of adjacent states $(q_i, v_i)(q_{i+1}, v_{i+1})$, one of the following holds.

1. There is a $(q_i, q_{i+1}, g, r) \in T$ such that $v_i(C) \models g$ and $v_{i+1} = r(v_i)$.
2. There is a $t \geq 0$ such that $v_{i+1}(C) = t \cdot v_i(C)$ and $q_i = q_{i+1}$.

We call condition 1 a discrete transition and condition 2 a time transition. We say a run is *minimal* if it alternates between discrete and time transitions.

Definition 6.2.3. Product Timed Automaton
The product of timed automata $\mathcal{A}_i = (Q_i, S_i, T_i, C_i)$ for $i \in I$ is (Q, S, T, C) where

$$\begin{aligned} Q &= \Pi_{i \in I} Q_i \\ S(\bar{q}) &= \bigwedge_{i \in I} S_i(q_i) \\ T &= \{(\{q_i\}, \{q_i'\}, \{g_i\}, \{r_i\}) \mid i \in I \text{ and } (q_i, q_i', g_i, r_i) \in T_i \text{ for every } i \in I\} \\ C &= \bigcup_{i \in I} C_i \end{aligned}$$

Definition 6.2.4. Unwinding
The *k-unwinding* of a timed automaton $\mathcal{A}$ is the set of all minimal runs of length k.

6.2.2 Circuit Timing Analysis

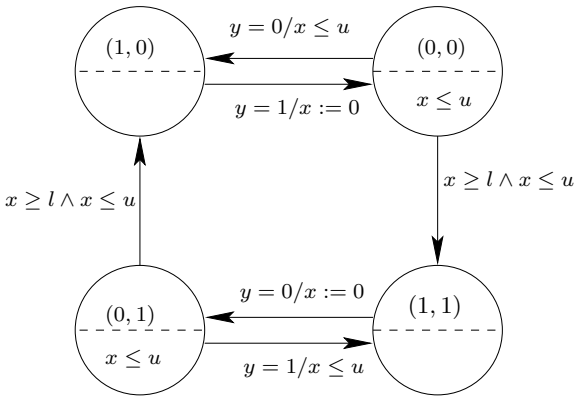
Following [20] We consider a circuit to be an acyclic network of gates (V, E, F, I) in which each gate $g \in V$ computes a boolean function $F(g)$ over its inputs $In(g) = \{v \in V \mid (v, g) \in E\}$. In this way, every gate without any inputs acts as a variable or input to the circuit as a whole. We are interested in analysing the timed behavior of a circuit given some change in its inputs. In particular, we consider that each gate g will take some time to propogate a change in its inputs to its output and that moreover this time must fall within a specified interval $I(g) = [l_g, u_g]$. Note that in this model, it is possible that a temporary change in the inputs of some gate may not cause the gate to propogate the change to its output.

To model the possible behaviors of a such a circuit, we will treat each gate as a single timed automaton with one clock. Such an automaton is pictured in figure 6.7. For the sake of explication, suppose the automaton in the figure represents a gate g. The states of the automaton consist of tuples $(\texttt{stable}, y)$ where `stable` is a bit indicating whether the automaton has propogated its inputs to its output y. A transition from a stable state to an unstable state represents the situation in which the input changes from $In(g)$ to $In'(g)$ such that $F(g)(In(g)) \neq F(g)(In'(g))$. At the moment this event occurs, the gate's clock x is reset to zero and the automaton switches to an unstable state. The automaton can only remain in an unstable state for at most u_g time. In addition, as long as $0 \leq x \leq u_g$, the input to the gate may change again to $In''(g)$ such that $F(g)(In''(g)) = F(g)(In(g))$. If this occurs while the automaton is an unstable state, the automaton will regret to its previous state. This action is represented in the figure by the two horizontal transitions labelled with constraints $x \leq u$. On the other hand, if the automaton does not regret, a transition to a stable state will occur. This transition represents the event that g has propogated its input to its output and is represented in the figure by the two vertical transitions.

To construct a single timed automaton representing an entire circuit, we will first augment the staying conditions for each state in the automata for a gate g with the Boolean function $F(g)$. Then we take the product automaton of all the gates in the circuit. The required synchronization of the product naturally follows by variable sharing between the automata in the staying conditions.

To perform BMC on such an automaton, we will unwind the automaton to some bounded depth and encode the set of paths in the unwound automaton

Figure 6.7: Timed automaton for a gate.



into a formula consisting of a Boolean combination of difference constraints and propositional variables. With such a formula φ in hand, we can then hunt for a violation of a property P over the runs by checking the satisfiability of the formula $\varphi \wedge \neg P$.

6.2.3 Coding the BMC problem

Following [24], we encode the k unwinding of the product automaton over a set of gates G as

$$\bigwedge_{0 \le i \le k} \bigwedge_{g \in G} \bigvee_{t \in T_g} \varphi(t, i)$$

where T_g is the set of transitions defined for gate g.

We describe how to define transitions $\varphi(t, i)$ for the automaton associated with gate g. We will index the variables `stable` and y by the unwinding depth. For notational convenience, we write $s^+(i)$ for $\neg\texttt{stable}_i \wedge \texttt{stable}_{i+1}$, $s^-(i)$ as $\texttt{stable}_i \wedge \neg\texttt{stable}_{i+1}$, and $s^=(i)$ for $\texttt{stable}_i \leftrightarrow \texttt{stable}_{i+1}$. We write $y^=(i)$ for $y_i \leftrightarrow y_{i+1}$ and $y^{\neq}(i)$ for $y_i \not\leftrightarrow y_{i+1}$. We also write $F_i(g)$ for $F(g)$ with all variables x replaced with their respective indexed counterparts x_i. Finally we write $f^=(i)$ for $F_i(g) \leftrightarrow F_{i+1}(g)$, and similarly $f^{\neq}(i)$ for $F_i(g) \not\leftrightarrow F_{i+1}(g)$. Letting x be the clock associated with automaton for gate g, we can now define the transitions:

- Excite Transition:

$$s^{-}(i) \wedge y^{=} \wedge f^{\neq}(i) \wedge x_{i+1} = 0$$

- Regret Transition:

$$s^{+}(i) \wedge y^{=} \wedge f^{\neq}(i) \wedge x_i \leq u_g \wedge x_{i+1} = x_i$$

- Stabilize Transition:

$$s^{+}(i) \wedge y^{\neq} \wedge f^{=}(i) \wedge x_i \leq u_g \wedge x_i \geq l_g \wedge x_{i+1} = x_i$$

- Time Transition:

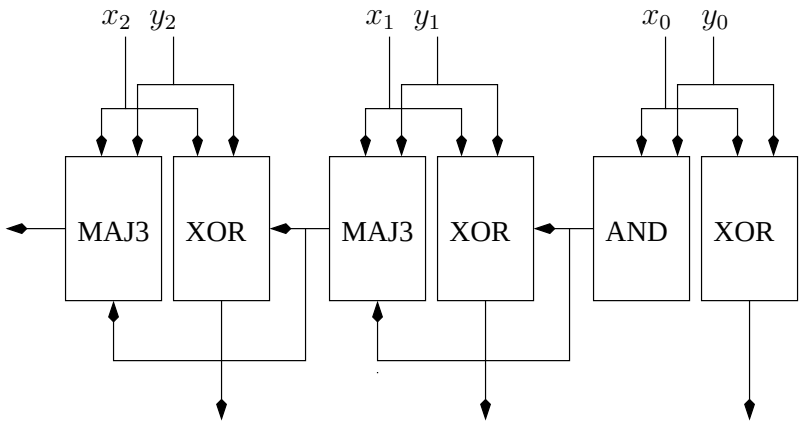
$$s^{=}(i) \wedge y^{=} \wedge f^{=}(i) \wedge \exists t \geq 0.\ x_{i+1} = t + x_i$$

In order to encode this formula as a Boolean combination of propositional variables and difference constraints, we need to express all the timing constraints as difference constraints. This is accomplished by introducing a sequence of variables $g_0, g_1, \ldots g_k$ reflecting the values of a global clock which is never reset. For clock resets, we require $x_i = g_i$ instead of $x_i = 0$. For all other transitions, we require $x_i = x_{i+1}$. To manage the global clock, we add the constraint $g_i = g_{i+1}$ for discrete transitions and $g_i \leq g_{i+1}$ for time transitions. Under this scheme, the value of a local clock x at unwinding depth i is $g_i - x_i$. Hence all temporal staying conditions and transition guards are expressed as $g_i - x_i \leq u$ or $g_i - x_i \geq l$. Finally, using the equivalences $x = y \leftrightarrow x - y \leq 0 \wedge y - x \leq 0$, $x - y \geq c \leftrightarrow y - x \leq -c$, and $x \leq y \leftrightarrow y - x \leq 0$, we can express every timing constraint as a conjunction of difference constraints.

6.2.4 Experiments

The problem we address for circuit timing analysis is that of maximum stabilization time. In particular, given a circuit (V, E, F, I), we wish to find the maximum time required for the circuit to go from any state in which all the outputs are stable to any other state in which all the outputs are stable. We assume the inputs to the circuit start in one state and then immediately change to some other value which is fixed indefinitely, however the particular values of the inputs are not assumed. The query "is there a minimal run of length k which leaves some circuit output in an unstable state after d time

Figure 6.8: A 3-bit adder.



units" is coded as a $\varphi \wedge g_k \geq d \wedge (\bigvee_{v \in V} \neg \texttt{stable}_k^v)$ where φ is a Boolean combination of difference constraints describing the runs of length k as described above, and $\texttt{stable}_k^v$ describes the stability of gate v at step k. If k is long enough to propogate some input to some output, and the answer is satisfiable, then the maximum stabilization time exceeds d. Similarly, if k is sufficiently long and the answer is unsatisfiable, then d is an upper bound on the maximum stabilization.

We ran such queries on n-bit adders, each of which takes $2n$ input bits $(\bar{x}, \bar{y})$ and adds them together to form n outputs. The bit adders are comprised of one half-adder and $n - 1$ full-adders. Both half adders and full adders contain 2 gates, one of which determines an output bit and the other of which determines a carry bit. Full adders take as input three bits one taken from $\bar{x}$ another from $\bar{y}$, and the third from the carry bit of the previous adder. Half adders take two bits taken from $\bar{x}$ and $\bar{y}$ as input. Figure 6.8 shows a 3-bit adder.

6.2.5 Analysis

Figure 6.9 gives a table of execution times for various values of n, k, and d. The results indicate that the method is less scalable in the depth of the unwinding than in the size of the circuit. Even so, the size of the largest circuit we treated is miniscule by industrial standards, only 20 gates. On the

Figure 6.9: Maximum stabilization query results.

NBits	Unwinding (k)	Duration	nvars/bvars/clauses	Time (Secs)	Result
2	8	12	45/740/2048	0.78	sat
2	8	13	45/740/2048	0.54	unsat
3	12	18	91/1722/4943	7.09	sat
3	12	19	91/1722/4943	6.50	unsat
4	16	24	153/3112/9086	64.32	sat
4	16	25	153/3112/9086	39.90	unsat
5	20	30	231/4910/14477	305.62	sat
5	20	31	231/4910/14477	406.82	unsat
6	10	30	143/3095/8831	57.86	sat
6	10	31	143/3095/8831	61.47	unsat
7	10	30	165/3565/10400	128.68	sat
7	10	31	165/3565/10400	14.09	unsat
8	10	30	187/4095/11969	143.76	sat
8	10	31	187/4095/11969	119.82	unsat
9	10	30	209/4625/13538	124.81	sat
9	10	31	209/4625/13538	20.85	unsat
10	10	30	231/5155/15107	688.70	sat
10	10	31	231/5155/15107	211.67	unsat
10	12	36	273/5918/15588	929.10	sat
10	12	37	273/5918/15588	186.36	unsat
10	14	42	315/6871/18169	2063.75	sat
10	14	43	315/6871/18169	794.45	unsat

other hand, it is good to see the solver can figure out which of the 2^{40} pairs of inputs lead to long paths. Additionally, we see in the last four queries that as k grows, the stabilization time can grow as well. In our example, we gave each gate a lower bound of 4 time units and an upper bound of 6 time units, thus including regret transitions in the search for a longest path. If each gate never regrets, then the unfolding depth can allow for two discrete transitions and atleast one time transition for a single gate to propogate input to output. When gates are chained together, the stabilization of one gate will simultaneously coincide with the excitation of the next gate, thus allowing for the propogation through chains of length n within $2n$ steps. With regret transitions, the longest path in the automaton can grow much longer. In this toy bit adder example, regrets never play a role in the longest path.

It is in addition clear that the solver finds it more difficult to find satisfying assignments than it does to prove the unsatisfiability of a formula. This is in contrast to the job shop problems, where the unsatisfiable problems tended to take a bit longer. In an effort to find the cause of this phenomenon, we tried many of the same problems in table 6.9, with all input values fixed and delays deterministic. The runtimes for this easier problem were not in general very different from the problem presented here. As a result, we believe that the coding of the BMC problem for timed automata has room for improvement.

Chapter 7

Conclusion

Difference constraints form an interesting class of formula for study within the framework of satisfiability checking. While they admit much more efficient algorithms for satisfiability and inference than general linear constraints, embedding these algorithms within the DPLL procedure remains a delicate matter. We have presented a novel method of doing just that which appears to work relatively well in practice.

Several other methods for this problem have been proposed and used [30, 22, 27, 29, 35, 19]. Our method is unique in that it uses a model based approach and is one of the few methods which incorporate numerical constraint propogation into the DPLL procedure. The model based approach in turn allows for efficient constraint propogation in sparse systems of difference constraints, which is the norm in all the problems we have encountered. In addition, the model based approach supports incomplete numerical constraint propogation by providing a basis for an effective heuristic for choosing numeric variables which are candidates for constraint propogation. As a result, significant amounts of constraint propogation occur without increasing the asymptotic run time of the theory interpreter beyond what is required to detect inconsistent sets of assignments incrementally.

In the future, we would like to explore methods of combining different theories into the DPLL procedure. This is in particular motivated by the circuit stabilization problems, where the translation to SAT involved many equalities. If the theory of equalities could be combined effectively with the theory of difference constraints, such problems may become much more tractable. In addition, it would be interesting to combine the theory of difference constraints with more general classes of linear constraints so that

problems consisting of many difference constraints and a few general linear constraints may be more effectively solved. Finally, all the problems that we treated experimentally are optimization problems formulated in a way which uses SAT as a subprocedure. It would be worthwhile to treat these problems in a more unified fashion, perhaps by reusing appropriate learned clauses accross instances, applying interpolation, or lifting other SAT based BMC techniques for discrete problems to numerical SAT problems.

I hereby declare that this thesis is entirely my own work and that I have not used any other media than those mentioned in this thesis.
May 13, 2005

Bibliography

[1] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34:391–401, 1988.

[2] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation into sat. In *14th International Conference on Computer Aided Verification (CAV)*, 2002.

[4] R. Bayardo and R. Shrag. Using csp look-back techniques to solve real-worl sat instances. In *Proceedings of the 14th National Conference on Artificial Intelligence*, 1997.

[5] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[6] Boris V. Cherkassky and Andrew 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.

[7] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[8] Scott Cotton, Oded Maler, Eugene Asarin, and Peter Niebert. Some progress in satisfiability checking for difference logic. In *FORMATS '04*, 2004.

[9] Alexandre David, John Håkansson, Kim G. Larsen, and Paul Petterson. Minimal dbm substraction. In Paul Petterson and Wang Yi, editors, *Nordic Workshop on Programming Theory*, number 2004-041 in IT Technical Report of Uppsala University, pages 17–20, October 2004.

[10] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Cmmunications of the ACM*, 5:394–397, 1962.

[11] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In *European Symposium on Algorithms*, pages 320–331, 1998.

[12] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll(t): Fast decision procedures. In *Computer Aided Verification*, 2004.

[13] A. V. Goldberg and T. Radzik. A heuristic improvement of the bellmanford algorithm. In *Applied Mathematics Letters, 6*, 1993.

[14] E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE'02)*, pages 142–149, 2002.

[15] A.S. Jain, J. Pearson, C. Weise, and W. Yi. Deterministic job shop scheduling: Past, present and future. *European Journal of Operations Research*, 113:390–434, 1999.

[16] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[17] J.P.Marques-Silva and K.A. Sakallah. Grasp-a search algorithm for propostional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[18] Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock difference diagrams. *Nordic Journal of Computing*, 6:271–298, 1999.

[19] Moez Mahfoudh. *Sur la Vérification de la Satisfaction pour la Logique des Différences*. PhD thesis, Université Joseph Fourier, 2003.

[20] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In *CHARME'95*, volume LNCS 987, pages 189–205, 1995.

[21] M.Bozzano, R.Bruttomesso, A.Cimatti, T.Junttila, P.v.Rossum, S.Schulz, and R.Sebastiani. Mathsat: Tight integration of sat and mathematical decision procedures. *Journal of Automated Reasoning*, Special Issue on SAT, 2005.

[22] Jesper Møller and Jakob Lichtenberg. Difference decision diagrams. Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, August 1998.

[23] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the* 39^{th} *Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[24] Peter Niebert, Moez Mahfoudh, Eugene Asarin, Marius Bozga, Oded Maler, and Navendu Jain. Verification of timed automata via satisfiability checking. In *FTRTFT'02, LNCS 2469*, pages 225–244. Springer, 2002.

[25] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. In *Proceedings of the DIMACS Callenge II Workshop*, 1993.

[26] S. Seshia and R. Bryant. Deciding quantifier-free presburger formulas using parameterized solution bounds, 2004.

[27] H. M. Sheini and K. A. Sakallah. A sat-based decision procedure for mixed logical/integer linear problems. In *AIOR*, 2005.

[28] J.P. Marques Silva and K. A. Sakallah. GRASP-a search algorithm for propositional satisfiability. In *IEEE Transactions on Computers*, volume 48, pages 506–521, 1999.

[29] M. Sorea. Bounded model checking for timed automata. In *In Proceedings of MTCS*, 2002.

[30] O. Strichman, S.A. Seisha, and R.E. Bryant. Deciding separation formulas with sat. In *14th International Conference on Computer Aided Verification (CAV)*, 2002.

[31] R. E. Tarjan. Shortests paths. Technical report, AT&T Bell Laboraties, 1981.

[32] C. Tinelli. A dpll based calculus for ground satisfiability modulo theories. In *8th European Conference on logics in artificial intelligence.*, 2002.

[33] 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.

[34] J. M. Wilson. Compact normal forms in propositional logic and integer programming formulations. In *Computers and Operation Research*, pages 309–314, 1990.

[35] B. Woznia, A. Zbrzezny, and W. Penczek. Checking reachability properties for timed automata via sat. *Fundamenta Informatica*, 55:223–241, 2003.

[36] S. Yovine. Kronos: A verification tool for real time systems. *International Journal of Software Tools for Technology Transfer*, 1:123–133, 1997.

[37] H. Zhang and M. Stickel. An efficient algorithm for unit propogation. In *In Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics*, 1996.