SAT Modulo Monotonic Theories

by

Sam Bayless

B. Sc., The University of British Columbia, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

March 2017

Abstract

Satisfiability Modulo Theories (SMT) solvers are a class of efficient constraint solvers which form integral parts of many algorithms. Over the years, dozens of different Satisfiability Modulo Theories solvers have been developed, supporting dozens of different logics. However, there are still many important applications for which specialized SMT solvers have not yet been developed.

We develop a framework for easily building efficient SMT solvers for previously unsupported logics. Our techniques apply to a wide class of logics which we call *monotonic theories*, which include many important elements of graph theory and automata theory.

Using this *SAT Modulo Monotonic Theories* framework, we created a new SMT solver, MONOSAT. We demonstrate that MONOSAT improves the state of the art across a wide body of applications, ranging from circuit layout and data center management to protocol synthesis — and even to video game design.

Preface

The work in this thesis was developed in a collaboration between the Integrated Systems Design Laboratory (ISD) and the Bioinformatics and Empirical & Theoretical Algorithmics Laboratory (BETA), at the University of British Columbia.

This thesis includes elements of several published and unpublished works. In addition to my own contributions, each of these works includes significant contributions from my co-authors, as detailed below.

Significant parts of the material found in chapters 3, 4, 5, 7, and 6.1 appeared previously at the conference of the Association for the Advancement of Artificial Intelligence, 2015, with co-authors N. Bayless, H. H. Hoos, and A. J. Hu. I was the lead investigator in this work, responsible for the initial concept and its execution, for conducting all experiments, and for writing the manuscript.

Significant parts of the material found in Chapter 8 appeared previously at the International Conference on Computer Aided Verification, 2016, with co-authors T. Klenze and A. J. Hu. In this work, Tobias Klenze was the lead investigator; I also made significant contributions throughout, including playing a supporting role in the implementation. I was also responsible for conducting all experiments.

Significant parts of Section 6.2 were previously published at the International Conference on ComputerAided Design, 2016, with co-authors H. H. Hoos, and A. J. Hu. In this work, I was the lead investigator, responsible for conducting all experiments, and for writing the manuscript. This work also included assistance and insights from Jacob Bayless.

Finally, Section 6.3 is based on unpublished work in collaboration with co-authors N. Kodirov, I. Beschastnikh, H. H. Hoos, and A. J. Hu, in which I was the lead investigator, along with Nodir Kodirov. In this work, I was responsible for the implementation, as well as conducting all experiments and writing the manuscript.

Table of Contents

List of Tables

List of Figures

Acknowledgments

Alan and Holger, for giving me both the freedom and the support that I needed. The students I have collaborated with, including Celina, Mike, Nodir, Tobias, Kuba, and many others who have passed through our lab, for being a continuing source of inspiration. My parents and siblings, for everything else.

And the many shelves worth of science fiction that got me through it all....

Chapter 1

Introduction

Constraint solvers have widespread applications in all areas of computer science, and can be found in products ranging from drafting tools [18] to user interfaces [20]. Intuitively, constraint solvers search for a solution to a logical formula, with different constraint solvers using different reasoning methods and supporting different logics.

One particular family of constraint solvers, Boolean satisfiability (SAT) solvers, has made huge strides in the last two decades. Fast SAT solvers now form the core engines of many other important algorithms, with applications to AI planning (e.g., [138, 174]), hardware verification (e.g., [35, 40, 148]), software verification (e.g., [50, 57]), and even to program synthesis (e.g., [136, 184]).

Building on this success, the introduction of a plethora of Satisfiability Modulo Theories (SMT) solvers (see, e.g., [44, 70, 81, 192]) has allowed SAT solvers to efficiently solve problems from many new domains, most notably from arithmetic (e.g., [80, 100, 190]) and data structures (e.g., [29, 173, 191]). In fact, since their gradual formalization in the 1990's and 2000's, SMT solvers — and, in particular, *lazy* SMT solvers, which delay encoding the formula into Boolean logic — have been introduced for dozens of logics, greatly expanding the scope of problems that SAT solvers can effectively tackle in practice.

Unfortunately, many important problem domains, including elements of graph and automata theory, do not yet have dedicated SMT solvers. In Chapter 2, we review the literature on SAT and SMT solvers, and the challenges involved in designing SMT solvers for previously unsupported theories.

We have identified a class of theories for which one can create efficient SMT solvers with relative ease. These theories, which we term *monotonic theories*, are integral to many problem domains with major real-world applications in computer networks, computer-aided design, and protocol synthesis. In Chapter 3, we formally define what constitutes a monotonic theory, and in Chapter 4 we describe the *SAT Modulo Monotonic Theories* (SMMT) framework, a comprehensive set of techniques for building efficient lazy SMT solvers for monotonic theories. (Further practical implementation details can be found in Appendix A.)

As we will show, important properties from graph theory (Chapters 5, 6), geometry (Chapter 7), and automata theory (Chapter 8), among many others, can be modeled as monotonic theories and solved efficiently in practice. Using the SMMT framework, we have implemented a lazy SMT solver (MONOSAT) supporting these theories, with greatly improved performance over a range of important applications, from circuit layout and data center management to protocol synthesis — and even to video game design.

We do not claim that solvers built using our techniques are *always* the best approach to solving monotonic theories — to the contrary, we can demonstrate at least some cases where there exist dedicated solvers for monotonic theories that out-perform our generic approach.[1] However, this thesis does make three central claims:

1. A wide range of important, useful monotonic theories exist.

2. Many of these theories can be better solved using the constraint solver described in this thesis than by any previous constraint solver.

3. The SAT Modulo Monotonic Theories framework described in this thesis can be used to build efficient constraint solvers.

[1] Specifically, in Chapter 5 we discuss some applications in which our graph theory solver is outperformed by other solvers. More generally, many arithmetic theories can be modeled as monotonic theories, for which there already exist highly effective SMT solvers against which our approach is not competitive. Examples include linear arithmetic, difference logic, and, as we will discuss later, pseudo-Boolean constraints.

Chapter 2

Background

Before introducing the *SAT Modulo Monotonic Theories* (SMMT) framework in Chapters 3 and 4, we review several areas of relevant background. First, we review the theoretical background of Boolean satisfiability, and then survey the most relevant elements of modern conflict-driven clause-learning (CDCL) SAT solvers. Then we review the aspects of many-sorted first-order logic that form the theoretical background of Satisfiability Modulo Theories (SMT) solvers, and describe the basic components of lazy SMT solvers.

2.1 Boolean Satisfiability

Boolean satisfiability (SAT) is the problem of determining, for a given propositional formula, whether or not there exists a truth assignment to the Boolean variables of that formula such that the formula evaluates to TRUE (we will define this more precisely shortly). SAT is a fundamental decision problem in computer science, and the first to be proven NP-Complete [61]. In addition to its important theoretical properties, many real-world problems can be efficiently modeled as SAT problems, making solving Boolean satisfiability in practice an important problem to tackle. Surprisingly, following a long period of stagnation in SAT solvers (from the 1960s to the early 1990s), great strides on several different fronts have now been made in solving SAT problems efficiently, allowing for wide classes of instances to be solved quickly enough to be useful in practice. Major advancements in SAT solver technology include the introduction of WALKSAT [180] and other stochastic local search solvers in the early 1990s, and the development of GRASP [146], CHAFF [151] and subsequent *CDCL* solvers in the late 1990's and following decade. We will review these in Section 2.2.

A Boolean propositional formula is a formula over Boolean variables that, for any given assignment to those variables, evaluates to either TRUE or FALSE. Formally, a propositional formula ϕ can be a *literal*, which is either a Boolean variable v or its negation $\neg v$; or it may be built from other propositional formulas using, e.g. one of the standard binary logical operators $\neg\phi_0, (\phi_0 \wedge \phi_1)$, $(\phi_0 \vee \phi_1)$, $(\phi_0 \implies \phi_1), \ldots$, with ϕ_i a Boolean formula. A truth assignment $\mathcal{M} : v \mapsto \{T, F\}$ for ϕ is a map from the Boolean variables of ϕ to $\{\text{TRUE}, \text{FALSE}\}$. It is also often convenient to treat a truth assignment as a set of literals, containing the literal v iff $v \mapsto T \in \mathcal{M}$, and the literal $\neg v$ iff $v \mapsto F \in \mathcal{M}$. A truth assignment $\mathcal{M}$ is a *complete* truth assignment for a formula ϕ if it maps all variables in the formula to assignments; otherwise it is a *partial* truth assignment. A formula ϕ is *satisfiable* if there exists one or more complete truth assignments to the Boolean variables of ϕ such that the formula evaluates to TRUE; otherwise it is *unsatisfiable*. A propositional formula is *valid* or tautological if it evaluates to TRUE under all possible complete assignments;[1] a propositional formula is *invalid* if it evaluates to FALSE under at least one assignment.

A complete truth assignment $\mathcal{M}$ that satisfies the formula ϕ may be called a *model* of ϕ, written $\mathcal{M} \models \phi$ (especially in the context of first order logic); or it may be called a *witness* for the satisfiability of ϕ (in the context of SAT as an $\mathcal{NP}$-complete problem). A partial truth assignment $\mathcal{M}$ is an *implicant* of ϕ, written $\mathcal{M} \implies \phi$, if all possible completions of the partial assignment $\mathcal{M}$ into a complete assignment must satisfy ϕ; a partial assignment $\mathcal{M}$ is a *prime implicant* of ϕ if $\mathcal{M} \implies \phi$ and no proper subset of $\mathcal{M}$ is itself an implicant of ϕ.[2]

In most treatments, Boolean satisfiability is assumed to be applied only to propositional formulas in Conjunctive Normal Form (CNF), a restricted subset of

[1] In purely propositional logic, there is no distinction between valid formulas and tautologies; however, in most presentations of first order logic there is a distinction between the two concepts.

[2] There is also a notion of *minimal models*, typically defined as complete, satisfying truth assignments of ϕ in which a (locally) minimal number of variables are assigned TRUE.

$$(a \wedge b) \vee c \qquad\qquad (\neg a \vee \neg b \vee x) \wedge (a \vee \neg x) \wedge (b \vee \neg x) \wedge (x \vee c)$$

Figure 2.1: *Left:* a formula not in CNF. *Right:* an equisatisfiable formula in CNF, produced by the Tseitin transformation. The new formula has introduced an auxiliary variable, x, which is constrained to be equal to the subformula $(a \wedge b)$.

propositional formulas. CNF formulas consist of a conjunction of *clauses*, where each clause is a disjunction of one or more Boolean literals. In the literature, a formula in conjunctive normal form is often assumed to be provided in the form of a *set* of clauses $\{c_0, c_1, c_2, \ldots\}$, as opposed to as a *formula*. In this thesis, we use the two forms interchangeably (and refer to both as 'formulas'), but it should always be clear from context which form we mean. Similarly, a clause may itself be provided as a set of literals, rather than as a logical disjunction of literals. For example, we use the set interpretation of a CNF formula in the following paragraph.

If a formula ϕ in conjunctive normal form is unsatisfiable, and ϕ' is also unsatisfiable, and $\phi' \subseteq \phi$, then ϕ' is an *unsatisfiable core* of ϕ. If ϕ' is unsatisfiable, and every proper subset of ϕ' is satisfiable, then ϕ' is a minimal unsatisfiable core.[3]

Any Boolean propositional formula ϕ can be transformed directly into a logically equivalent formula ϕ' in conjunctive normal form through the application of De Morgan's law and similar logical identities, but doing so may require a number of clauses exponential in the size of the original formula. Alternatively, one can apply transformations such as the Tseitin transformation [197] to construct a formula ϕ' in conjunctive normal form (see Figure 2.1). The Tseitin transformation produces a CNF formula ϕ' with a number of clauses linear in the size of ϕ, at the cost of also introducing an extra auxiliary variable for every binary logical operator in ϕ. A formula created through the Tseitin transformation is *equisatisfiable* to the original, meaning that it is satisfiable if and only if the original formula was satisfiable (but a formula created this way might not be logically equivalent to the original formula, as it may have new variables not present in the original formula). In different contexts, Boolean satisfiability solvers may be assumed to be restricted

[3] There is a related notion of a globally minimum unsatisfiable core, which is a minimal unsatisfiable core of ϕ that has the smallest possible number of clauses of all unsatisfiable cores of ϕ.

to operating on CNF formulas, or they may be allowed to take non-normal-form inputs. As the Tseitin transformation (and similar transformations, such as the Plaisted-Greenbaum [167] transformation) can be applied in linear time, in many contexts it is simply left unspecified whether a Boolean satisfiability solver is restricted to CNF or not, with the assumption that a non-normal form formula can be implicitly transformed into CNF if required.[4]

Boolean satisfiability is an $\mathcal{NP}$-complete problem, and all known algorithms for solving it require at least exponential time in the worst case. Despite this, as we will discuss in the next section, SAT solvers often can solve even very large instances with millions of clauses and variables quite easily in practice.

2.2 Boolean Satisfiability Solvers

SAT solvers have a long history of development, but at present, successful solvers can be roughly divided into complete solvers (which can both find a satisfying solution to a given formula, if such a solution exists, or prove that no such solution exists) and incomplete solvers, which are able to find satisfying solutions if they exist, but cannot prove that a formula is unsatisfiable.

Incomplete solvers are primarily stochastic local search (SLS) solvers, which are only guaranteed to terminate on satisfiable instances, originally introduced in the WALKSAT [180] solver. Most complete SAT solvers can trace their development back to the resolution rule([66]), and the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [67]. At the present time, DPLL-based complete solvers have diverged into two main branches: look-ahead solvers (e.g., [97]), which perform well on unsatisfiable 'random' (phase transition) instances (e.g., [76]), or on difficult 'crafted' instances (e.g., [118]); and Conflict-Driven Clause-Learning (CDCL) solvers, which have their roots in the GRASP [146] and CHAFF [151] solvers (along with many other contributing improvements over the years), which perform well on so-called application, or industrial, instances. These divisions have solidified in the last 15 years: in all recent SAT competitions [134], CDCL solvers have won in the applications/industrial category, look-ahead solvers have won in the

[4] There are also some SAT solvers specifically designed to work on non-normal-form inputs (e.g., [163, 204]), with applications to combinatorial equivalence checking.

crafted and unsatisfiable/mixed random categories, while stochastic local search solvers have won in the satisfiable random instances category.

In this thesis, we are primarily interested in CDCL solvers, and their relation to SMT solvers. Other branches of development exist (such as Stålmarck's method [188], and Binary Decision Diagrams [45]), but although BDDs in particular are competitive with SAT solvers in some applications (such as certain problems arising in formal verification [168]), neither of these techniques has been successful in any recent SAT competitions. There have been solvers which combine some of the above mentioned SAT solver strategies, such as *Cube and Conquer* [119], which combines elements of CDCL and lookahead solvers, or SATHYS, which combines techniques from SLS and CDCL solvers [17], with mixed success. As CDCL solvers developed out of the DPLL algorithm, we will briefly review DPLL before moving on to CDCL.

2.2.1 DPLL SAT Solvers

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is shown in Algorithm 1. DPLL combines recursive case-splitting (over the space of truth assignments to the variables in the CNF formula) with a particularly effective search-space pruning strategy and deduction technique, *unit propagation*.[5]

DPLL takes as input a propositional formula ϕ in conjunctive normal form, and repeatedly applies the unit propagation rule (and optionally the pure-literal elimination rule) until a fixed point is reached. After applying unit propagation, if all clauses have been satisfied (and removed from ϕ), DPLL returns TRUE. If any clause has been violated (represented as an empty clause), DPLL returns FALSE. Otherwise, DPLL picks an unassigned Boolean variable v from ϕ, assigning v to either TRUE or FALSE, and recursively applies DPLL to both cases. These free assignments are called *decisions*.

The unit propagation rule takes a formula ϕ in conjunctive normal form, and checks if there exists a clause in ϕ that contains only a single literal l (a *unit clause*). Unit propagation then removes any clauses in ϕ containing l, and removes from each remaining clause any occurrence of $\neg l$. After this process, ϕ does not contain

[5] Unit propagation is also referred to as Boolean Constraint Propagation; in the constraint programming literature, it would be considered an example of the arc consistency rule [199].

Algorithm 1 The Davis-Putnam-Logemann-Loveland (DPLL) algorithm for Boolean satisfiability. Takes as input a set of clauses ϕ representing a formula in conjunctive normal form, and returns TRUE iff ϕ is satisfiable.

```
function DPLL($\phi$)
    while $\phi$ contains unit clause $\{l\}$ do
        Apply unit propagation
        $\phi \leftarrow$ PROPAGATE($\phi, l$)
    if $\phi = \{\}$ then
        All clauses satisfied
        return TRUE
    else if $\{\} \in \phi$ then
        $\phi$ contains empty clause, is unsatisfiable
        return FALSE
    Select a variable $v \in vars(\phi)$
    return DPLL($\phi \cup \{v\}$) or DPLL($\phi \cup \{\neg v\}$)

function PROPAGATE($\phi$, $l$)
    $\phi' \leftarrow \{\}$
    for clause $c \in \phi$ do
        if $l \in c$ then
            Do not add c to $\phi'$
        else if $\neg l \in c$ then
            Remove literal $\neg l$ from c
            $\phi' \leftarrow \phi' \cup (c/\{\neg l\})$
        else
            $\phi' \leftarrow \phi' \cup c$
    return $\phi'$
```

any occurrence of l or $\neg l$.

The strength of the unit propagation rule comes from two sources. First, it can be implemented very efficiently in practice (using high performance data structures such as two-watched-literals [151], which have supplanted earlier techniques such as head/tail lists [216] or unassigned variable counters [146]). Secondly, each time a new unit clause is discovered, the unit-propagation rule can be applied again, potentially making additional deductions. This can be (and is) applied repeatedly until a fixed point is reached and no new deductions can be made. In practice, for many common instances, the unit propagation rule ends up very cheaply making long chains of deductions, allowing it to eliminate large parts of the search space. Because it makes many deductions, and because it does so inexpensively, the unit propagation rule is very hard to augment without unduly slowing down the solver. Other deduction rules have been proposed; in particular, the original DPLL algorithm also applied *pure literal elimination* (in which any variable occurring in only one polarity throughout the CNF is assigned that polarity), however, modern SAT solvers almost always apply the unit propagation rule exclusively.[6]

In practice, the decision of which variable to pick, and whether to test the positive assignment (v) or the negative assignment ($\neg v$), has an enormous impact on the performance of the algorithm, and so careful implementations will put significant effort into good *decision heuristics*. Although many heuristics have been proposed, most current solvers use the VSIDS heuristic (introduced in Chaff), with a minority using the Berkmin [109] heuristic. Both of these heuristics choose which variable to split on next, but do not select which polarity to try first; common polarity selection heuristics include Jeroslow-Wang [135], phase-learning [166], choosing randomly, or simply always choosing one of TRUE or FALSE first.

2.2.2 CDCL SAT Solvers

Conflict-Driven Clause Learning (CDCL) SAT solvers consist of a number of extensions and improvements to DPLL. In addition to the improved decision heuristics and unit-propagation data structures discussed in the previous section, a few of the key improvements include:

1. Clause learning
2. Non-chronological backtracking
3. Learned-clause discarding heuristics
4. Pre-processing
5. Restarts

[6] Note that this generalization applies mostly to CDCL SAT solvers. In contrast, solvers that extend the CDCL framework to other logics, and in particular SMT solvers, sometimes *do* apply additional deduction rules.

Algorithm 2 A simplified Conflict-Driven Clause-Learning (CDCL) solver. Takes as input a set of clauses ϕ representing a formula in conjunctive normal form, and returns TRUE iff ϕ is satisfiable. PROPAGATE applies unit propagation to a partial assignment, adding any unit literals to the assignment, and potentially finding a conflict clause. ANALYZE and BACKTRACK perform clause learning and non-chronological backtracking, as described in the text. SIMPLIFY uses heuristics to identify and discard some of the learned clauses. Lines marked *occasionally* are only applied when heuristic conditions are met.

```
function SOLVE($\phi$)
    level ← 0
    assign ← {}
    loop
        if PROPAGATE(assign) returns a conflict then
            if level = 0 then
                return FALSE
            $c, backtrackLevel \leftarrow$ ANALYZE($conflict$)
            $\phi \leftarrow \phi \cup c$
            level ← BACKTRACK($backtrackLevel, assign$)
        else
            if All variables are assigned then
                return TRUE
            if level > 0 then occasionally restart the solver to level 0
                level ← BACKTRACK($0, assign$)
            if level = 0 then
                Occasionally discard some learned or redundant clauses
                $\phi \leftarrow$ SIMPLIFY($\phi$)
            Select unassigned literal $l$
            level ← level +1
            assign[$l$] ← TRUE
```

Of these, the first two are the most relevant to this thesis, so we will restrict our attention to those. Clause learning and non-chronological backtracking (sometimes ‘back-jumping’) were originally developed independently, but in current solvers they are tightly integrated techniques. In Algorithm 2, we describe a simplified CDCL loop. Unlike our recursive presentation of DPLL, this presentation is in a stateful, iterative form, in which PROPAGATE operates on an assignment structure, rather than removing literals and clauses from ϕ.

Clause learning, introduced in the SAT solver GRASP [146], is a form of 'no-good' learning [74], a common technique in CSP solvers for pruning unsatisfiable parts of the search space. When a conflict occurs in a DPLL solver (that is, when ϕ contains an empty clause), the solver simply backtracks past the most recent decision and then continues exploring the search tree. In contrast, when a clause-learning SAT solver encounters a conflict, it derives a *learned* clause, c, not present in ϕ, that serves to 'explain' (in a precise sense) the conflict. This learned clause is then appended to ϕ. Specifically, the learned clause is such that if the clause had been present in the formula, unit propagation would have prevented (at least) one of the decisions made by the solver that resulted in the conflict. By adding this new clause to ϕ, the solver is prevented in the future from making the same set of decisions that led to the current conflict. In this sense, learned clauses can be seen as augmenting or strengthening the unit propagation rule. Learned clauses are sometimes called *redundant* clauses, because they can be added to or removed from the formula without removing any solutions from the formula.

After a conflict, there may be many choices for the learned clause. Modern CDCL solvers learn *asserting* clauses, which have two additional properties: 1) all literals in the clause are false in the current assignment, and 2) exactly one literal in the clause was assigned at the current decision level. Although many possible strategies for learning clauses have been proposed (see, e.g., [32, 146]), by far the dominant strategy is the *first unique implication point* (1-UIP) method [218], sometimes augmented with post-processing to further reduce the size of the clause ([84, 186]).

Each time a conflict occurs, the solver learns a new clause, eliminating at least one search path (the one that led to the conflict). In practice, clauses are often much smaller than the number of decisions made by the solver, and hence each clause may eliminate many search paths, including ones that have not yet been explored by the solver. Eventually, this process will either force the solver to find a satisfying assignment, or the solver will encounter a conflict in which no new clause can be learned (because no decisions have yet been made by the solver). In this second case, the formula is unsatisfiable, and the solver will terminate. A side benefit of clause learning is that as the clauses serve to implicitly prevent the solver from exploring previously searched paths, the solver is freed to explore paths in any order without having to explicitly keep track of previously explored paths. This allows CDCL solvers to apply restarts cheaply, without losing the progress they have already made.

In a DPLL solver, when a conflict is encountered, the solver backtracks up the search tree until it reaches the most recent decision where only one branch of the search space has been explored, and then continues down the unexplored branch. In a non-chronological backtracking solver, the solver may backtrack more than one level when a conflict occurs. Non-chronological backtracking, like clause learning, has its roots in CSP solvers (e.g. [169]), and was first applied to SAT in the SAT solver REL_SAT [32], and subsequently combined with clause learning in GRASP. When a clause is learned following a conflict, modern CDCL solvers backtrack to the level of the *second* highest literal in the clause. So long as the learned clause was an *asserting* clause (as discussed above), the newly learned clause will then be unit at the current decision level (after having backtracked), and this will trigger unit propagation. The solver can then continue the solving process as normal. Intuitively, the solver backtracks to the lowest level at which, if the learned clause had been in ϕ from the start, unit propagation would have implied an additional literal.

Further components that are also important in CDCL solvers include heuristics and policies for removing learned clauses that are no longer useful [14], restart policies (see [15] for an up-to-date discussion), and pre-processing [83], to name just a few. However, the elements already described in this section are the ones most relevant to this thesis. For further details on the history and implementation of CDCL solvers, we refer readers to [217].

2.3 Many-Sorted First Order Logic & Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) solvers extend SAT solvers to support logics or operations that cannot be efficiently or conveniently expressed in pure propositional logic. Although many different types of SMT solvers have been developed, sometimes with incompatible notions of what a theory is, the majority of SMT solvers have consolidated into a mostly standardized framework built around many-sorted first order logic. We will briefly review the necessary concepts here, before delving into the implementation of Satisfiability Modulo Theories solvers in the next section.

A many-sorted first order formula ϕ differs from a Boolean propositional formula in two ways. First, instead of just being restricted to Boolean variables, each variable in ϕ is associated with a sort (analogous to the 'types' of variables in a programming language). For example, in the formula $(a \leq b) \vee (y + z = 1.5)$, the variables a and b might be of sort integer, while variables y, z might be of sort rational (we'll discuss the mathematical operators in this formula shortly). Formally, a sort σ is a logical symbol, whose interpretation is a (possibly infinite) set of constants, with each term of sort σ in the formula having a value from that set.[7]

The second difference is that a first order formula may contain function and predicate symbols. In many-sorted first order logic, function symbols have typed signatures, mapping a fixed number of arguments, each of a specified sort, to an output of a specified sort. For example, the function $min(integer, integer) \mapsto integer$ takes two arguments of sort integer and returns an integer; the function $quotient(integer, integer) \mapsto rational$ takes two integers and returns a rational. An argument to a function may either be a variable of the appropriate sort, or a function whose output is of the appropriate sort. A *term* t is either a variable v or an instance of a function, $f(t_0, t_1, \ldots)$, where each argument t_i is a term. The number of arguments a function takes is called the 'arity' of that function; functions of arity

[7] In principle, an interpretation for a formula may select any domain of constants for each sort, so long as it is a superset of the values assigned to the terms of that sort, in the same way that non-standard interpretations can be selected, for example, for the addition operator. However, we will only consider the case where the interpretations of each sort are fixed in advance; i.e., the sort $\mathbb{Z}$ maps to the set of integer constants in the expected way.

0 are called constants or constant symbols.

Functions that map to Boolean variables are *predicates*. Although predicates are functions, in first order logic predicates are treated a little differently than other functions, and are associated with special terminology. Intuitively, the reason for this special treatment of Boolean-valued functions is that they form the bridge between propositional (Boolean) logic and first-order logic. A predicate p takes a fixed number of arguments $p(t_0, t_1, \ldots)$, where each t_i is a term, and outputs a Boolean. By convention, it is common to write certain predicates, such as equality and comparison operators, in infix notation. For example, in the formula $a \leq b$, where the variables a and b are of sort integer, the $\leq$ operation is a predicate, as $a \leq b$ evaluates to a truth value.

A predicate with arity 0 is called a propositional symbol or a constant. Each individual instance of a predicate p in a formula is called an *atom* of p. For example, TRUE and FALSE are Boolean constants, and in the formula $p \vee q$, p and q are propositional symbols. TRUE, FALSE, p, and q are all examples of arity-0 predicates. In many presentations of first order logic, an *atom* may also be any Boolean variable that is not associated with a predicate; however, to avoid confusion we will always refer to non-predicate variables as simply 'variables' in this thesis. Atoms *always* have Boolean sorts; an atom is a term (just as a predicate is a function), but not all terms are atoms. If a is an atom, then a and $\neg a$ are literals, but $\neg a$ is not an atom.

A many-sorted first order formula ϕ is either a Boolean variable v, a predicate atom $p(t_0, t_1, \ldots)$, or a propositional logic operator applied to another formula: $\neg\phi_0, (\phi_0 \wedge \phi_1), (\phi_0 \vee \phi_1), (\phi_0 \implies \phi_1), \ldots$ Finally, the variables of a formula ϕ may be existentially or universally quantified, or they may be free. Most SMT solvers only support quantifier-free formulas, in which each variable is implicitly treated as existentially quantified at the outermost level of the formula.

Predicate and function symbols in a formula may either be *interpreted*, or *uninterpreted*. An uninterpreted function or predicate term has no associated semantics. For example, in an uninterpreted formula, the addition symbol '$+$' is not necessarily treated as arithmetic addition. The only rule for uninterpreted functions and predicates is that any two applications of the same function symbol with equal arguments must return the same value.

In first order logic, the propositional logic operators ($\neg, \vee, \wedge.$ etc.) are always interpreted. However, by default, other function and predicate symbols are typically assumed to be uninterpreted. Many treatments of first order logic also assume that the equality predicate, '$=$', is always interpreted; this is sometimes called *first order logic with equality*.

A structure $\mathcal{M}$ for a formula ϕ is an assignment of a value of the appropriate sort to each variable, predicate atom, and function term. Unlike an assignment in propositional logic, a structure must also supply an assignment of a concrete function (of the appropriate arity and sorts) to each function and predicate symbol in ϕ. For example, in the formula $z = \mathit{func}(x,y)$, both $\{x \mapsto 1, y \mapsto 2, z \mapsto 1, \text{`}\mathit{func}\text{'} \mapsto \mathit{min}, \text{`}{=}\text{'} \mapsto \mathit{equality}\}$ and $\{x \mapsto 1, y \mapsto 2, z \mapsto 2, \text{`}\mathit{func}\text{'} \mapsto \mathit{max}, \text{`}{=}\text{'} \mapsto \mathit{equality}\}$ are both structures for ϕ.

Given a structure $\mathcal{M}$, a formula ϕ evaluates to either TRUE or FALSE. A structure is a complete structure if it provides an assignment to every variable, atom, and term (and every uninterpreted function symbol); otherwise it is a partial structure. A (partial) structure $\mathcal{M}$ *satisfies* ϕ (written $\mathcal{M} \models \phi$) if ϕ must evaluate to TRUE in every possible completion of the assignment in $\mathcal{M}$. A formula ϕ is *satisfiable* if there exists a structure $\mathcal{M}$ that satisfies ϕ, and unsatisfiable if no such structure exists. A structure that satisfies formula ϕ may be called a *model*, an interpretation, or a solution for ϕ.

In the context of Satisfiability Modulo Theories solvers, we are primarily interested in interpreted first order logic.[8] Those interpretations are supplied by *theories*. A theory T is a (possibly infinite) set of logical formulas, the conjunction of which must hold in any satisfying assignment of that theory. A structure $\mathcal{M}$ is said to satisfy a formula ϕ modulo theory T, written $\mathcal{M} \underset{T}{\Longrightarrow} \phi$, iff $\mathcal{M}$ satisfies all the formulas in $T \cup \{\phi\}$.

The formulas in a theory serve to constrain the possible satisfying assignments to (some of) the function and predicate symbols. For example, a theory of integer addition may contain the infinite set of formulas $(0+1=1, 1+0=1, 1+1=2, \ldots)$ defining the binary addition function. A structure $\mathcal{M}$ satisfies a formula ϕ modulo theory T if and only if $\mathcal{M}$ satisfies $(T \cup \phi)$. In other words, the structure must

[8] With the specific exception of SMT solvers for the theory of uninterpreted functions, SMT solvers always operate on interpreted formulas.

simultaneously satisfy both the original formula ϕ, *and* all the (possibly infinite) formulas that make up T.

Returning to the first order formula $(a \leq b) \vee (y+z = 1.5)$ that we saw earlier, $\leq$ and $=$ are predicates, while $+$ is a function. If a, b are integers, and y, z are rationals, then a theory of linear integer arithmetic may constrain the satisfiable interpretations of the $\leq$ function so that in all satisfying models, the predicate behaves as the mathematical operator would be expected to. Similarly, a theory of linear rational arithmetic might specify the meaning of the '$+$' predicate, while the interpretation of the equality predicate may be assumed to be supplied by an implicit theory of equality (or it might be explicitly provided by the theory of linear rational arithmetic).

A theory is a (possibly infinite) set of logical formulas. The *signature* of a theory, Σ, consists of three things:

1. The sorts occurring in the formulas in that theory,
2. the predicate symbols occurring in the formulas of that theory, and
3. the function symbols occurring in the formulas of that theory.

In most treatments, the Boolean sort, and the standard propositional logic operators $(\neg, \vee, \wedge, etc.)$ are implicitly available in every theory, without being counted as members of their signature. Similarly, the equality predicate is typically also implicitly available in all theories, while being excluded from their signatures. Constants (0-arity functions) are included in the signature of a theory.

For example, the signature of the theory of linear integer arithmetic (LIA) consists of:

1. Sorts = $\{\mathbb{Z}\}$
2. Predicates = $\{<, \leq, \geq, >\}$
3. Functions =$\{+, -, 0, 1, -1, 2, -2 \ldots\}$

A formula ϕ in which all sorts, predicates, and function symbols appearing in ϕ can be found in the signature of theory T (aside from Booleans, propositional logic operators, and the equality predicate) is said to be written in the language of T. An example of a formula in the language of the theory of linear integer arithmetic is:

$$(x > y) \vee \big((x + y = 2) \wedge (x < -1)\big)$$

In this section we have presented one internally self-consistent description of many-sorted first order logic, covering the most relevant elements to Satisfiability Modulo Theories solvers. However, in the literature, there are many variations of these concepts, both in terminology and in actual semantics. For an introduction to many-sorted first order logic as it applies to SMT solvers, we refer readers to [71].

2.4 Satisfiability Modulo Theories Solvers

Satisfiability Modulo Theories solvers extend Boolean satisfiability solvers so that they can solve many-sorted first order logic formulas written in the language of one or more theories, with different SMT solvers supporting different theories.

Historically, many of the first SMT solvers were ‘eager’ solvers, which convert a formula into a purely propositional CNF formula, and then subsequently apply an unmodified SAT solver to that formula; an example of an early eager SMT solvers was UCLID [46]. For some theories, eager SMT solvers are still state of the art (in particular, the bitvector solver STP [100] is an example of a state-of-the-art eager solver; another example would be MINISAT+, a pseudo-Boolean constraint solver, although pseudo-Boolean constraint solvers are not traditionally grouped together with SMT solvers). However, for many theories (especially those involving arithmetic), eager encodings require exponential space.

The majority of current high-performance SMT solvers are *lazy* SMT solvers, which attempt to avoid or delay encoding theory constraints into a propositional formula. While the predecessors of lazy SMT solvers go back to the mid 1990s [27, 107], they were gradually formalized into a coherent SMT framework in the early to mid 2000s [9, 16, 31, 73, 101, 161, 192].

The key idea behind lazy SMT solvers is to have the solver operate on an abstracted, Boolean *skeleton* of the original first order formula, in which each theory atom has been swapped out for a fresh Boolean literal (see Figure 2.2). The first lazy SMT solvers were *offline* [31, 72, 73]. Offline SMT solvers combine an un-

$((x > 1) \wedge (2x < 5)) \vee \neg(y = 0)$ $\qquad$ $(a \wedge b) \vee \neg c$

Figure 2.2: *Left:* A first order formula in the theory of linear real arithmetic, with three theory atoms. *Right:* Boolean skeleton of the same formula. Boolean variables a, b, c replace atoms $(x > 1), (2x < 5), (y = 0)$.

modified SAT solver with a specialized *theory solver* for each of the theories in the formula. Offline lazy SMT solvers still treat the SAT solver as a black box, but unlike an eager solver, they encode the propositional formula incrementally, repeatedly solving and refining the abstracted Boolean version of the first order formula. Initially, a SAT solver solves the Boolean skeleton of the formula, which produces a truth assignment to the fresh literals introduced for each atom.

The solver passes the corresponding assignment to the original theory atoms to a specialized *theory solver*, which checks if there exists a satisfying model for the theory under that assignment to those theory atoms. If there is a satisfying assignment in the theory, then the solver terminates, returning SAT. If the formula is unsatisfiable in the theory under that assignment to the atoms, then the theory solver derives a learned clause to add to the Boolean skeleton which blocks that assignment, and the process repeats. We will describe theory solvers in more detail shortly.

Although there are still some state-of-the-art offline SMT solvers (the bitvector solver Boolector [42] is an example), in most cases, offline solvers perform very poorly [101], producing many spurious solutions to the propositional formula that are trivially false in the theory solver. The vast majority [179] of modern SMT solvers are instead *online* lazy SMT solvers, formalized in [101]. Unlike an eager or offline SMT solver, an *online* SMT solver is tightly integrated into a modified SAT solver. For example, the Abstract DPLL framework explicitly formalizes CDCL solvers as state machines [161] for the purpose of reasoning about the correctness of SMT solvers. While there has been work extending stochastic local search SAT solvers into SMT solvers (e.g., [98, 112, 158]), by far the dominant approach at this time is to combine CDCL solvers with specialized reasoning procedures. For this reason, we will phrase our discussion in this section explicitly in terms of CDCL solvers.

An online lazy SMT solver consists of two components. The first, as with offline lazy solvers, is a specialized theory solver (or '*T-solver*'). The second component is a modified CDCL solver, which adds a number of hooks to interact with the theory solver. As before, the SAT solver operates on an abstracted Boolean skeleton of the first order formula, ϕ'. Unlike in offline solvers, an online solver does not wait to generate a complete satisfying assignment to the Boolean formula before checking whether the corresponding assignment to the theory atoms is satisfiable in the theory solver, but instead calls the theory solver eagerly, as assignments to theory literals are made in the SAT solver.

Theory solvers may support some or all of the following methods:

1. *T-Propagate* ($\mathcal{M}$):

   The most basic version of *T-Propagate* (also called *T-Deduce*) takes a complete truth assignment $\mathcal{M}$ to the theory atoms (for a single theory), returning TRUE if $\mathcal{M}$ is satisfiable in the theory, and FALSE otherwise.

   In contrast to offline solvers, online lazy SMT solvers (typically) make calls to *T-Propagate* eagerly [101], as assignments are being built in the CDCL solver, rather than waiting for a complete satisfying assignment to be generated. In this case, *T-Propagate* takes a partial, rather than a complete, assignment to the theory atoms. When operating on a partial assignment, *T-Propagate* makes deductions on a best-effort basis: if it can prove that the partial assignment is unsatisfiable, then it returns FALSE, and otherwise it returns TRUE. By calling *T-Propagate* eagerly, assignments that may satisfy the abstract Boolean skeleton, but which are unsatisfiable in the theory solver, can be pruned from the CDCL solvers search space. This technique is sometimes called *early pruning*.

   All theory solvers must support at least this basic functionality, but some theory solvers can do more. If $\mathcal{M}$ is a partial assignment, efficient theory solvers may also be able to deduce assignments for some of the unassigned atoms. These are truth assignments to theory literals l that *must* hold in all satisfying completions of $\mathcal{M}$, written $\mathcal{M} \underset{T}{\Longrightarrow} l$. Along with early pruning, deducing unassigned theory literals may greatly prune the search space of the solver, avoiding many trivially unsatisfiable solutions.

As with unit propagation, the CDCL solver typically continues applying unit propagation and theory propagation until no further deductions can be made (or the assignment is found to be UNSAT). However, there are many possible variations on the exact implementation of theory propagation. For many theories of interest, testing satisfiability may be very expensive; deducing assignments for unassigned atoms may be an additional cost on top of that, in some cases a prohibitively expensive one. Many theory solvers can only cheaply make deductions of a subset of the possible forced assignments. Theory solvers that make all possible deductions of unassigned literals are called *deduction complete*.

There are many possible optimizations that have been explored in the literature to try to mitigate the cost of expensive theory propagation calls. A small selection include: only calling theory propagate when one or more theory literals have been assigned [108]; *pure literal filtering* [16, 39], which can remove some redundant theory literals from $\mathcal{M}$; or only calling theory propagate every k unit propagation calls, rather than for every single call [10]. A theory solver that does not directly detect implied literals can still be used to deduce such literals by testing, for each unassigned literal l, whether $\mathcal{M} \cup \{\neg l\}$ is unsatisfiable. This technique is known as *theory plunging*[77], however, it is rarely used in practice due to its cost.

2. *T-Analyze* ($\mathcal{M}$):
   When an assignment to the theory atoms $\mathcal{M}$ is unsatisfiable in the theory, then an important task for a theory solver is to find a (possibly minimal) subset of $\mathcal{M}$ that is sufficient to be unsatisfiable. This unsatisfiable subset may be called a conflict set or justification set or infeasible set; its negation (a clause of theory literals, at least one of which must be true in any satisfying model) may be called a theory lemma or a learned clause.

   T-Analyze may always return the entire $\mathcal{M}$ as a conflict set; this is known as the naïve conflict set. Unfortunately, the naïve conflict set is typically a very poor choice, as it blocks only the exact assignment $\mathcal{M}$ to the theory atoms. Conversely, it is also possible to find a (locally) minimal conflict set through a greedy search, repeatedly dropping literals from $\mathcal{M}$ and checking if it is still unsatisfiable. However, in addition to not guaranteeing a globally minimal conflict set, such a method is typically prohibitively expensive, especially if the theory propagate method is expensive. Finding small conflict sets quickly is a key challenge for theory solvers.

3. *T-Backtrack*(level):
   In many cases, efficient theory solvers maintain incremental data structures, so that as *T-Propagate* ($\mathcal{M}$) is repeatedly invoked, redundant computation in the theory solver can be avoided. *T-Backtrack* is called when the CDCL solver backtracks, to keep those incremental data structures in sync with the CDCL solver's assignments.

4. *T-Decide*($\mathcal{M}$):
   Some theory solvers are able to heuristically suggest unassigned theory literals as decisions to the SAT solver. If there are multiple theory solvers capable of suggesting decisions, the SAT solver may need to have a way of choosing between them.

5. *T-Model*($\mathcal{M}$):
   Given a theory-satisfiable assignment $\mathcal{M}$ to the theory atoms, this method extends $\mathcal{M}$ into a satisfying model to each of the (non-Boolean) variables and function terms in the formula. Generating a concrete model may require a significant amount of additional work on top of simply checking the satisfiability of a Boolean assignment $\mathcal{M}$, and so may be separated off into a separate method to be called when the SMT solver finds a satisfying solution. In some solvers (such as Z3 [70]), models can be generated on demand for specified subsets of the theory variables.

Writing an efficient lazy SMT solver involves finding a balance between more expensive deduction capabilities in the theory solver and faster searches in the CDCL solver; finding the right balance is difficult and may depend not only on the implementation of the theory solver but also on the instances being solved. For a more complete survey of some of the possible optimizations to CDCL solvers for integration with lazy theory solvers, we refer readers to Chapter 6 of [179].

2.5 Related Work

In the following chapters, we will introduce the main subject of this thesis: a general framework for building efficient theory solvers for a special class of theories, which we call *monotonic theories*. Several other works in the literature have also introduced high-level, generic frameworks for building SMT solvers. There are also previous constraint solvers which exploit monotonicity. Here, we briefly survey some of the most closely related works. Additionally, the applications and specific theories we consider later in this thesis (Chapters 5, 6, 7, and 8) also have their own, subject-specific literature reviews, and we describe related work in the relevant chapters.

There have been a number of works proposing generic SMT frameworks that are not specialized for any particular class of first order theories. The best known of these are the DPLL(T) and Abstract DPLL frameworks [101, 161]. Additionally, an approach based on extending simplex into a generic SMT solver method is described in [111]. These frameworks have in common that they are not specific to the class of theories being supported by the SMT solver; the techniques described in them apply generically to all (quantifier free, first order) theories. Therefore, the work in this thesis could be considered to be an instance of the DPLL(T) framework. However, these high-level frameworks do not articulate any concept of monotonic theories, and provide no guidance for exploiting their properties.

As we will see in Chapters 5 and 6, some of our most successful examples of monotonic theories are graph theoretic. Two recent works have proposed generic frameworks for extending SMT solvers with support for graph properties:

The first of the two, [220], proposed MathCheck, which extends an SMT solver by combining it with an external symbolic math library (such as SAGE [189]). This allows the SMT solver great flexibility in giving it access to a wide set of operations that can be specified by users in the language of the symbolic math library. They found that they were able to prove bounded cases of several discrete mathematical theorems using this approach. Unfortunately, their approach is fundamentally an off-line, lazy SMT integration, and as such represents a extremal point in the expressiveness-efficiency trade-off: their work can easily support many previously unsupported theories (including theories that are non-monotonic), however, those solvers are typically much less efficient than SMT solvers with dedicated support for the same theories.

The second of the two, [133], introduced SAT-to-SAT. Like [220], they describe an off-line, lazy SMT solver. However, rather than utilizing an external math library to implement the theory solver, they utilize a second SAT solver to implement the theory solver. Further, they negate the return value of that second SAT solver, essentially treating it as solving a negated, universally quantified formula. Their approach is similarly structured to the 2QBF solvers proposed in [172] and [185]. Like MathCheck, SAT-to-SAT is not restricted to monotonic theories. However, SAT-to-SAT requires encoding the supported theory in propositional logic, and suffers a substantial performance penalty compared to our approach (for the theories that both of our solvers support).

One of the main contributions of this thesis is our framework for interpreted monotonic functions in SMT solving. Although there has been work addressing *uninterpreted* monotonic functions in SMT solvers [23], we believe that no previous work has specifically addressed exploiting interpreted monotonic functions in SMT solvers.

Outside of SAT and SMT solving, other techniques in formal methods have long considered monotonicity. In the space of non-SMT based constraint solvers, two examples include [213], who proposed a generic framework for building CSP solvers for monotonic constraints; and interval arithmetic solvers, which are often designed to take advantage of monotonicity directly (see, e.g., [122] for a survey). Neither of these techniques directly extend to SMT solvers, nor have they been applied to the monotonic theories that we consider in this thesis. Additionally, several works have exploited monotonicity within SAT/SMT-based model-checkers [43, 96, 115, 142].

Chapter 3

SAT Modulo Monotonic Theories

Our interest in this work is to develop tools for building solvers for finite-domain theories in which all predicates and functions are monotonic — i.e., they consistently increase (or consistently decrease) as their arguments increase. These are the *finite monotonic theories*.[1] In this chapter, we formally define monotonic predicates (Section 3.1) and monotonic theories (Section 3.2).

While this notion of a finite monotonic theory may initially seem limited, we will show that such theories are natural fits for describing many useful properties of discrete, finite structures, such as graphs (Chapter 5) and automata (Chapters 8). Moreover, we have found that many useful finite monotonic theories can be efficiently solved in practice using a common set of simple techniques for building lazy SMT theory solvers (Section 2.4). We will describe these techniques — the SAT Modulo Monotonic Theories (SMMT) framework — in Chapter 4.

3.1 Monotonic Predicates

Conceptually, a (positive) monotonic predicate P is one for which if $P(x)$ holds, and $y \geq x$, then $P(y)$ holds. An example of a monotonic predicate is $\mathit{IsPositive}(x) : \mathbb{R} \mapsto \{T,F\}$, which takes a single real-valued argument x, and returns TRUE iff $x > 0$. An example of a non-monotonic predicate is $\mathit{IsPrime}(x)$. Formally:

[1] To forestall confusion, note that our concept of a 'monotonic theory' here has no direct relationship to the concept of monotonic/non-monotonic logics.

Definition 1 (Monotonic Predicate)**.** *A predicate P:* $\{\sigma_1, \sigma_2, \dots \sigma_n\} \mapsto \{T, F\}$*, over sorts* σ_i *is monotonic iff, for each i in* $1..n$*, the following holds:*

$$\begin{cases} \textit{Positive monotonic in argument i:} \\ \forall s_1 \dots s_n, \forall x \leq y : P(\dots, s_{i-1}, x, s_{i+1}, \dots) \rightarrow P(\dots, s_{i-1}, y, s_{i+1}, \dots) \\ \textit{—or—} \\ \textit{Negative monotonic in argument i:} \\ \forall s_1 \dots s_n, \forall x \leq y : \neg P(\dots, s_{i-1}, x, s_{i+1}, \dots) \rightarrow \neg P(\dots, s_{i-1}, y, s_{i+1}, \dots) \end{cases}$$

We will say that a predicate P is positive monotonic in argument i if $\forall s_1 \dots s_n$, $\forall x \leq y : P(\dots, s_{i-1}, x, s_{i+1}, \dots) \rightarrow P(\dots, s_{i-1}, y, s_{i+1}, \dots)$; we will say that P is negative monotonic in argument i if $\forall s_1 \dots s_n, \forall x \leq y : \neg P(\dots, s_{i-1}, x, s_{i+1}, \dots) \rightarrow \neg P(\dots, s_{i-1}, y, s_{i+1}, \dots)$. Notice that although P must be monotonic in all of its arguments, it may be positive monotonic in some, and negative monotonic in others.

Notice that Definition 1 does not specify whether $\leq$ forms a total or a partial order. In this work, we will typically assume total orders, though we consider extensions to support partial orders in Section 4.4.

This work is primarily concerned with monotonic predicates over finite sorts (such as Booleans and bit vectors); we refer to such predicates as *finite monotonic predicates*. Two closely related special cases of finite monotonic predicates deserve attention: monotonic predicates over Booleans, and monotonic predicates over powerset lattices. Formally, we define Boolean monotonic predicates as:

Definition 2 (Boolean Monotonic Predicate)**.** *A predicate P: $\{T,F\}^n \mapsto \{T,F\}$ is Boolean monotonic iff, for each i in $1..n$, the following holds:*

$$\begin{cases} \textit{Positive monotonic in argument i:} \\ \forall s_1 \dots s_n : P(\dots, s_{i-1}, F, s_{i+1}, \dots) \rightarrow P(\dots, s_{i-1}, T, s_{i+1}, \dots) \\ \textit{---or---} \\ \textit{Negative monotonic in argument i:} \\ \forall s_1 \dots s_n : \neg P(\dots, s_{i-1}, F, s_{i+1}, \dots) \rightarrow \neg P(\dots, s_{i-1}, T, s_{i+1}, \dots) \end{cases}$$

As an example of a Boolean monotonic predicate, consider the pseudo-Boolean inequality $\sum_{i=0}^{n-1} c_i b_i \geq c_n$, with each b_i a variable in $\{T,F\}$, and each c_i a non-negative integer constant. This inequality can be modeled as a positive Boolean monotonic predicate P over the Boolean arguments b_i, such that P is TRUE iff the inequality is satisfied.

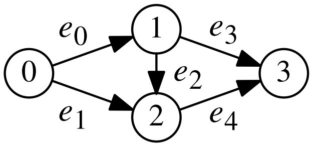
This definition of monotonicity over Booleans is closely related to a definition of monotonic predicates of powerset lattices found in [41] and [145]. Given a set S, [41] defines a predicate $P : 2^S \mapsto \{T,F\}$ to be monotonic if $P(S_x) \rightarrow P(S_y)$ for all $S_x \subseteq S_y$.[2] Slightly generalizing on that definition, we define a monotonic predicate over set arguments as:

Definition 3 (Set-wise Monotonic Predicate)**.** *Given sets $S_1, S_2 \dots S_n$ a predicate P: $\{2^{S_1}, 2^{S_2} \dots 2^{S_n}\} \mapsto \{T,F\}$, is monotonic iff, for each i in $1..n$, the following holds:*

$$\begin{cases} \textit{Positive monotonic in argument i:} \\ \forall s_1 \dots s_n, \forall s_x \subseteq s_y : P(\dots, s_{i-1}, s_x, s_{i+1}, \dots) \rightarrow P(\dots, s_{i-1}, s_y, s_{i+1}, \dots) \\ \textit{---or---} \\ \textit{Negative monotonic in argument i:} \\ \forall s_1 \dots s_n, \forall s_x \subseteq s_y : \neg P(\dots, s_{i-1}, s_x, s_{i+1}, \dots) \rightarrow \neg P(\dots, s_{i-1}, s_y, s_{i+1}, \dots) \end{cases}$$

[2] Actually, [41] poses a slightly stronger requirement, requiring that $P(S)$ must hold. We relax this requirement, in addition to generalizing their definition to support multiple arguments of both positive and negative monotonicity.

As an illustrative example of a finite monotonic predicate of sets, consider *graph reachability*: $reach_{s,t}(E)$, over a set of edges $E \subseteq V \times V$, for a finite set of vertices V. Predicate $reach_{s,t}(E)$ is TRUE iff there is a path from s to t through the edges of E (see example in Figure 3.1).



$$reach_{0,3}(E) \wedge \neg reach_{1,3}(E) \wedge \neg(e_0 \in E \wedge e_1 \in E)$$

Figure 3.1: A finite symbolic graph over set E of five potential edges, and a formula constraining those edges. A satisfying assignment to E is $E = \{e_1, e_4\}$.

Notice that $reach_{s,t}(E)$ describes a family of predicates over the edges of the graph: for each pair of vertices s, t, there is a separate *reach* predicate in the theory. Each $reach_{s,t}(E)$ predicate is monotonic with respect to the set of edges E: if a node v is reachable from another node u in a given graph that does not contain $edge_i$, then it must still be reachable in an otherwise identical graph that also contains $edge_i$. Conversely, if a node v is not reachable from node u in the graph, then removing an edge cannot make v reachable.

Notice that these set-wise predicates are monotonic with respect to subset inclusion, a partial-order. However, observe that any finite set-wise predicate $p(S_0, S_1, \ldots)$ can be converted into Boolean predicates, by representing each set argument S_i as a vector of Booleans, where Boolean argument $b_{S_{(i,j)}}$ is TRUE iff element $j \in S_i$. For example, consider again the set-wise monotonic predicate $reach_{s,t}(E)$. As long as E is finite, we can convert $reach_{s,t}(E)$ into a logically equivalent Boolean monotonic predicate over the membership of E,

$reach_{s,t,E}(edge_1, edge_2, edge_3, \ldots)$, where the Boolean arguments $edge_i$ define which edges (via some mapping to the fixed set of possible edges E) are included in the graph.

This transformation of set-wise predicates into Boolean predicates will prove to be advantageous later, as the Boolean formulation is totally-ordered (with respect to each individual Boolean argument), whereas the set-wise formulation is only partially ordered. As we will discuss in Section 4.4, the SMMT framework — while applicable also to partial orders — works better for total orders. Below, we assume that monotonic predicates of finite sets are always translated into logically equivalent Boolean monotonic predicates, unless otherwise stated.

3.2 Monotonic Theories

Monotonic predicates and functions — finite or otherwise — are common in the SMT literature, but in most cases are considered alongside collections of non-monotonic predicates and functions. For example, the inequality $x + y > z$, with x, y, z real-valued, is an example of an infinite domain monotonic predicate in the theory of linear arithmetic (positive monotonic in x, y, and negative monotonic in z). However, the theory of linear arithmetic — as with most common theories — can also express non-monotonic predicates (e.g., $x = y$, which is not monotonic in either argument).

We introduce the restricted class of *finite monotonic theories*, which are theories over finite domain sorts, in which all predicates and functions are monotonic. Formally, we define a finite monotonic theory as:

Definition 4 (Finite Monotonic Theory)**.** *A theory T with signature Σ is finite monotonic if and only if:*

1. *All sorts in Σ have finite domains;*
2. *all predicates in Σ are monotonic; and*
3. *all functions in Σ are monotonic.*

As is common in the context of SMT solving, we consider only decidable, quantifier-free, first-order theories. All predicates in the theory must be monotonic; atypically for SMT theories, monotonic theories do *not* include equality (as equality is non-monotonic). Rather, as each sort $\sigma \in T$ is ordered, we assume the presence of comparison predicates in T: $<, \leq, \geq, >$, over $\sigma \times \sigma$. Unlike the equality predicate, the comparison predicate *is* monotonic, and we will take advantage of this property subsequently.

Above, we described two finite monotonic predicates: a predicate of pseudo-Boolean comparisons, and a predicate of finite graph reachability. A finite monotonic theory might collect together several monotonic predicates that operate over one or more sorts. For example, many common graph properties are monotonic with respect to the edges in the graph, and a finite monotonic theory of graphs might include in its signature several predicates, including the above mentioned $reach_{s,t}(E)$, as well as additional monotonic predicates such as $acyclic(E)$, which evaluates to TRUE iff edges E do not contain a cycle, or $planar(E)$ which evaluates to TRUE iff edges E induce a planar graph. [3]

Although almost all theories considered in the SMT literature are non-monotonic, particularly as most theories implicitly include the equality predicate, we will show that many useful properties — including many properties that have not previously had practical support in SAT solvers — can be modeled as monotonic finite theories, and solved efficiently. Moreover, we will show in Chapter 4, that building high-performance SMT solvers for such theories is simple and straightforward. Subsequently, in Chapters 5, 7, and 8 of this thesis, we will demonstrate state-of-the-art implementations of lazy SMT solvers for several important finite monotonic theories that previously had poor support in SAT solvers.

[3] In previous work [33], we considered the special case of a Boolean monotonic theory, in which the only sort in the theory is Boolean (and hence all monotonic predicates are Boolean monotonic predicates). In this thesis, the notion of a Boolean monotonic theory is subsumed by the more general notion of a finite monotonic theory.

Chapter 4

A Framework for SAT Modulo Monotonic Theories

This chapter introduces a set of techniques — the SAT Modulo Monotonic Theories (SMMT) framework — taking advantage of the special properties of finite monotonic theories in order to create efficient SMT solvers, by providing efficient theory propagation and improving upon naïve conflict analysis.[1]

Many successful SMT solvers follow the *lazy* SMT design [70, 179], in which a SAT solver is combined with a set of theory solvers (or *T-solvers*), which are responsible for reasoning about assignments to theory atoms (see Section 2.4 for more details about lazy SMT solvers).

While *T-solvers* may support more complex interfaces, every *T-solver* must *at minimum* support two procedures:

1. *T-Propagate* $(\mathcal{M})$, which takes a (possibly partial) assignment $\mathcal{M}$ to the theory atoms, and returns FALSE if it can prove that $\mathcal{M}$ is unsatisfiable in the theory, and TRUE if it could not prove $\mathcal{M}$ to be unsatisfiable. [2]

2. *T-Analyze* $(\mathcal{M})$, which, given an unsatisfiable assignment to the theory literals $\mathcal{M}$, returns a subset (a conflict set) that is also unsatisfiable.[3]

[1] This chapter describes the SMMT framework at a high-level; we refer readers to Appendix A for a discussion of some of the practical considerations required for an efficient implementation.

[2] *T-Propagate* is sometimes referred to as *T-Deduce* in the literature.

[3] *T-Analyze* is sometimes referred to as conflict or justification set derivation, or as lemma learning

The SMMT framework provides a pattern for implementing these two procedures.

In principle, *T-Propagate* is only required to return FALSE when a complete assignment to the theory atoms is unsatisfiable (returning TRUE on partial assignments), and *T-Analyze* may always return the naïve conflict set (i.e., it may return the entire assignment $\mathcal{M}$ as the conflict set). However, efficient *T-Propagate* implementations typically support checking the satisfiability of partial assignments (allowing the SAT solver to prune unsatisfiable branches early). Efficient *T-Propagate* implementations are often also able to derive unassigned literals l that are implied by a partial assignment $\mathcal{M}$, such that $T \cup \mathcal{M} \models l$. Any derived literals l can be added to $\mathcal{M}$, and returned to the SAT solver to extend the current assignment. Efficient implementations of *T-Analyze* typically make an effort to return small (sometimes minimal) unsatisfiable subsets as conflict sets. Although there are many other implementation details that can impact the performance of a theory solver (such as the ability for the theory solver to be applied efficiently as an assignment is incrementally built up or as the solver backtracks), the practical effectiveness of a lazy SMT solver depends on fast implementations of *T-Propagate* and *T-Analyze*. This implies a design trade-off, where more powerful deductive capabilities in the theory solver must be balanced against faster execution time.

In Sections 4.1 and 4.2, we show how efficient theory propagation and conflict analysis can be implemented for finite monotonic theories. The key insight is to observe that many predicates have known, efficient algorithms for evaluating their truth value when given *fully* specified inputs, but not for partially specified or symbolic inputs.[4] For example, given a concretely specified graph, one can find the set of nodes reachable from u simply using any standard graph reachability algorithm, such as depth-first-search.

Given only a procedure for computing the truth-values of the monotonic predicates P from complete assignments, we will show how we can take advantage of the properties of a monotonic theory to form a complete decision procedure for any

or clause learning (in which case the method returns a clause representing the negation of a conflict set).

[4] Evaluating a function given a partially specified or symbolic input is typically more challenging than evaluating a function on a fully specified, concrete input, as the return value may be a set of possible values, or a formula, rather than a concrete value. Additionally, evaluating a function on a partial or symbolic assignment may entail an expensive search over the remaining assignment space.

finite monotonic theory, and, further, we will show that in many cases the resulting theory solver performs much better than preexisting solvers in practice.

For clarity, in Sections 4.1 and 4.2 we initially describe how to apply theory propagation and conflict analysis for the special case of *function-free*, totally-ordered finite monotonic theories. These are finite monotonic theories that have no functions in their signature, aside from predicates (of any arity), and aside from constant (arity-0) functions, and for which every sort has a total order relationship (rather than a partial order relationship). Subsequently, we relax these restrictions to support formulas with compositions of positive and negative monotonic functions (Section 4.3), and to support finite sorts that are only partially-ordered (Section 4.4).

4.1 Theory Propagation for Finite Monotonic Theories

First, we show how efficient theory propagation can be performed for function-free, totally ordered, finite monotonic theories. Consider a theory with a monotonic predicate $P(x_0, x_1, \ldots)$, with x_i of finite sort σ. Let σ be totally ordered, over the finite domain of constants $\sigma_\bot, \sigma_1, \ldots, \sigma_\top$, with $\sigma_\bot \leq \sigma_1 \leq \ldots \leq \sigma_\top$.

In Algorithm 3, we describe the simplest version of our procedure for performing theory propagation on partial assignments for function-free, totally-ordered finite monotonic theories. For simplicity, we also assume for the moment that all predicates are positive monotonic in all arguments.[5]

Algorithm 3 is straightforward. Intuitively, it simply constructs conservative upper and lower bounds (x^+, x^-) for each variable x, and then evaluates each predicate on those upper and lower bounds. Since the predicates are positive monotonic, if $p(x^-)$ evaluates to TRUE, then $p(x)$, for any possible assignment of $x \geq x^-$, must also evaluate to TRUE. Therefore, if $p(x^-)$ evaluates to TRUE, we can conclude that $p(x)$ must be TRUE in any satisfying completion of the partial assignment $\mathcal{M}$. Similarly, if $p(x^+)$ evaluates to FALSE, then $p(x)$ must be FALSE in any satisfying completion of $\mathcal{M}$. The extension to multiple arguments is obvious, and presented in Algorithm 3; we discuss further extensions in Sections 4.3 and 4.4.

[5] In the special case of a function-free Boolean monotonic theory, in which all predicates are restricted to purely Boolean arguments, Algorithm 3 can be simplified slightly by reducing comparisons to truth assignments (e.g., $(x < 1) \equiv \neg x$, if $x \in \{T, F\}$.).

Because x^- and x^+ are constants, rather than symbolic expressions, evaluating $p(x^-)$ or $p(x^+)$ does not require any reasoning about p over symbolic arguments. As such, so long as one has any algorithm for computing predicate p on concrete inputs (not symbolic), one can plug that algorithm into Algorithm 3 to get a complete theory propagation procedure. In the typical case that computing p is well-studied, one has the freedom to use any existing algorithm for computing p in order to implement theory propagation and theory deduction — so long as p is monotonic. For example, if p is the graph reachability predicate we considered in Chapter 3, one can directly apply depth-first-search — or any other standard graph reachability algorithm from the literature — to evaluate p on the under and over approximations of its graph.

In other words, for finite monotonic theories, the same algorithms that can be used to check the truth value of a predicate in a witnessing model (i.e., a complete assignment of every atom and also of each variable to a constant value) can also be used to perform theory propagation and deduction (and, we will demonstrate experimentally, does so very efficiently in practice). This is in contrast to the usual case in lazy SMT solvers, in which the algorithms capable of evaluating the satisfiability of partial assignments may bear little resemblance to algorithms capable of evaluating a witness — typically, the latter is trivial, while the former may be very complex.

The distinction here is well-illustrated by any of the widely supported arithmetic theories, such as difference logic, linear arithmetic, or integer arithmetic. Checking a witness for any of these three arithmetic theories is trivial, requiring linear time in the formula size. In contrast, finding efficient techniques for checking the satisfiability of a (partial or complete) assignment to the atoms in any of these three theories bears little resemblance to witness checking — for example, typical linear arithmetic theory solvers check satisfiability of partial assignments using variations of the simplex algorithm(e.g., [80]), while difference logic solvers may perform satisfiability checking on partial assignments by testing for cycles in an associated constraint graph [144, 200]).

The correctness of Algorithm 3 relies on Lemma 4.1.1, which relates models of atoms which are comparisons to constants, and models of atoms of positive monotonic predicates. We observe that the following lemma holds:

Lemma 4.1.1 (Model Monotonicity)**.** *Let $\mathcal{M}_A$ be an assignment only to atoms comparing variables to constants.*[6] *For any given variable x with sort σ, any two constants $\sigma_i \leq \sigma_j$, and any positive monotonic predicate atom p:*[7]

$$\mathcal{M}_A \cup \{(x \geq \sigma_i)\} \underset{T}{\Rightarrow} p \quad \rightarrow \quad \mathcal{M}_A \cup \{(x \geq \sigma_j)\} \underset{T}{\Rightarrow} p \tag{4.1}$$

$$\mathcal{M}_A \cup \{(x \leq \sigma_j)\} \underset{T}{\Rightarrow} \neg p \quad \rightarrow \quad \mathcal{M}_A \cup \{(x \leq \sigma_i)\} \underset{T}{\Rightarrow} \neg p \tag{4.2}$$

A proof of this lemma can be found in Appendix C.1; the lemma easily generalizes for predicates of mixed positive and negative monotonicity.

Algorithm 3 constructs, for each variable x, an over-approximation constant x^+, and an under-approximation constant x^-. It also constructs an under-approximate assignment $\mathcal{M}_A^-$ in which for each variable x, $(x \leq x^-) \in \mathcal{M}_A^-$ and $(x \geq x^-) \in \mathcal{M}_A^-$, forcing the evaluation of x to be exactly x^- in $\mathcal{M}_A^-$. Similarly, Algorithm 3 creates an over-approximate model $\mathcal{M}_A^+$ forcing each variable x to x^+. Note that both of these assignments contain only atoms that compare variables to constants, and hence match the prerequisites of Lemma 4.1.1.

By Lemma 4.1.1, if $\mathcal{M}_A^- \underset{T}{\Longrightarrow} p$, then $\mathcal{M} \underset{T}{\Longrightarrow} p$. Also by Lemma 4.1.1, if $\mathcal{M}_A^+ \underset{T}{\Longrightarrow} \neg p$, then $\mathcal{M} \underset{T}{\Longrightarrow} \neg p$. By evaluating each predicate in $\mathcal{M}_A^-$ and $\mathcal{M}_A^+$, Algorithm 3 safely under- and over-approximates the truth value of each p in $\mathcal{M}$, using only two concrete evaluations of each predicate atom.

Lemma 4.1.1 guarantees that Algorithm 3 returns FALSE only if $T \models \neg\mathcal{M}$; however, it does not guarantee that Algorithm 3 returns FALSE for all unsatisfiable partial assignments. In the case where $\mathcal{M}_A$ is a complete and consistent assignment to every variable x, we have $x^- = x^+$ for each x (and hence $\mathcal{M}_A^- = \mathcal{M}_A^+$), and so for each p, it must be the case that either $p(x_0^-, x_1^-, \ldots) \mapsto$ TRUE or $p(x_0^+, x_1^+, \ldots) \mapsto$ FALSE. Therefore, if $\mathcal{M}_A$ is a complete assignment, Algorithm 3 returns FALSE if and only if $T \models \neg\mathcal{M}$.

[6] $\mathcal{M}_A$ contains *only* atoms of comparisons of variables to constants. Comparisons between two non-constant variables are treated as monotonic predicates, and are not included in $\mathcal{M}_A$. Also, for simplicity of presentation (but without loss of generality), we assume that all comparison atoms in $\mathcal{M}_A$ are normalized to the form $(x \geq \sigma_i)$, or $(x \leq \sigma_i)$. In the special case that atom $(x < \sigma_\perp)$ or $(x > \sigma_\top)$ (or, equivalently, theory literals $\neg(x \geq \sigma_\perp), \neg(x \leq \sigma_\top)$) are in $\mathcal{M}_A$, the assignment is trivially unsatisfiable, but cannot be normalized to inclusive comparisons.

[7] Where $A \underset{T}{\Longrightarrow} B$ is shorthand for '$(A \wedge \neg B)$ is unsatisfiable in theory T'.

Algorithm 3 is guaranteed to be complete only when $\mathcal{M}_A^+ = \mathcal{M}_A^-$, which is only guaranteed to occur when $\mathcal{M}$ is a complete assignment to the comparison atoms (but not necessarily a complete assignment to the theory predicates). If $\mathcal{M}_A^+ \neq \mathcal{M}_A^-$, then it is possible for Algorithm 3 to fail to deduce conflicts, or unassigned atoms, that are in fact implied by the theory (in the terminology of Section 2.4, Algorithm 3 is not *deduction complete*). For example, Algorithm 3 may fail to deduce non-comparison atom p_1 from a second non-comparison atom p_0 in the case that $p_0 \underset{T}{\Longrightarrow} p_1$.

For this reason, Algorithm 3 must be paired with a procedure for searching the space of possible assignments to A in order to form a complete decision procedure. For finite domain theories, this does not pose a problem, as one can either perform internal case splitting in the theory solver, or simply encode the space of possible assignments to each variable in fresh Boolean variables (for example, as a unary or binary encoding over the space of possible values in the finite domain of σ), allowing the SAT solver to enumerate that (finite) space of assignments (an example of *splitting-on-demand* [28]). In practice, we have always chosen the latter option, and found it to work well across a wide range of theories.

As we will show in subsequent chapters, Algorithm 3, when combined with an appropriate conflict analysis procedure, is sufficient to build efficient solvers for a wide set of useful theories.

Returning to our earlier example from Section 3.1 of a pseudo-Boolean constraint predicate $\sum_{i=0}^{n-1} c_i b_i \geq c_n$, with Boolean arguments b_i and constants $c_0, \ldots c_n$, Algorithm 3 could be implemented as shown in Algorithm 5.

Our presentation of Algorithm 3 above is stateless: the **UpdateBounds** section recomputes the upper- and lower-approximation assignments at each call to *T-Propagate*. In a practical implementation, this would be enormously wasteful, as the under- and over-approximate assignments will typically be expected to change only in small ways between theory propagate calls. In a practical implementation, one would store the upper- and lower-approximation assignments between calls to *T-Propagate*, and only alter them as theory literals are assigned or unassigned in $\mathcal{M}$. A practical, stateful implementation of Algorithm 3 is described in more detail in Appendix A.

For example, the pseudo-code in Algorithm 5 is a nearly complete representation of the actual implementation of theory propagation for pseudo-Boolean constraints that we have implemented in our SMT solver, with the only difference being that in our implementation we maintain running under- and over-approximate sums ($\sum_{i=0}^{n-1} c_i b_i^-, \sum_{i=0}^{n-1} c_i b_i^+$), updated each time a theory literal is assigned or unassigned in the SAT solver, rather than re-calculating them at each *T-Propagate* call.

Algorithm 3 Theory propagation for function-free monotonic theories, *assuming all predicates are positive monotonic, and all sorts are totally ordered.* Algorithm 3 takes a partial assignment $\mathcal{M}$ to the theory atoms of T. If $\mathcal{M}$ is found to be unsatisfiable in T, it returns a tuple $(\textsc{False}, c)$, with c a conflict set produced by *T-Analyze*; else it returns $(\textsc{True}, \mathcal{M})$, with any deduced literals added to $\mathcal{M}$.

```
function T-Propagate($\mathcal{M}$)
    UpdateBounds:
    $\mathcal{M}_A^- \leftarrow \{\}, \mathcal{M}_A^+ \leftarrow \{\}$
    for each sort $\sigma \in T$ do
        for each variable $x$ of sort $\sigma$ do
            if $(x < \sigma_\bot) \in \mathcal{M}$ or $(x > \sigma_\top) \in \mathcal{M}$ then
                return FALSE, T-Analyze($\mathcal{M}$)
            $\mathcal{M} \leftarrow \mathcal{M} \cup \{(x \geq \sigma_\bot), (x \leq \sigma_\top)\}$
            $x^- \leftarrow \max(\{\sigma_i | (x \geq \sigma_i) \in \mathcal{M}\})$.
            $x^+ \leftarrow \min(\{\sigma_i | (x \leq \sigma_i) \in \mathcal{M}\})$.
            if $x^- > x^+$ then return FALSE, T-Analyze($\mathcal{M}$)
            $\mathcal{M}_A^- \leftarrow \mathcal{M}_A^- \cup \{(x \leq x^-), (x \geq x^-)\}$
            $\mathcal{M}_A^+ \leftarrow \mathcal{M}_A^+ \cup \{(x \leq x^+), (x \geq x^+)\}$
    PropagateBounds:
    for each positive monotonic predicate atom $p(x_0, x_1, \ldots)$ do
        if $\neg p \in \mathcal{M}$ then
            if $evaluate(p(x_0^-, x_1^-, \ldots)) \mapsto$ TRUE then
                return FALSE, T-Analyze($\mathcal{M}$)
            else
                Tighten bounds (optional)
                $\mathcal{M} \leftarrow$ TIGHTENBOUNDS$(p, \mathcal{M}, \mathcal{M}_A^+, \mathcal{M}_A^-)$
        else if $p \in \mathcal{M}$ then
            if $evaluate(p(x_0^+, x_1^+, \ldots)) \mapsto$ FALSE then
                return FALSE, T-Analyze($\mathcal{M}$)
            else
                Tighten bounds (optional)
                $\mathcal{M} \leftarrow$ TIGHTENBOUNDS$(\neg p, \mathcal{M}, \mathcal{M}_A^-, \mathcal{M}_A^+)$
        else
            Deduce unassigned predicate atoms:
            if $evaluate(p(x_0^-, x_1^-, \ldots)) \mapsto$ TRUE then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{p\}$
            else if $evaluate(p(x_0^+, x_1^+, \ldots)) \mapsto$ FALSE then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{\neg p\}$
    return TRUE, $\mathcal{M}$
```

Algorithm 4 TIGHTENBOUNDS is an optional routine which may be called by implementations of Algorithm 3. TIGHTENBOUNDS takes a predicate p that has been assigned a truth value, and performs a search over the arguments x_i of p to find tighter bounds on x_i that can be added to $\mathcal{M}$.

```
function TIGHTENBOUNDS($p(x_0, x_1, \dots, x_n), \mathcal{M}, \mathcal{M}_A^+, \mathcal{M}_A^-$)
    for each $x_0 \dots x_n$ do
        if $p$ is positive monotonic in argument $i$ (which has sort $\sigma$) then
            for each $y \in \sigma, \mathcal{M}_A^-[x_i] < y \leq \mathcal{M}_A^+[x_i]$ do
                if $evaluate(p(\dots, \mathcal{M}_A^-[x_{i-1}], y, \mathcal{M}_A^-[x_{i+1}], \dots)) \mapsto$ TRUE then
                    $\mathcal{M} \leftarrow \mathcal{M} \cup \{(x_i < y)\}$
        else
            for each $y \in \sigma, \mathcal{M}_A^+[x_i] > y \geq \mathcal{M}_A^-[x_i]$ do
                if $evaluate(p(\dots, \mathcal{M}_A^+[x_{i-1}], y, \mathcal{M}_A^+[x_{i+1}], \dots)) \mapsto$ TRUE then
                    $\mathcal{M} \leftarrow \mathcal{M} \cup \{(x_i > y)\}$
    return $\mathcal{M}$
```

Algorithm 5 Instantiation of Algorithm 3 for the theory of pseudo-Boolean constraints. A practical implementation would store the under- and over-approximate sums between calls to *T-Propagate*, updating them as theory literals are assigned, rather than recalculating them each time.

```
function T-Propagate($\mathcal{M}$)
    UpdateBounds:
    for each argument $b_i$ do
        $b_i^- \leftarrow$ FALSE, $b_i^+ \leftarrow$ TRUE
        if $b_i \in \mathcal{M}$ then
            $b_i^- \leftarrow$ TRUE
        else if $\neg b_i \in \mathcal{M}$ then
            $b_i^+ \leftarrow$ FALSE
    PropagateBounds:
    for each predicate atom $p = \sum_{i=0}^{n-1} c_i b_i \geq c_n$ do
        if $\neg p \in \mathcal{M}$ then
            if $\sum_{i=0}^{n-1} c_i b_i^- \geq c_n$ then
                return FALSE, T-Analyze($\mathcal{M}$)
            else
                TightenBounds:
                for each $b_j$ do
                    if $b_j^+ \wedge \neg b_j^- \wedge \sum_{i=0}^{n-1} c_i b_i^- + c_j \geq c_n$ then
                        $\mathcal{M} \leftarrow \mathcal{M} \cup \{\neg b_j\}$
        else if $p \in \mathcal{M}$ then
            if $\sum_{i=0}^{n-1} c_i b_i^+ < c_n$ then
                return FALSE, T-Analyze($\mathcal{M}$)
            else
                TightenBounds:
                for each $b_j$ do
                    if $b_j^+ \wedge \neg b_j^- \wedge \sum_{i=0}^{n-1} c_i b_i^+ - c_j < c_n$ then
                        $\mathcal{M} \leftarrow \mathcal{M} \cup \{b_j\}$
        else
            if $\sum_{i=0}^{n-1} c_i b_i^- \geq c_n$ then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{p\}$
            else if $\sum_{i=0}^{n-1} c_i b_i^+ < c_n$ then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{\neg p\}$
    return TRUE, $\mathcal{M}$
```

4.2 Conflict Analysis for Finite Monotonic Theories

In Section 4.1, we described a technique for theory propagation in function-free, totally-ordered finite monotonic theories. As discussed at the opening of this chapter, the other function that efficient SMT solvers must implement is *T-Analyze*, which takes a partial assignment $\mathcal{M}$ that is unsatisfiable in theory T, and returns a subset of $\mathcal{M}$ (a 'conflict set') which remains unsatisfiable. Ideally, this conflict set will be both small, and efficient to derive.

All SMT solvers have the option of returning the naïve conflict set, which is just to return the entire $\mathcal{M}$ as the conflict set. However, for the special case of conflicts derived by Algorithm 3, we describe an improvement upon the naïve conflict set, which we call the default monotonic conflict set. So long as Algorithm 3 is used, returning the default monotonic conflict set is always an option (and is never worse than returning the naïve monotonic conflict set).

While in many cases we have found that theory specific reasoning allows one to find even better conflict sets than this default monotonic conflict set, in some cases, including the theory of pseudo-Booleans described in Algorithm 5, as well as some of the predicates mentioned in subsequent chapters, our implementation does in fact fall back on the default monotonic conflict set.

We describe our algorithm for deriving a default monotonic conflict set in Algorithm 6. The first for-loop of Algorithm 6 simply handles the trivial conflicts that can arise when building the over- and under-approximations of each variable x^+, x^- in Algorithm 3 (for example, if $\mathcal{M}$ is inconsistent with the total order relation of σ); these cases are all trivial and self-explanatory.

In the last 5 lines, Algorithm 6 deals with conflicts in which the under- or over-approximations x^-, x^+ are in conflict with assignment in $\mathcal{M}$ of a single predicate atom, p. Consider a predicate atom $p(x_0, x_1)$, with p positive monotonic in each argument. Algorithm 3 can derive a conflict involving p in one of only two cases: Either $p(x_0^+, x_1^+)$ evaluates to FALSE, when $p(x_0, x_1)$ is assigned TRUE in $\mathcal{M}$, or $p(x_0^-, x_1^-)$ evaluates to TRUE, when $p(x_0, x_1)$ is assigned FALSE in $\mathcal{M}$. In the first case, a sufficient conflict set can be formed from just the literals in $\mathcal{M}$ used to set the upper bounds $x_0^+, x_1^+, \ldots$ (along with the conflicting predicate literal $p(x_0, x_1)$). In the second case, only the atoms of $\mathcal{M}$ that were used to form the

Algorithm 6 Conflict set generation for finite monotonic theories. *Assuming all predicates are positive monotonic.* Algorithm 6 takes $\mathcal{M}$, a (partial) assignment that is unsatisfiable in T, and returns a conflict set, a subset of the atoms assigned in $\mathcal{M}$ that are mutually unsatisfiable in T.

```
function T-Analyze($\mathcal{M}$)
    for each Variable $x$ of sort $\sigma$ do
        Compute $x^-, x^+$ as in T-Propagate.
        case $(x < \sigma_\bot) \in \mathcal{M}$
            return $\{(x < \sigma_\bot)\}$
        case $(x > \sigma_\bot) \in \mathcal{M}$
            return $\{(x > \sigma_\bot)\}$
        case $\nexists \sigma_i : (\sigma_i \geq x^-) \land (\sigma_i \leq x^+)$
            return $\{(x \geq x^-), (x \leq x^+)\}$
    for each monotonic predicate atom $p(t_0, t_1, \ldots)$ do
        case $\neg p \in \mathcal{M}$, $evaluate(p(x_0^-, x_1^-, \ldots)) \mapsto$ TRUE
            return $\{\neg p, (x_0 \geq x_0^-), (x_1 \geq x_1^-), \ldots\}$
        case $p \in \mathcal{M}$, $evaluate(p(x_0^+, x_1^+, \ldots)) \mapsto$ FALSE
            return $\{p, (x_0 \leq x_0^+), (x_1 \leq x_1^+), \ldots\}$
```

under-approximations x_0^-, x_1^- need to be included in the conflict set (along with the negated predicate atom). In other words, when Algorithm 3 finds a non-trivial conflict on monotonic predicate atom p, the default monotonic conflict set can exclude either all of the over-approximate atoms, or all of the under-approximate atoms, from $\mathcal{M}$ (except for the atom of the conflicting predicate atom p) — improving over the naïve conflict set, which does not drop these atoms.

Formally, the correctness of this conflict analysis procedure follows from Lemma 4.1.1, in the previous section. As before, let $\mathcal{M}_A$ be an assignment that contains only the atoms of $\mathcal{M}$ comparing variables to constants: $\{(x \leq \sigma_i), (x \geq \sigma_i)\}$. By Lemma 4.1.1, if $\mathcal{M}_A^- \underset{T}{\Longrightarrow} p$, then the comparison atoms $(x_i \geq \mathcal{M}_A^-[x_i])$ in $\mathcal{M}_A$, for the arguments $x_0, x_1, \ldots$ of p, are sufficient to imply p by themselves. Also by Lemma 4.1.1, if $\mathcal{M}_A^+ \underset{T}{\Longrightarrow} \neg p$, then the comparison atoms $(x_i \leq \mathcal{M}_A^+[x_i])$ in $\mathcal{M}_A$, for the arguments $x_0, x_1, \ldots$ of p, are sufficient to imply $\neg p$ by themselves. Therefore, the lower-bound (resp. upper-bound) comparison atoms in $\mathcal{M}_A$ that are arguments of p can safely form justification sets for p (resp. $\neg p$).[8]

The default monotonic conflict set is always available when Algorithm 3 is used, and improves greatly on the naïve conflict set, but it is not required (or even recommended) in most cases. In practice, it is often possible to improve on this default monotonic conflict set, for example by discovering that looser constraints than $\mathcal{M}_A^-[x_i]$ or $\mathcal{M}_A^+[x_i]$ would be sufficient to imply the conflict, or that some arguments of p are irrelevant.

Many common algorithms are constructive in the sense that they not only compute whether p is true or false, but also produce a witness (in terms of the inputs of the algorithm) that is a sufficient condition to imply that property. In many cases, the witness will be constructed strictly in terms of inputs corresponding to atoms that are assigned TRUE (or alternatively, strictly in terms of atoms that are assigned FALSE). This need not be the case — the algorithm might not be constructive, or it might construct a witness in terms of some combination of the true and false atoms in the assignment — but, as we will show in Chapters 5, 7, and 8, it commonly *is* the case for many theories of interest. For example, if we used depth-first search to find that node v is reachable from node u in some graph, then we obtain as a side effect a path from u to v, and the theory atoms corresponding to the edges in that path imply that v can be reached from u.

Any algorithm that can produce a witness containing only $(x \geq \sigma_i)$ atoms can be used to produce conflict sets for any positive predicate atom assignments propagated from the under-approximation arguments above. Similarly, any algorithm that can produce a witness of $(x \leq \sigma_i)$ atoms can also produce conflict sets for any negative predicate atom assignments propagated from over-approximation arguments above. Some algorithms can produce both. In practice, we have often found standard algorithms that produce one, but not both, types of witnesses. In cases where improved conflict analysis procedures are not available, one can always safely fall back on the default monotonic conflict analysis procedure of Algorithm 6.

[8] Note that we can safely drop any comparisons $(x \geq \sigma_\bot)$ or $(x \leq \sigma_\top)$ from the conflict set.

4.3 Compositions of Monotonic Functions

In the previous two sections, we described algorithms for *T-Propagate* and *T-Analyze*, for the special case of function-free finite theories with totally ordered sorts and positive monotonic predicates. We now extend that approach in two ways: first, to support both positive and negative predicates, and second, to support both positive and negative monotonic functions. In fact, the assumption in Algorithm 3 that all predicates are positive monotonic in all arguments was was simply for ease of presentation, as handling predicates that are negative monotonic, or have mixed monotonicity (positive monotonic in some arguments, negative in others) is straightforward. However, dealing with compositions of positive and negative monotonic multivariate functions (rather than predicates) is more challenging, as a composition of positive and negative monotonic functions is not itself guaranteed to be monotonic.

For example, consider the functions $f(x,y) = x+y$, $g(x) = 2x$, $h(x) = -x^3$. Even though each of these is either a positive or a negative monotonic function, the composition $f(g(x),h(x))$ is non-monotonic in x. As a result, a naïve approach of evaluating each term in the over- or under-approximative assignments might not produce safe upper and lower bounds (e.g., $f(g(\mathcal{M}_A^+[x]),h(\mathcal{M}_A^+[x]))$ may not be a safe over-approximation of $f(g(x),h(x))$).

One approach to resolve this difficulty would be to flatten function compositions through the introduction of fresh auxiliary variables, and then to replace all functions with equivalent predicates relating the inputs and outputs of those functions (this is the approach we have taken to support bitvectors, as described in Chapter 5). After such a transformation, the resulting formula no longer has any non-predicate functions, and as each of the newly introduced predicates are themselves monotonic, the translated formula can be handled by Algorithm 3. A drawback to this approach is that Algorithm 3 only considers each predicate atom on its own, and does not directly reason about interactions among these predicates, instead repeatedly propagating derived upper and lower bounds on arguments of the predicates back to the SAT solver. This may result in a large number of repeated calls to Algorithm 3 as $\mathcal{M}$ is gradually refined by updating the bounds on the arguments of each predicate until a fixed point is reached, which may result in

Algorithm 7 Recursively, builds up a safe, approximate evaluation of composed positive and negative monotonic functions and predicates. If $\mathcal{M}_A^+$ is an over-approximative assignment, and $\mathcal{M}_A^-$ an under-approximative assignment, then APPROX returns a safe over-approximation of the evaluation of ϕ. If $\mathcal{M}_A^+$ is instead an *under*-approximation, and $\mathcal{M}_A^-$ an over-approximation, then APPROX returns a safe *under*-approximation of the evaluation of ϕ.

```
function APPROX($\phi$,$\mathcal{M}_A^+$,$\mathcal{M}_A^-$)
    $\phi$ is a formula, $\mathcal{M}_A^+$,$\mathcal{M}_A^-$ are assignments.
    if $\phi$ is a variable or constant term then
        return $\mathcal{M}_A^+[\phi]$
    else $\phi$ is a function term or predicate atom $f(t_0,t_1,\dots,t_n)$
        for $0 \leq i \leq n$ do
            if $f$ is positive monotonic in $t_i$ then
                $x_i =$ APPROX($t_i,\mathcal{M}_A^+,\mathcal{M}_A^-$)
            else arguments $\mathcal{M}_A^+,\mathcal{M}_A^-$ are swapped
                $x_i =$ APPROX($t_i,\mathcal{M}_A^-,\mathcal{M}_A^+$)
        return $evaluate(f(x_0,x_1,x_2,\dots,x_n))$
```

unacceptably slow performance for large formulas.

As an alternative, we introduce support for compositions of monotonic functions by evaluating compositions of functions approximately, without introducing auxiliary variables, in such a way that the approximation remains monotonic under partial assignments, while under a complete assignment the approximation converges to the correct value.

To do so, we introduce the function $approx(\phi,\mathcal{M}_A^+,\mathcal{M}_A^-)$, presented in Algorithm 7. This function can be used to form both safe over-approximations and safe under-approximations of compositions of both positive and negative monotonic functions. Function $approx(\phi,\mathcal{M}_A^+,\mathcal{M}_A^-)$ takes a term ϕ, which is either a variable, a constant, or a monotonic function or predicate (either positive or negative monotonic, or a mixture thereof); and in which $\mathcal{M}_A^+,\mathcal{M}_A^-$ are both complete assignments to the variables in ϕ. Intuitively, if $\mathcal{M}_A^+$ is a safe over-approximation to the variables of ϕ, and $\mathcal{M}_A^-$ a safe under-approximation, then $approx(\phi,\mathcal{M}_A^+,\mathcal{M}_A^-)$ will return a safe over-approximation of ϕ. Conversely, $approx(\phi,\mathcal{M}_A^-,\mathcal{M}_A^+)$ (swapping the 2nd and 3rd arguments) will return a safe *under*-approximation of ϕ. Further, if both assignments are identical, then $approx(\phi,\mathcal{M}_A,\mathcal{M}_A)$ returns an exact

Algorithm 8 Replacement for **PropagateBounds** section of Algorithm 3. *Supporting compositions of mixed positive and negative monotonic predicates and functions.*

```
function PROPAGATEBOUNDS($\mathcal{M}, \mathcal{M}_A^+, \mathcal{M}_A^-$)
    $\mathcal{M}, \mathcal{M}_A^+, \mathcal{M}_A^-$ are assignments, with $\mathcal{M}_A^+, \mathcal{M}_A^-$ computed by UpdateBounds.
    for each monotonic predicate atom $p(t_0, t_1, \ldots)$ do
        if $\neg p \in \mathcal{M}$ then
            if APPROX($p, \mathcal{M}_A^-, \mathcal{M}_A^+$) $\mapsto$ TRUE then
                return FALSE, T-Analyze($\mathcal{M}$)
            else
                Tighten bounds (optional)
                $\mathcal{M} \leftarrow$ TIGHTENBOUNDS($p, \mathcal{M}, \mathcal{M}_A^+, \mathcal{M}_A^-$)
        else if $p \in \mathcal{M}$ then
            if APPROX($p, \mathcal{M}_A^+, \mathcal{M}_A^-$) $\mapsto$ FALSE then
                return FALSE, T-Analyze($\mathcal{M}$)
            else
                Tighten bounds (optional)
                $\mathcal{M} \leftarrow$ TIGHTENBOUNDS($\neg p, \mathcal{M}, \mathcal{M}_A^-, \mathcal{M}_A^+$)
        else
            if APPROX($p, \mathcal{M}_A^-, \mathcal{M}_A^+$) $\mapsto$ TRUE then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{p\}$
            else if APPROX($p, \mathcal{M}_A^+, \mathcal{M}_A^-$) $\mapsto$ FALSE then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{\neg p\}$
    return TRUE, $\mathcal{M}$
```

evaluation of ϕ in $\mathcal{M}_A$.

Formally, if we have some $\mathcal{M}_A^*$ such that $\forall x, \mathcal{M}_A^+[x] \geq \mathcal{M}_A^*[x] \geq \mathcal{M}_A^-[x]$, then $approx(\phi, \mathcal{M}_A^+, \mathcal{M}_A^-) \geq \mathcal{M}_A^*[\phi] \geq approx(\phi, \mathcal{M}_A^-, \mathcal{M}_A^+)$. A proof can be found in Appendix C.2.

As $approx(\phi, \mathcal{M}_A^+, \mathcal{M}_A^-)$ and $approx(\phi, \mathcal{M}_A^-, \mathcal{M}_A^+)$ form safe, monotonic, upper- and lower-bounds on the evaluation of $\mathcal{M}_A[\phi]$, we can directly substitute *approx* into Algorithm 3 in order to support compositions of mixed positive and negative monotonic functions and predicates. These changes are described in Algorithm 8.

In Algorithm 9, we introduce corresponding changes to the conflict set generation algorithm in order to support Algorithm 8. Algorithm 9 makes repeated calls

Algorithm 9 Conflict set generation for finite monotonic theories with functions. *Allowing mixed positive and negative monotonic predicates and functions.*

```
function T-Analyze($\mathcal{M}$)
    $\mathcal{M}$ is a partial assignment.
    Compute $\mathcal{M}_A^-, \mathcal{M}_A^+$ as in UpdateBounds.
    for each Variable $x$ of sort $\sigma$ do
        case $x < \sigma_\perp \in \mathcal{M}$
            return $\{x < \sigma_\perp\}$
        case $x > \sigma_\perp \in \mathcal{M}$
            return $\{x > \sigma_\perp\}$
        case $\nexists \sigma_i : (\sigma_i \geq \mathcal{M}_A^-[x]) \land (\sigma_i \leq \mathcal{M}_A^+[x])$
            return $\{x \geq \mathcal{M}_A^-[x], x \leq \mathcal{M}_A^+[x]\}$
    for each monotonic predicate atom $p$ do
        case $\neg p \in \mathcal{M}$, $\textsc{approx}(p, \mathcal{M}_A^-, \mathcal{M}_A^+) \mapsto \textsc{True}$
            return $\{\neg p\} \cup \textsc{analyzeApprox}(p, \leq, \textsc{False}, \mathcal{M}_A^-, \mathcal{M}_A^+)$
        case $p \in \mathcal{M}$, $\textsc{approx}(p, \mathcal{M}_A^+, \mathcal{M}_A^-) \mapsto \textsc{False}$
            return $\{p\} \cup \textsc{analyzeApprox}(p, \geq, \textsc{True}, \mathcal{M}_A^+, \mathcal{M}_A^-)$
```

to a recursive procedure, *analyzeApprox* (shown in Algorithm 10). Algorithm 10 produces a conflict set in terms of the upper and lower bounds of the variables x in $\mathcal{M}_A^+$ and $\mathcal{M}_A^-$. Algorithm 10 recursively evaluates each predicate and function on its respective upper and lower bounds, swapping the upper and lower bounds when evaluating negatively monotonic arguments.

Notice that if p is positive monotonic, and t_i is a ground variable, then Algorithm 9 simply returns the same atom $t_i \geq \sigma_i$ or $t_i \leq \sigma_i$, for some constant σ_i, as it would in Algorithm 6. As such, Algorithm 9 directly generalizes Algorithm 6.

Algorithm 10 The analyzeApprox function provides conflict analysis for the recursive *approx* function.

```
function ANALYZEAPPROX($\phi, op, k, \mathcal{M}_A^+, \mathcal{M}_A^-$)
    $\phi$ is a formula, op is a comparison operator, k is a constant, $\mathcal{M}_A^+, \mathcal{M}_A^-$ are assignments.
    if $\phi$ is a constant term then
        return $\{\}$
    else if $\phi$ is a variable then
        if $op$ is $\leq$ then
            return $\{\phi \leq k\}$
        else
            return $\{\phi \geq k\}$
    else $\phi$ is a function or predicate term $f(t_0, t_1, \dots, t_n)$
        $c \leftarrow \{\}$
        for $0 \leq i \leq n$ do
            if $f$ is positive monotonic in $t_i$ then
                $x_i =$ ANALYZEAPPROX($t_i, \mathcal{M}_A^+, \mathcal{M}_A^-$)
                $c \leftarrow c \cup$ ANALYZEAPPROX($t_i, op, x_i, \mathcal{M}_A^+, \mathcal{M}_A^-$)
            else arguments $\mathcal{M}_A^+, \mathcal{M}_A^-$ are swapped
                $x_i =$ ANALYZEAPPROX($t_i, \mathcal{M}_A^-, \mathcal{M}_A^+$)
                $c \leftarrow c \cup$ ANALYZEAPPROX($t_i, -op, x_i, \mathcal{M}_A^-, \mathcal{M}_A^+$)
        return $c$
```

4.4 Monotonic Theories of Partially Ordered Sorts

Algorithms 3 and 8, as presented above, assume finite, totally ordered sorts. However, observing that Lemma 4.1.1 applies also to partial-orders, Algorithms 3 and 8 can also be modified to handle finite, partially-ordered sorts.

Conceptually, theory propagation applied to partial orders proceeds similarly to theory propagation as applied to total orders, with two major changes. Firstly, checking whether the comparison atoms in $\mathcal{M}$ are consistent is slightly more involved (and, correspondingly, conflict analysis for the case where the comparison atoms are inconsistent is also more involved). Secondly, even in cases where the comparison atoms are consistent, in a partial order there may not exist a constant x^+ to form the upper bound for variable x (or there may not exist a lower bound x^-); or those bounds may exist, but be very 'loose', and hence the theory propagation procedure may fail to make deductions that are in fact implied by the comparison atoms of $\mathcal{M}$.

For example, if σ is partially ordered, $\neg(x \geq y)$ does not necessarily imply $x < y$, as it may also be possible for x and y to be assigned incomparable values; consequently, it may be possible for both atoms $\neg(x \geq y), \neg(x \leq y)$ to be in $\mathcal{M}_A$ without implying a conflict. This causes complications for conflict analysis, and also for the forming of upper and lower bounds needed by Algorithms 3 and 8.

In Algorithm 11, we describe modifications to the **UpdateBounds** section of Algorithm 3 to support partially ordered sorts, by forming conservative lower and upper bounds x^- and x^+, where possible, and returning without making deductions when such bounds do not exist. These changes are also compatible with the changes to **PropagateBounds** described in Algorithm 8, and by combining both, one can support theory propagation for finite monotonic theories with functions and partially ordered sorts.

Unfortunately, if σ is only a partial order, then it is possible that either or both of $meet(max(X^-))$, $join(min(X^+))$ may not exist. In the case that either or both do not exist, Algorithm 11 simply returns, making no deductions at all. Consequently, for partial orders, Algorithm 11 may fail to make deductions that are in fact implied by the comparisons of $\mathcal{M}_A$ (here, we define $\mathcal{M}_A$ as we did in previous sections: $\mathcal{M}_A$ is a subset of $\mathcal{M}$ containing assignments only to atoms comparing variables to constants ($(x \leq \sigma_i)$, $(x \geq \sigma_i)$). In fact, even in the case where safe upper and lower bounds do exist, there may exist deductions $\mathcal{M}_A \underset{T}{\Longrightarrow} p$ (or $\mathcal{M}_A \underset{T}{\Longrightarrow} \neg p$) that this algorithm fails to discover. In the special case that σ is totally ordered, X^+ and X^- can always be represented as singletons (or the empty set, if there is a conflict), $max(X^-)$ and $min(X^+)$ will always return single constants, the meet and join will always be defined, and the resulting construction of x^+, x^- becomes completely equivalent to our presentation in Algorithm 3.

The imprecision of Algorithm 11's deductions is a consequence of reducing the upper and lower bounds to singletons, which allows the upper and lower bounds of each predicate p to be checked with only two evaluations of p (one for the upper bound, and one for the lower bound). An alternative approach, which would discover all deductions $\mathcal{M}_A \underset{T}{\Longrightarrow} p$ (or $\mathcal{M}_A \underset{T}{\Longrightarrow} \neg p$), would be to search over the space of (incomparable) maximal and minimal elements $max(x^+)$ and $min(x^-)$ for each variable x. In principle, by evaluating all possible combinations of maximal and minimal assignments to each argument of p, one could find the absolute maximum and minimum evaluations of p. However, while there are general techniques for optimizing functions over posets (see *ordinal optimization*, e.g. [78]), implementing such a search would likely be impractical unless the domain of σ is very small.

Above, we described a general purpose approach to handling monotonic theories over finite-domain partially ordered sorts. An alternative approach to supporting finite partially ordered sorts would be to convert the partial order into a total order (e.g., using a topological sorting procedure), after which one could directly apply the approach of Section 4.1. Unfortunately, strengthening the order relationship between elements of the sort may force the solver to learn weaker clauses than it otherwise would. For example, if $a \leq b$ holds in the total order, then Algorithm 6 may in some cases include that (spurious) comparison atom in learned clauses, even if a and b are in fact incomparable in the original partial order.

An important special case of a partially ordered sort is a finite powerset lattice. If the sort is a powerset lattice, then we can translate the space of possible subsets into a bit string of Boolean variables, with each Boolean determining the presence of each possible element in the set. In this case, the size of the represented set is monotonic with respect to not only the value of the bit string, but also with respect to each individual Boolean making up that string. This transformation allows one to model the powerset lattice as a totally ordered, monotonic theory over the individual Booleans of the bit string, and hence to perform theory propagation using Algorithms 3 or 8. This is the approach that we take in our implementation of the graph and finite state machine theories in Chapters 5 and 8, which operate over sets of graphs and Kripke structures, respectively. We have found it to perform very well in practice.

Algorithm 11 Replacement for **UpdateBounds** section of Algorithm 3, supporting theory propagation for finite monotonic theories over partially ordered sorts.

```
function UPDATEBOUNDS($\mathcal{M}$)
    $\mathcal{M}_A^- \leftarrow \{\}, \mathcal{M}_A^+ \leftarrow \{\}$
    for each variable $x$ of sort $\sigma$ do
        $X^+ \leftarrow \{\sigma_\bot, \sigma_1, \sigma_2 \ldots \sigma_\top\}$
        $X^- \leftarrow \{\sigma_\bot, \sigma_1, \sigma_2 \ldots \sigma_\top\}$
        for each $(x > \sigma_j)$ in $\mathcal{M}$ do
            $X^- = X^- \setminus \{\forall \sigma_i | \sigma_i \leq \sigma_j\}$
        for each $(x \geq \sigma_j)$ in $\mathcal{M}$ do
            $X^- = X^- \setminus \{\forall \sigma_i | \sigma_i < \sigma_j\}$
        for each $\neg(x < \sigma_j)$ in $\mathcal{M}$ do
            $X^- = X^- \setminus \{\forall \sigma_i | \sigma_i < \sigma_j\}$
        for each $\neg(x \leq \sigma_j)$ in $\mathcal{M}$ do
            $X^- = X^- \setminus \{\forall \sigma_i | \sigma_i \leq \sigma_j\}$
        for each $(x < \sigma_j)$ in $\mathcal{M}$ do
            $X^+ = X^+ \setminus \{\forall \sigma_i | \sigma_i \geq \sigma_j\}$
        for each $(x \leq \sigma_j)$ in $\mathcal{M}$ do
            $X^+ = X^+ \setminus \{\forall \sigma_i | \sigma_i > \sigma_j\}$
        for each $\neg(x > \sigma_j)$ in $\mathcal{M}$ do
            $X^+ = X^+ \setminus \{\forall \sigma_i | \sigma_i > \sigma_j\}$
        for each $\neg(x \geq \sigma_j)$ in $\mathcal{M}$ do
            $X^+ = X^+ \setminus \{\forall \sigma_i | \sigma_i \geq \sigma_j\}$
        if $X^+ \cap X^- = \{\}$ then return FALSE, $\textit{T-Analyze}(\mathcal{M})$
        Note: As the minimal (resp. maximal) elements of $X^-$ (resp. $X^+$) may not
    be unique, min (resp. max) returns a set of minimal (resp. maximal) elements.
        $x^- \leftarrow meet(min(X^-))$
        $x^+ \leftarrow join(max(X^+))$
        if $x^-$ or $x^+$ does not exist, or $x^-$ and $x^+$ are incomparable then
            return TRUE, $\mathcal{M}$
        if $x^- > x^+$ then return FALSE, $\textit{T-Analyze}(\mathcal{M})$
        $\mathcal{M}_A^- \leftarrow \mathcal{M}_A^- \cup \{(x \leq x^-), (x \geq x^-)\}$
        $\mathcal{M}_A^+ \leftarrow \mathcal{M}_A^+ \cup \{(x \leq x^+), (x \geq x^+)\}$
```

4.5 Theory Combination and Infinite Domains

Two additional concerns should be touched upon before continuing, both related to the finite domain requirement of finite monotonic theories. First, we can ask, why are the techniques introduced in this section restricted to finite domain sorts?

If the domain is infinite, then in the general case the conflict analysis procedure (Algorithm 6 or Algorithm 9) may not be sufficient to guarantee termination, as it may introduce a non-converging sequence of new comparison atoms with infinitesimally tighter bounds. Unfortunately, we have found no obvious way — in the general case — to extend support to monotonic theories of infinite domains that would resolve this concern.

The second concern is theory combination. In general, finite domain sorts — including Booleans, bit vectors, and finite sets — violate the stably-infinite requirement of Nelson-Oppen [156] theory combination, which requires that a theory have, for every sort, an infinite number of satisfying assignments (models). As such, Nelson-Oppen theory combination is not directly applicable to finite monotonic theories.[9]

Finite domain theories can, in general, be combined at the propositional level by introducing fresh variables for each shared variable in the two theories, and then asserting equality at the propositional level. This can either be accomplished by passing equality predicates over shared constants between the two theories (as in Nelson-Oppen), or by enumerating over the space of possible values each variable can take and asserting that exactly the same assignment is chosen for those fresh variables.

Either of the above theory combination approaches are poor choices for monotonic theories, for the reason that it provides a very poor mechanism for communicating derived bounds on the satisfiable ranges of each variable between each theory solver, when operating on partial assignments. For example, when operating on a partial assignment, one theory solver may have derived the bounds $3 \leq x \leq 5$,

[9] Note that, even if we were to consider monotonic theories with infinite domains, we would still encounter a difficulty, which is that the theory of equality is non-monotonic, and hence monotonic theories do not directly support equality predicates. It seems hopeful that Nelson-Oppen style theory combination could be applied to theories supporting only comparisons, for example by emulating equality through conjunctions of $\leq, \geq$ constraints. However, we present no proof of this claim.

however, until x is forced to a specific value (e.g., $x = 3$), the above approach provides no way to communicate those bounds on x to other theory solvers.

Instead, we recommend a form of delayed-theory combination, over the space of comparison atoms. This approach is based on Algorithm 12, which iteratively propagates comparison atoms between two finite monotonic theory solvers for T_0, T_1. By itself, Algorithm 12 only performs theory propagation to the combined theory $T_0 \cup T_1$; it must be combined with a complete decision procedure (for example, by integrating it into an SMT solver as a theory solver), to become a complete theory combination procedure.[10]

Consider two finite theories, T_0 and T_1 with shared variables $x_0, x_1, \ldots$, and with T_0 a monotonic theory. So long as both theories support comparisons to constants ($x \geq \sigma_i, x \leq \sigma_i$), the upper and lower bounds generated by Algorithm 3 can be passed from one theory solver to the other (by lazily introducing new comparison-to-constant atoms), each time bounds are tightened during theory propagation.[11] If theory T_1 happens also to be able to compute upper and lower bounds during theory propagation (which may be the case whether or not T_1 is a monotonic theory), then those upper and lower bounds can also be added to $\mathcal{M}_A$ in T.

It is easy to see that Algorithm 12 must terminate, as at each step in which a conflict does not occur, $\mathcal{M}$ must grow to include a new comparison atom, or else the algorithm will terminate. As σ is finite, there are only a finite number of comparison atoms that can be added to $\mathcal{M}$, so termination is guaranteed. That Algorithm 12 converges to a satisfying model in which the shared variables have the same assignment in each theory solver is also easy to see: When $\mathcal{M}$ is a complete assignment, it must either be the case that either T_0 or T_1 derives a conflict and Algorithm 12 returns FALSE, or it must be the case that, for each shared variable x, $\exists \sigma_i, (x \geq \sigma_i) \in \mathcal{M} \wedge (x \leq \sigma_i) \in \mathcal{M} \wedge (x' \geq \sigma_i) \in \mathcal{M} \wedge (x' \leq \sigma_i) \in \mathcal{M}$.

As we will describe in Chapter 5, we have used Algorithm 12 to combine our graph theory with a non-wrapping theory of bitvectors, and found it to work well in practice.

[10] Note that the theory combination technique we describe is not a special case of Nelson-Oppen theory combination, as the theories in question are neither signature-disjoint nor stably-infinite.

[11] As the domain of σ is finite, there are only a finite number of new comparison atoms that can be introduced, and so we can safely, lazily introduce new comparison atoms as needed during theory propagation without running into termination concerns.

Algorithm 12 Apply theory propagation to the combination of two finite monotonic theories, $T_0 \cup T_1$. As in Nelson-Oppen, we purify expressions over shared variables by introducing a fresh variable x'_i for each shared variable x_i, such that x_i only appears in expressions over predicates or functions from a single theory, with x'_i replacing x_i in expressions over predicates or functions from the other theory. We assume both T_0 and T_1 both support comparisons to constant predicates $(x \leq \sigma_i)$,$(x \geq \sigma_i)$, and the shared set of constants $\{\sigma_\bot, \sigma_1, \ldots \sigma_\top\}$.

```
function T-Propagate($\mathcal{M}, T_0, T_1$)
    $\mathcal{M}$ is a (partial) assignment; $T_0, T_1$ are finite, monotonic theories with shared
  variables $x_0, x'_0, x_1, x'_1, \ldots$ of sort $\sigma$.
    changed ← TRUE
    while changed do
        changed ← FALSE
        for $T_i \in \{T_0, T_1\}$ do
            status, $\mathcal{M}'$ ← $T_i$-PROPAGATE($\mathcal{M}$)
            if status = FALSE then
                status is FALSE, $\mathcal{M}' \subseteq \mathcal{M}$ is a conflict set
                return FALSE, $\mathcal{M}'$
            else
                status is TRUE, $\mathcal{M}' \supseteq \mathcal{M}$ is a (possibly strengthened) assignment
                $\mathcal{M} \leftarrow \mathcal{M}'$
                for each shared variable $x_j$ of $T_i$ do
                    for each atom $(x_j \geq \sigma_k) \in \mathcal{M}$ do
                        if $(x'_j \geq \sigma_k) \notin \mathcal{M}$ then
                            changed ← TRUE
                            $\mathcal{M} \leftarrow \mathcal{M} \cup \{(x'_j \geq \sigma_k)\}$
                    for each atom $(x_j \leq \sigma_k) \in \mathcal{M}$ do
                        if $(x'_j \leq \sigma_k) \notin \mathcal{M}$ then
                            changed ← TRUE
                            $\mathcal{M} \leftarrow \mathcal{M} \cup \{(x'_j \leq \sigma_k)\}$
    return TRUE, $\mathcal{M}$
```

Chapter 5

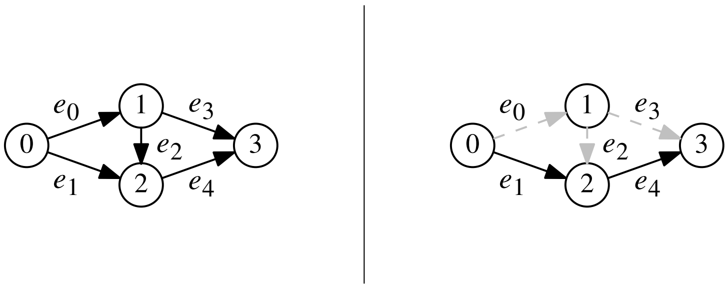
Monotonic Theory of Graphs

Chapter 4 described a set of techniques for building lazy SMT solvers for finite, monotonic theories. In this chapter, we introduce a set of monotonic graph predicates collected into a theory of graphs, as well as an implementation of an SMT solver for this theory, built using the SMMT framework.

Many well-known graph properties — such as reachability, shortest path, acyclicity, maximum *s-t* flow, and minimum spanning tree weight — are monotonic with respect to the edges or edge weights in a graph. We describe our support for predicates over these graph properties in detail in Section 5.2.

The corresponding graph theory solver for these predicates forms the largest part of our SMT solver, MONOSAT (described in Appendix A). In Section 5.1, we give an overview of our implementation, using the techniques described in Chapter 4. In Section 5.3, we extend support to bitvector weighted edges, by combining the theory of graphs with a theory of bitvectors.

In later chapters, we will describe how we have successfully applied this theory of graphs to problems ranging from procedural content generation (Chapter 6.1) to PCB layout (Chapter 6.2), in the latter case solving complex, real-world graph constraints with more than 1,000,000 nodes — a massive improvement in scalability over comparable SAT-based techniques.



$$reach_{0,3}(E) \wedge \neg reach_{1,3}(E) \wedge (\neg(e_0 \in E) \vee \neg(e_1 \in E))$$

Figure 5.1: *Left:* A finite symbolic graph over four nodes and set E of five potential edges, along with a formula constraining that graph. A typical instance solved by MONOSAT may have constraints over multiple graphs, each with hundreds of thousands of edges. *Right:* A satisfying assignment (disabled edges dashed, enabled edges solid), corresponding to $\{(e_0 \notin E), (e_1 \in E), (e_2 \notin E), (e_3 \notin E), (e_4 \in E)\}$.

5.1 A Monotonic Graph Solver

The theory of graphs that we introduce supports predicates over several common graph properties (each of which is monotonic with respect to the set of edges in the graph). Each graph predicate is defined for a directed graph with a finite set of vertices V and a finite set of *potential* edges $E \subseteq V \times V$, where a potential edge is an edge that may (or may not) be included in the graph, depending on the assignment chosen by the SAT solver.[1] For each potential edge of E, we introduce an atom $(e \in E)$, such that the edge e is enabled in E if and only if theory atom $(e \in E)$ is assigned TRUE. In order to distinguish the potential edges e from the edges with edge literals $(e \in E)$ that are assigned TRUE, we will refer to an element for which $(e \in E)$ is assigned TRUE as enabled in E (or enabled in the corresponding graph over edges E), and refer to an element for which $(e \in E)$ is assigned FALSE as disabled in E, using the notation $(e \notin E)$ as shorthand for $\neg(e \in E)$.

Returning to our earlier example of graph reachability (Figure 5.1), the pred-

[1] Many works applying SAT or SMT solvers to graphs have represented the edges in the graph in a similar manner, using a literal to represent the presence of each edge, and sometimes also a literal to control the presence of each node, in the graph. See, e.g., [90, 106, 133, 220].

icate $reach_{s,t}(E)$ is TRUE if and only if node t is reachable from node s in graph G, under a given assignment to the edge literals $(e_i \in E)$. As previously observed, given a graph (directed or undirected) and some fixed starting node s, enabling an edge in E can increase the set of nodes that are reachable from s, but cannot decrease the set of reachable nodes. The other graph predicates we consider are each similarly positive or negative monotonic with respect to the set of edges in the graph; for example, enabling an edge in a graph may decrease the weight of the minimum spanning tree of that graph, but cannot increase it.

Theory propagation as implemented in our theory of graphs is described in Algorithm 13, which closely follows Algorithm 3. Algorithm 13 represents $\mathcal{M}_A^+, \mathcal{M}_A^-$ in the form of two concrete graphs, G^- and G^+. The graph G^- is formed from the edge assignments in $\mathcal{M}_A$: only edges e for which the atom $(e \in E)$ is in $\mathcal{M}$ are included in G^-. In the second graph, G^+, we include all edges for which $(e \notin E)$ is not in $\mathcal{M}_A$.

Algorithm 13 makes some minor improvements over Algorithm 3. The first is that we re-use the data structures for G^-, G^+ across separate calls to *T-Propagate*, updating them by adding or removing edges as necessary. As the graphs can be very large (e.g., in Section 6.2 we will consider applications with hundreds of thousands or even millions of edges), and there are typically only a few edges either added or removed between calls to *T-Propagate*, this saves a large amount of redundant effort that would otherwise be caused by repeatedly creating the graphs.

A second improvement is to check whether, under the current partial assignment, either G^- or G^+ is unchanged from the previous call to *T-Propagate*. If the solver has only enabled edges in E (resp. only disabled edges in E) since the last theory propagation call, then the graph G^+ (resp. G^-) will not have changed, and so we do not need to recompute properties for that graph.

Each time an assignment is made to a graph theory atom, Algorithm 13 evaluates each predicate atom on both the under- and over-approximative graphs G^- and G^+. The practical performance of this scheme can be greatly improved by using partially or fully dynamic graph algorithms, which can be efficiently updated as edges are removed or added to G^- or G^+. Similarly, care should be taken to represent G^- and G^+ using data structures that can be cheaply updated as edges are removed and added. In our implementation, we use an adjacency list in which

Algorithm 13 Theory propagation for the theory of graphs, adapted from Algorithm 3. $\mathcal{M}$ is a (partial) assignment. E^-, E^+ are sets of edges; G^-, G^+ are under- and over-approximate graphs. *T-Propagate* returns a tuple (FALSE, conflict) if $\mathcal{M}$ is found to be unsatisfiable, and returns tuple (TRUE, $\mathcal{M}$) otherwise.

```
function T-Propagate($\mathcal{M}$)
    UpdateBounds:
    $E^- \leftarrow \{\}, E^+ \leftarrow \{E\}$
    for each finite symbolic graph $G = (V, E)$ do
        for each edge $e_i$ of $E$ do
            if $(e_i \notin E) \in \mathcal{M}$ then
                $E^+ \leftarrow E^+ \setminus \{e_i\}$
            if $(e_i \in E) \in \mathcal{M}$ then
                $E^- \leftarrow E^- \cup \{e_i\}$
        $G^- \leftarrow (V, E^-), G^+ \leftarrow (V, E^+)$
    PropagateBounds:
    for each predicate atom $p(E)$ do
        If p is negative monotonic, swap $G^-, G^+$ below.
        if $\neg p \in \mathcal{M}$ then
            if $evaluate(p, G^-)$ then
                return FALSE, $analyze(p, G^-)$
        else if $p \in \mathcal{M}$ then
            if not $evaluate(p, G^+) \mapsto$ FALSE then
                return FALSE, $analyze(\neg p, G^+)$
        else
            if $evaluate(p, G^-)$ then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{p\}$
            else if not $evaluate(p, G^+)$ then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{\neg p\}$
    return TRUE, $\mathcal{M}$
```

every possible edge of E has a unique integer ID. These IDs map to corresponding edge literals $(e \in E)$ and are useful for conflict analysis, in which we often must work backwards from graph analyses to identify the theory literals corresponding to a subset of relevant edges. Each adjacency list also maintains a history of added/removed edges, which facilitates the use of dynamic graph algorithms in the solver.

5.2 Monotonic Graph Predicates

Many of the most commonly used properties of graphs are monotonic with respect to the edges in the graph. A few well-known examples include:

1. Reachability
2. Acyclicity
3. Connected component count
4. Shortest *s*-*t* path
5. Maximum *s*-*t* flow
6. Minimum spanning tree weight

The first three properties are unweighted, while the remainder operate over weighted graphs and are monotonic with respect to both the set of edges and the weights of those edges. For example, the length of the shortest path from node *s* to node *t* in a graph may increase as the length of an edge is increased, but cannot decrease.

These are just a few examples of the many graph predicates that are monotonic with respect to the edges in the graph; there are many others that are also monotonic and which may be useful (a few examples: planarity testing/graph skew, graph diameter, minimum global cut, minimum-cost maximum flow, and Hamiltonian-circuit). Some examples of *non*-monotonic graph properties include Eulerian-circuit (TRUE iff the graph has an Eulerian-circuit) and Graph-Isomorphism (TRUE iff two graphs are isomorphic to each other).

For each predicate p in the theory solver we implement three functions:

1. $\mathit{evaluate}(p, G)$, which takes a concrete graph (it may be the under-, or the over-approximative graph) and returns TRUE iff p holds in the edges of G,
2. $\mathit{analyze}(p, G)$, which takes a concrete graph G in which p evaluates to TRUE and returns a justification set for p, and
3. $\mathit{analyze}(\neg p, G)$, which takes a concrete graph G in which p evaluates to FALSE and returns a justification set for $\neg p$.

Each of these functions is used by Algorithm 13 to implement theory propagation and conflict analysis. In Sections 5.2.1 and 5.2.2, we describe the implementation of our theory solvers for reachability and acyclicity predicates; subsequently, in Section 5.3.1, we describe our implementation of a theory solver for maximum *s-t* flow over weighted graphs. In Appendix D.1, we list our implementations of the remaining supported graph properties mentioned above.

5.2.1 Graph Reachability

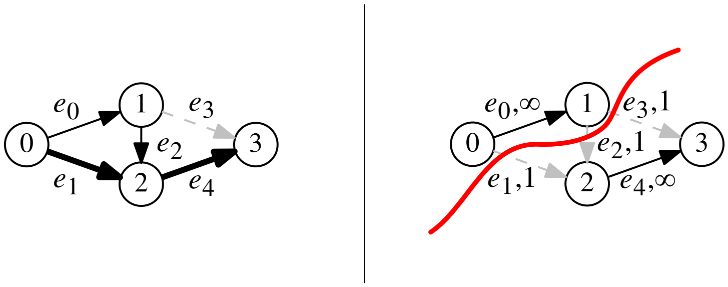


Figure 5.2: *Left:* A finite symbolic graph of edges E under assignment $\{(e_0 \in E),(e_1 \in E),(e_2 \in E),(e_3 \notin E),(e_4 \in E)\}$ in which $reach_{0,3}(E)$ holds. Edges e_1, e_4 (bold) form a shortest-path from node 0 to node 3. *Right:* The same graph under the assignment $\{(e_0 \in E),(e_1 \notin E),(e_2 \notin E),(e_3 \notin E),(e_4 \in E)\}$, in which $reach_{0,3}(E)$ does not hold. A cut (red) of disabled edges e_1, e_2, e_3 separates node 0 from node 3.

The first monotonic graph predicate we consider in detail is reachability in a finite graph (from a fixed starting node to a fixed ending node). A summary of our implementation can be found in Figure 5.3.

Both directed and undirected reachability constraints on explicit graphs arise in many contexts in SAT solvers.[2] A few applications of SAT solvers in which

[2] As with all the other graph predicates we consider here, this reachability predicate operates on graphs represented in explicit form, for example, as an adjacency list or matrix. This is in contrast to safety properties as considered in model checking [35, 47, 58], which can be described as testing the reachability of a node in a (vast) graph represented in an implicit form (i.e., as a BDD or SAT formula). Although SAT solvers are commonly applied to implicit graph reachability tasks such as model checking and planning, the techniques we describe in this section are not appropriate for model checking implicitly represented state graphs; nor are techniques such as SAT-based bounded [35] or unbounded [41] model checking appropriate for the explicit graph synthesis problems we consider.

Monotonic Predicate: $reach_{s,t}(E)$, true iff t can be reached from u in the graph formed of the edges enabled in E.

Implementation of $evaluate(reach_{s,t}(E), G)$**:** We use the dynamic graph reachability/shortest paths algorithm of Ramalingam and Reps [171] to test whether t can be reached from s in G. Ramalingam-Reps is a dynamic variant of Dijkstra's Algorithm [79]); our implementation follows the one described in [48]. If there are multiple predicate atoms $reach_{s,t}(E)$ sharing the same source s, then Ramalingam-Reps only needs to be updated once for the whole set of atoms.

Implementation of $analyze(reach_{s,t}(E), G^-)$ Node s reaches t in G^-, but $reach_{s,t}(E)$ is assigned FALSE in $\mathcal{M}$ (Figure 5.2, left). Let $e_0, e_1, \ldots$ be an $s - t$ path in G^-; return the conflict set $\{(e_0 \in E), (e_1 \in E), \ldots, \neg reach_{s,t}(E)\}$.

Implementation of $analyze(\neg reach_{s,t}(E), G^+)$ Node t cannot be reached from s in G^+, but $reach_{s,t}(E)$ is assigned TRUE. Let $e_0, e_1, \ldots$ be a cut of disabled edges ($e_i \notin E$) separating u from v in G^+. Return the conflict set $\{(e_0 \notin E), (e_1 \notin E), \ldots, reach_{s,t}(E)\}$.

We find a minimum separating cut by creating a graph containing all edges of E (including both the edges of G^+ and the edges that are disabled in G^+ in the current assignment), in which the capacity of each disabled edge of E is 1, and the capacity of all other edges is infinity (forcing the minimum cut to include only edges that correspond to disabled edge atoms). Any standard maximum s-t flow algorithm can then be used to find a minimum cut separating s from t (see Figure 5.2, right, for an example of such a cut). In our implementation, we use the dynamic Kohli-Torr [140] minimum cut algorithm for this purpose.

Decision Heuristic: (Optional) If $reach_{s,t}(E)$ is assigned TRUE in $\mathcal{M}$, but there does not yet exist a $s-t$ path in G^-, then find a $s-t$ path in G^+ and pick the first unassigned edge in that path to be assigned true as the next decision. In practice, such a path has typically already been discovered, during the evaluation of $reach_{s,t}$ on G^+ during theory propagation.

Figure 5.3: Summary of our theory solver implementation for $reach_{s,t}$.

reachability constraints play an integral role include many variations of routing problems (e.g., FPGA and PCB routing [153, 154, 177, 203] and optical switch routing [5, 102]), Hamiltonian cycle constraints [133, 220], and product line configuration [6, 149, 150]. The SAT-based solver for the Alloy relational modeling language[130] encodes transitive closure into CNF as a reachability constraint over an explicit graph [129, 131].

There are several common ways to encode reachability constraints into SAT and SMT solvers described in the literature. Some works encode reachability directly into CNF by what amounts to unrolling the Floyd-Warshall algorithm symbolically (e.g., [177]); unfortunately, this approach requires $\mathcal{O}(|E| \cdot |V|^2)$ clauses and hence scales very poorly.

SMT solvers have long had efficient, dedicated support for computing congruence closure over equality relations[19, 157, 159, 160], which amounts to fast all-pairs undirected reachability detection.[3] Undirected reachability constraints can be encoded into equality constraints (by introducing a fresh variable for each vertex, and an equality constraint between neighbouring vertices that is enforced only if an arc between neighbouring vertices is enabled). The resulting encoding is efficient and requires only $\mathcal{O}(|V| + |E|)$ constraints, however, the encoding is only one-sided: it can enforce that two nodes must not be connected, but it cannot enforce that they must be connected. Further, this equality constraint encoding cannot enforce directed reachability constraints at all.

While equality constraints cannot encode directed reachability, directed reachability constraints can still be encoded into SMT solvers using arithmetic theories. An example of such an approach is described in [90]. This approach introduces a non-deterministic distance variable for each node other than source (which is set to a distance of zero), and then adds constraints for every node other than the source node, enforcing that each node's distance is one greater than its nearest-to-source neighbor, or is set to some suitably large constant if none of its neighbours are reachable from source. This approach requires $\mathcal{O}(|E| \cdot |V| \cdot \log |V|)$ constraints if distances are encoded in bit-blasted bitvectors, or $\mathcal{O}(|E| \cdot |V|)$ constraints if dis-

We will consider applications related to model checking in Chapter 8.

[3] Note: Some writers distinguish undirected reachability from directed reachability using the term *connectivity* for the undirected variant.

tances are instead encoded in linear or integer arithmetic. However, while this produces a concise encoding, it forces the solver to non-deterministically guess the distance to each node, which in practice seems to work very poorly (see Figure 5.4). [4]

In addition to SAT and SMT solvers, many works have recently used answer set programming (ASP) [25] constraint solvers to enforce reachability constraints. Modern ASP solvers are closely related to CDCL solvers, however, unlike SAT, ASP can encode reachability constraints on arbitrary directed graphs in linear space, and ASP solvers can solve the resulting formulas efficiently in practice. The ASP solver CLASP [104], which is implemented as an extended CDCL solver, has been particularly widely used to enforce reachability constraints for procedural content generation applications (e.g. [125, 182, 183, 219] all apply CLASP to procedural content generation using reachability constraints).

Considered on its own, the directed reachability predicate implemented in MONOSAT's theory of graphs scales dramatically better than any of the above approaches for directed reachability, as can be seen in Figure 5.4.[5] In Section 6.1, we will see that this translates into real performance improvements in cases where the constraints are dominated by a small number of reachability predicates. However, in cases where a large number of reachability predicates are used (and in particular, in cases where those reachability predicates are mutually unsatisfiable or nearly unsatisfiable), this approach to reachability predicates scales poorly.

[4] If only one-sided directed reachability constraints are required (i.e., the formula is satisfiable only if node *s* reaches node *t*), then more efficient encodings are possible, by having the SAT solver non-deterministically guess a path. Some examples of one-sided reachability encodings are described in [106].

[5] All experiments in this chapter were run on a 2.67GHz Intel x5650 CPU (12Mb L3, 96 Gb RAM), on Ubuntu 12.04, restricted to 10,000 seconds of CPU time, and 16 GB of RAM.

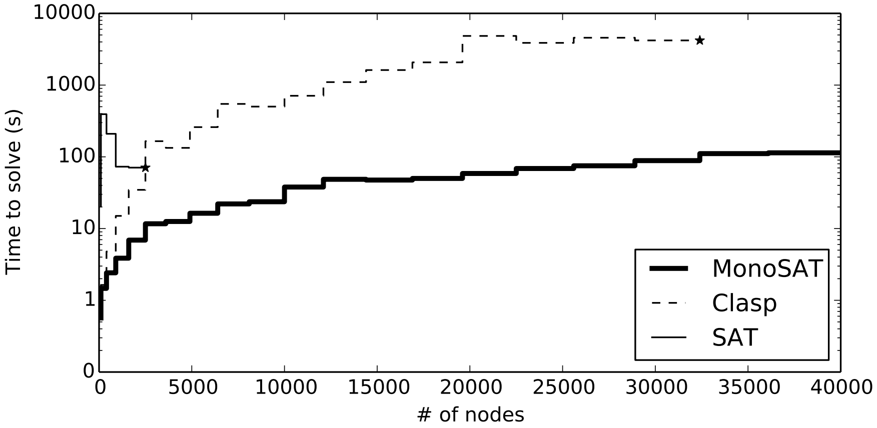


Figure 5.4: Run-times of MONOSAT, SAT, and ASP solvers on randomly generated, artificial reachability constraint problems of increasing size. The graphs are planar and directed, with 10% of the edges randomly asserted to be pair-wise mutually exclusive. In each graph, it is asserted that exactly one of (the bottom right node is reachable from the top left), or (the top right node is reachable from the bottom left) holds, using a pair of two-sided reachability constraints. The SAT solver results report the best runtimes found by Lingeling (version 'bbc') [34], Glucose-4 [14], and MiniSat 2.2 [84]. The SAT solvers have erratic runtimes, but run out of memory on moderately sized instances. CLASP eventually runs out of memory as well (shown with asterisk), but on much larger instances than the SAT solvers. We also tested the SMT solver Z3 [70] (version 4.3.2, using integer and rational linear arithmetic and bitvector encodings), however, we exclude it from the graph as Z3 timed out on all but the smallest instances.

Monotonic Predicate: $acyclic(E)$, true iff there are no (directed) cycles in the graph formed by the enabled edges in E.

Implementation of $evaluate(acyclic(E),G)$**:** Apply the PK dynamic topological sort algorithm (as described in [164]). The PK algorithm is a fully dynamic graph algorithm that maintains a topological sort of a directed graph as edges are added to or removed from the graph; as a side effect, it also detects directed cycles (in which case no topological sort exists). Return FALSE if the PK algorithm successfully produces a topological sort, and return TRUE if it fails (indicating the presence of a directed cycle).

Implementation of $\neg analyze(acyclic(E),G^-)$ There is a cycle in E, but $acyclic(E)$ is assigned TRUE. Let $e_0, e_1, \ldots$ be the edges that make up a directed cycle in E; return the conflict set $\{(e_0 \in E),(e_1 \in E),\ldots,acyclic(E)\}$.

Implementation of $analyze(acyclic(E),G^+)$ There is no cycle in E, but $acyclic(E)$ is assigned FALSE. Let $e_0, e_1, \ldots$ be the set of all edges not in G^+; return the conflict set $\{(e_0 \notin E),(e_1 \notin E),\ldots,\neg acyclic(E)\}$. (Note that this is the default monotonic conflict set.)

Figure 5.5: Summary of our theory solver implementation for *acyclic*.

5.2.2 Acyclicity

A second monotonic predicate we consider is acyclicity. Predicate $acyclic(E)$ is TRUE iff the (directed) graph over the edges enabled in E contains no cycles. The acyclicity of a graph is negative monotonic in the edges of E, as enabling an edge in E may introduce a cycle, but cannot remove a cycle. We describe our theory solver implementation in Figure 5.5. In addition to the directed acyclicity predicate above, there is an undirected variation of the predicate (in which the edges of E are interpreted as undirected) — however, for the undirected version, the PK topological sort algorithm cannot be applied, and so we fall back on detecting cycles using depth-first search. While depth-first search is fast for a single run, it is substantially slower than the PK topological sort algorithm for repeated calls as edges are removed or added to the graph.

Several works have explored pure CNF encodings of acyclicity constraints (e.g., [175] applied acyclicity constraints as part of encoding planning problems into SAT, and [65] applied acyclicity in their encoding of Bayesian networks into SAT). Typical CNF encodings require $\mathcal{O}(|E| \cdot |V|)$ or $\mathcal{O}(|E| \cdot \log |V|)$ clauses [106]. Acyclicity constraints can also be encoded into several existing SMT logics: the theories of linear arithmetic, integer arithmetic, and difference logic, as well as the more restrictive theory of ordering constraints ([103]) can all express acyclicity in a linear number of constraints. Recent work [105, 106] has shown that specialized SMT theories directly supporting (one-sided) acyclicity predicates can outperform SAT and arithmetic SMT encodings of acyclicity.

In Figure 5.6, we compare the performance of MONOSAT's acyclicity constraint to both SAT and SMT encodings. We can see that our approach greatly out-performs standard SAT encodings, but is only slightly faster than the dedicated, one-sided acyclicity SMT solvers ACYCGLUCOSE/ACYCMINISAT (described in [106]). In fact, for the special case of acyclicity constraints that are asserted to TRUE at the ground level, the implementation of the ACYCMINISAT solver, with incremental mode enabled and edge propagation disabled, essentially matches our own (with the exception that we detect cycles using a topological sort, rather than depth-first search).

Many variations of the implementations of the above two predicates could also be considered. For example, the Ramalingam-Reps algorithm we use for reachability is appropriate for queries in sparsely connected directed and undirected graphs. If the graph is densely connected, then other dynamic reachability algorithms (such as [116]) may be more appropriate. If there are many reach predicates that do not share a common source in the formula, then an all-pairs dynamic graph algorithm (such as [75]) might be more appropriate. In many cases, there are also specialized variations of dynamic graph algorithms that apply if the underlying graph is known to have a special form (for example, specialized dynamic reachability algorithms have been developed for planar graphs [117, 194].)

The reachability predicate theory solver implementation that we describe here is particularly appropriate for instances in which most reachability predicates share a small number of common sources or destinations (in which case multiple predicate atoms can be tested by a single call to the Ramalingam-Reps algorithm).

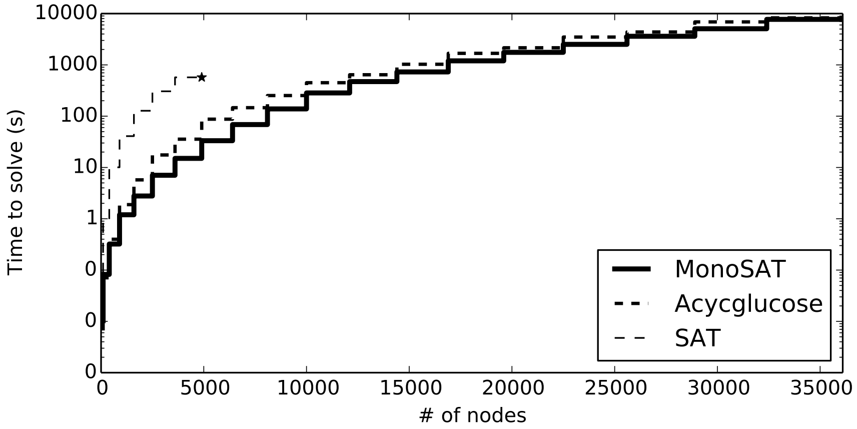


Figure 5.6: Run-times of SAT and SMT solvers on randomly generated, artificial acyclicity constraint problems of increasing size. We consider grids of edges, constrained to be partitioned into two graphs such that both graphs are disjoint and acyclic. The graphs are planar and directed, with 0.1% of the edges randomly asserted to be pair-wise mutually exclusive. The plain SAT entry represents the best runtimes obtained by the solvers Lingeling (version 'bbc') [34], Glucose-4 [14], or MiniSat 2.2 [84]. The SAT solvers run out of memory at 5000 edges, so we do not report results for SAT for larger instances. The ACYCGLUCOSE entry represents the best runtimes for ACYCGLUCOSE and ACYCMINISAT, with and without pre-processing, incremental mode, and edge propagation. While both MONOSAT and ACYCGLUCOSE greatly outperform a plain CNF encoding, the runtimes of MONOSAT and ACYCGLUCOSE are comparable (with MONOSAT having a slight edge).

Two recent works [90, 152] introduced SMT solvers designed for VLSI and clock routing (both of which entail solving formulas with many reachability or shortest path predicates with distinct sources and destinations). In [152] the authors provide an experimental comparison against our graph theory solver (as implemented in MONOSAT), demonstrating that their approach scales substantially better than ours for VLSI and clock routing problems. In the future, it may be possible to integrate the sophisticated decision heuristics described in those works into MONOSAT. In Section 6.2, we also describe a third type of routing problem, *escape routing*, for which we obtain state-of-the-art performance using the maximum flow predicate support that we discuss in the next section.

5.3 Weighted Graphs & Bitvectors

In Section 5.2, we considered two predicates that take a single argument (the set of edges, E), operating on an unweighted, directed graph. These two predicates, reachability and acyclicity, have many applications ranging from procedural content generation to circuit layout (for example, in a typical circuit layout application, one must connect several positions in a graph, while ensuring that the wires are acyclic and non-crossing).

However, many common graph properties, such as maximum *s*-*t* flow, shortest *s*-*t* path, or minimum spanning tree weight, are more naturally posed as comparison predicates operating over graphs with variable edge weights (or edge capacities).[6] As described in Section 4.5, a monotonic theory over a finite sort σ can be combined with another theory over σ, so long as that other theory supports comparison operators, with the two theory solvers communicating only through exchanges of atoms comparing variables to constants: $(x \leq c), (x < c)$. Extending Algorithm 13 to support bitvector arguments in this way requires only minor changes to theory propagation in the graph solver (shown in Algorithm 14); it also requires us to introduce a bitvector theory solver capable of efficiently deriving tight comparison atoms over its bitvector arguments. We describe our bitvector theory solver in Appendix B.

5.3.1 Maximum Flow

The maximum *s*-*t* flow in a weighted, directed graph, in which each edge of E has an associated capacity c, is positive monotonic with respect to both the set of edges enabled in E, and also with respect to the capacity of each edge. We introduce a predicate $maxFlow_{s,t}E, m, c_0, c_1, \ldots$, with E a set of edges, and $m, c_0, c_1, \ldots$ fixed width bit-vectors, which evaluates to TRUE iff the maximum *s*-*t* flow in the directed graph induced by edges E, with edge capacities $c_0, c_1, \ldots$, is greater or equal to m.[7]

[6] It is also possible to consider these predicates over constant edge-weight graphs. We described such an approach in [33]; one can recover that formulation from our presentation here by simply setting each bitvector variable to a constant value.

[7] There is also a variation of this predicate supporting strictly greater than comparisons, but we describe only the 'greater or equal to' variant. It is also possible to describe this predicate instead as a function that returns the maximum flow, constrained by a bitvector comparison constraint. For technical reasons this function-oriented presentation is less convenient in our implementation.

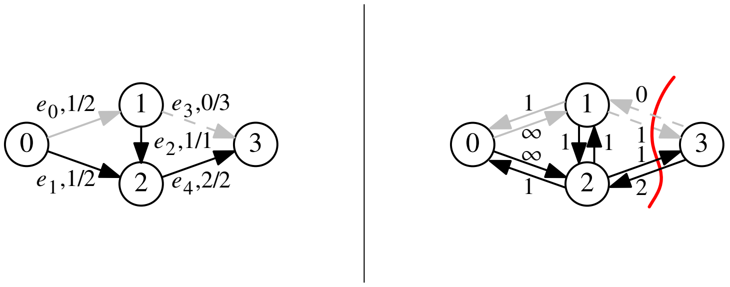


Figure 5.7: *Left:* Over-approximate graph G^+ for the partial assignment $\{(e_1 \in E), (e_2 \in E), (e_3 \notin E), (e_4 \in E)\}$, with $(e_0 \in E)$ unassigned. The maximum 0-3 flow in this graph is 2; each edge has its assigned flow and capacity shown as f/c. *Right:* The cut graph G^{cut} for this assignment, with corresponding minimum 0-3 cut (red): $\{e_3, e_4\}$. Either edge e_3 must be included in the graph, or the capacity of edge e_4 must be increased, in order for the maximum flow to be greater than 2.

This predicate is positive monotonic with respect to the edge literals and their capacity bitvectors: Enabling an edge in E can increase but cannot decrease the maximum flow; increasing the capacity of an edge can increase but cannot decrease the maximum flow. It is negative monotonic with respect to the flow comparison bitvector m. We summarize our theory solver implementation in Figure 5.8.

Maximum flow constraints appear in some of the same routing-related applications of SAT solvers as reachability constraints, in particular when bandwidth must be reasoned about; some examples from the literature include FPGA layout [4], virtual data center allocation [212] and routing optical switching networks [5, 102].

However, whereas many applications have successfully applied SAT solvers to reachability constraints, encoding maximum flow constraints into SAT or SMT solvers scales poorly (as we will show), and has had limited success in practice. In contrast, constraints involving maximum flows can be efficiently encoded into integer-linear programming (ILP) and solved using high-performance solvers, such as CPLEX [64] or Gurobi [162]. Many examples of ILP formulas including flow constraints (in combination with other constraints) can be found in the literature; examples include PCB layout [93, 123], routing optical switching networks [205, 214], virtual network allocation [38, 126], and even air-traffic routing [198].

Monotonic Predicate: $\mathit{maxFlow}_{s,t}(E, m, c_0, c_1, \ldots)$, true iff the maximum *s*-*t* flow in G with edge capacities $c_0, c_1, \ldots$ is $\geq m$.

Implementation of $\mathit{evaluate}(\mathit{maxFlow}_{s,t}, G, m, c_0, c_1, \ldots)$**:** We apply the dynamic minimum-cut/maximum *s*-*t* algorithm by Kohli and Torr [140] to compute the maximum flow of G, with edge capacities set by c_i. Return TRUE iff that flow is greater or equal to m.

Implementation of $\mathit{analyze}(\mathit{maxFlow}_{s,t}, G^-, m^+, c_0^-, c_1^-, \ldots)$**:** The maximum *s*-*t* flow in G^- is f, with $f \geq m^+$). In the computed flow, each edge e_i is either disabled in G^-, or it has been allocated a (possibly zero-valued) flow f_i, with $f_i \leq c_i^-$. Let $e_a, e_b, \ldots$ be the edges enabled in G^- with non-zero allocated flows $f_a, f_b, \ldots$. Either one of those edges must be disabled in the graph, or one of the capacities of those edges must be decreased, or the flow will be at least f. Return the conflict set $\{(e_a \in E), (e_b \in E), \ldots, (c_a \geq f_a), (c_b \geq f_b) \ldots, (m \leq f), \mathit{maxFlow}_{s,t}(E, m, c_0, c_1, \ldots)\}$.

Implementation of $\mathit{analyze}(\neg \mathit{maxFlow}_{s,t}, G^+, m^-, c_0^+, c_1^+, \ldots)$ The maximum *s*-*t* flow in G^+ is f, with $f < m^-$ (see Figure 5.7 left). In the computed flow, each edge that is enabled in G^+ has been allocated a (possibly zero-valued) flow f_i, with $f_i \leq c_i^+$.

If f is a maximum flow in G^+, then there must exist a cut of edges in G^+ whose flow assignments equal their capacity. Our approach to conflict analysis is to discover such a cut, by constructing an appropriate graph G^{cut}, as described below.

Create a graph G^{cut} (see Figure 5.7 right, for an example of such a graph). For each edge $e_i = (u, v)$ in G^+, with $f_i < c_i^+$, add a forward edge (u, v) to G^{cut} with infinite capacity, and also a backward edge (v, u) with capacity f_i. For each edge $e_i = (u, v)$ in G^+ with $f_i = c_i^+$, add a forward edge (u, v) to G^{cut} with capacity 1, and also a backward edge (v, u) with capacity f_i. For each edge $e_i = (u, v)$ that is disabled in G^+, add only the forward edge (u,v) to G^{cut}, with capacity 1.

Compute the minimum *s*-*t* cut of G^{cut}. Some of the edges along this cut may have been edges disabled in G^{+}, while some may have been edges enabled in G^{+} with fully utilized edge capacity. Let $e_a, e_b, \ldots$ be the edges of the minimum cut of G^{cut} that were disabled in G^{+}. Let $c_c, c_d, \ldots$ be the capacities of edges in the minimum cut for which the edge was included in G^{+}, with fully utilized capacity. Return the conflict set $\{(e_a \notin E), (e_b \notin E), \ldots, (c_c \leq f_c), (c_d \leq f_d), \ldots, (m > f), \neg maxFlow_{s,t}(E, m, c_0, c_1, \ldots)\}$.

In practice, we maintain a graph G^{cut} for each maximum flow predicate atom, updating its edges only lazily when needed for conflict analysis.

Decision Heuristic: (Optional) If $maxFlow_{s,t}$ is assigned TRUE in $\mathcal{M}$, but there does not yet exist a sufficient flow in G^{-}, then find a maximum flow in G^{+}, and pick the first unassigned edge with non-zero flow to be assigned TRUE as the next decision. If no such edge exists, then pick the first unassigned edge capacity and assign its capacity to its flow G^{+}, as the next decision. In practice, such a flow is typically already discovered, during the evaluation of $maxFlow_{s,t}$ on G^{+} during theory propagation.

Figure 5.8: Summary of our theory solver implementation for $maxFlow_{s,t}$.

Maximum flow constraints on a graph $G = (V,E)$ in which each edge $(u,v) \in E$ has an associated capacity $c(u,v)$ can be encoded into arithmetic SMT theories in two parts. The first part of the encoding introduces, for each edge (u,v), a fresh bitvector, integer, or real flow variable $f(u,v)$, constrained by the standard network flow equations [62]:

$$
\begin{aligned}
\forall u, v \in V : f(u,v) \leq c(u,v) \\
\forall u \in V/\{s,t\}, \sum_{v \in V} f(u,v) = \sum_{w \in V} f(v,w) \\
\sum_{v \in V} f(s,v) - \sum_{w \in V} f(w,s) = \sum_{w \in V} f(w,t) - \sum_{v \in V} f(t,v)
\end{aligned}
$$

The second part of the encoding non-deterministically selects an *s*-*t* cut by introducing a fresh Boolean variable $a(v)$ for each node $v \in V$, with $a(v)$ TRUE iff v is on the source side of the cut. The sum of the capacities of the edges passing through that cut are then asserted to be equal to the flow in the graph:

$$a(s) \wedge \neg a(t) \tag{5.1}$$

$$\sum_{v \in V} f(s,v) - \sum_{w \in V} f(w,s) = \sum_{(u,v) \in E, a(u) \wedge \neg a(v)} c(u,v) \tag{5.2}$$

By the max-flow min-cut theorem [86], an *s*-*t* flow in a graph is equal to an *s*-*t* cut if and only if that flow is a maximum flow (and the cut a minimum cut). This encoding is concise (requiring $\mathcal{O}(|E| + |V|)$ arithmetic SMT constraints, or $\mathcal{O}(|E| \cdot \log|V| + |V| \cdot \log|V|)$ Boolean variables if a bitvector encoding is used); unfortunately, the encoding depends on the solver non-deterministically guessing both a valid flow and a cut in the graph, and in practice scales very poorly (see Figure 5.9).

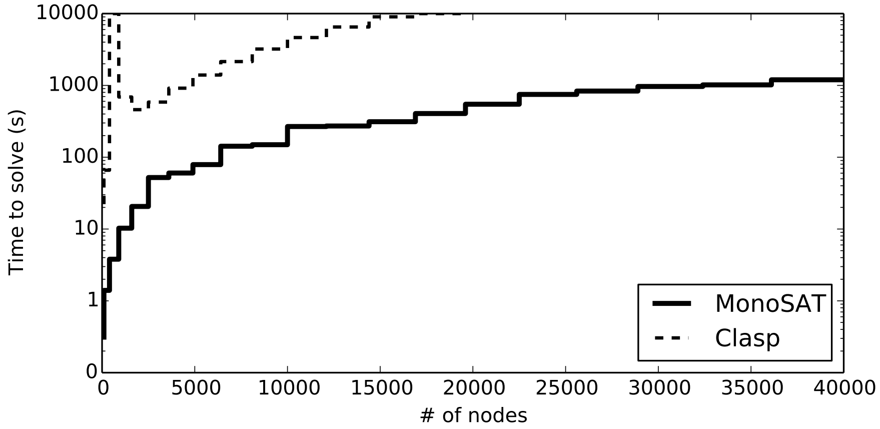


Figure 5.9: Run-times of MONOSAT and CLASP on maximum flow constraints. The SMT solver Z3 (using bitvector, integer arithmetic, and linear arithmetic encodings) times out on all but the smallest instance, so we omit it from this figure. We can also see that CLASP is an order of magnitude or more slower than MONOSAT.[8]In these constraints, randomly chosen edge-capacities (between 5 and 10 units) must be partitioned between two identical planar, grid-graphs, such that the maximum *s*-*t* flow in both graphs (from the top left to the bottom right nodes) is exactly 5. This is an (artificial) example of a multi-commodity flow constraint, discussed in more detail in Section 6.3.2.

In Figure 5.9 we compare the performance of MONOSAT's maximum flow predicate to the performance of Z3 (reporting the best results from linear arithmetic, integer arithmetic, and bitvector encodings as described above), and to the performance of CLASP, using a similar encoding into ASP [181].

In this case, Z3 is only able to solve the smallest instance we consider within a 10,000 second cutoff. We can also see that the encoding into ASP performs substantially better than Z3, while also being orders of magnitude slower than the encoding in MONOSAT.

[8] CLASP is unexpectedly unable to solve some of the smallest instances (top left of Figure 5.9). While I can speculate as to why this is, I do not have a definitive answer.

5.4 Conclusion

In this chapter we described our implementations of three important graph predicates in MONOSAT: reachability, acyclicity, and maximum *s*-*t* flow. Each of these predicates is implemented following the techniques described in Chapter 4, and for each we have demonstrated in this chapter state-of-the-art performance on large, artificially generated constraints. In Chapter 6, we will describe a series of applications for these graph predicates, demonstrating that the SMMT framework, as embodied in MONOSAT, can achieve great improvements in scalability over comparable constraint solvers in realistic, rather than artificial, scenarios. MONOSAT also provides high performance support for several further important graph predicates, including shortest path, connected component count, and minimum spanning tree weight. The implementations of the theory solvers for these predicates are similar to implementations discussed in this chapter, and are summarized in Appendix D.1.

Algorithm 14 Theory propagation, as implemented for the theory of graphs in combination with the theory of bitvectors. $\mathcal{M}$ is a (partial) assignment. E^-, E^+ are sets of edges; G^-, G^+ are under- and over-approximate graphs. *T-Propagate* returns a tuple (FALSE, conflict) if $\mathcal{M}$ is found to be unsatisfiable, and returns tuple (TRUE, $\mathcal{M}$) otherwise.

```
function T-Propagate($\mathcal{M}$)
    $E^- \leftarrow \{\}, E^+ \leftarrow \{E\}$
    for each finite symbolic graph $G = (V,E)$ do
        for each edge $e_i$ of $E$ do
            if $(e_i \notin E) \in \mathcal{M}$ then
                $E^+ \leftarrow E^+ \setminus \{e_i\}$
            if $(e_i \in E) \in \mathcal{M}$ then
                $E^- \leftarrow E^- \cup \{e_i\}$
        $G^- \leftarrow (V,E^-), G^+ \leftarrow (V,E^+)$
    for each bitvector variable $x$ of width $n$ do
        if $(x < 0) \in \mathcal{M}$ or $(x > 2^n - 1) \in \mathcal{M}$ then return FALSE
        $\mathcal{M} \leftarrow \mathcal{M} \cup \{(x \geq 0), (x \leq 2^n - 1)\}$
        $x^- \leftarrow \max(\{\sigma_i | (x \geq \sigma_i) \in \mathcal{M}\})$.
        $x^+ \leftarrow \min(\{\sigma_i | (x \leq \sigma_i) \in \mathcal{M}\})$.
        if $x^- > x^+$ then return FALSE
    for each predicate atom $p(E, x_0, x_1, \ldots)$ do
        E is a set of edge literals, and $x_i$ are bitvectors. If p is negative monotonic
  in argument E, swap $G^-, G^+$ below; if p is negative monotonic in argument $x_i$,
  swap $x_i^-, x_i^+$.
        if $\neg p \in \mathcal{M}$ then
            if $evaluate(p, G^-, x_0^-, x_1^-, \ldots)$ then
                return FALSE, $analyze(p, G^-, x_0^-, x_1^-, \ldots)$
        else if $p \in \mathcal{M}$ then
            if not $evaluate(p, G^+, x_0^+, x_1^+, \ldots) \mapsto$ FALSE then
                return FALSE, $analyze(\neg p, G^+, x_0^+, x_1^+, \ldots)$
        else
            if $evaluate(p, G^-, x_0-, x_1-, \ldots)$ then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{p\}$
            else if not $evaluate(p, G^+, x_0^+, x_1^+, \ldots)$ then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{\neg p\}$
    return TRUE, $\mathcal{M}$
```

Chapter 6

Graph Theory Applications

Over the last several chapters, we have claimed that theory solvers implemented using the techniques we have described in Chapter 4 can have good performance in practice. Here we describe applications of our theory of graphs (as implemented in our SMT solver MONOSAT, described in Appendix A) to three different fields, along with experimental evidence demonstrating that, for the theory of graphs described above, we have achieved — and in many cases greatly surpassed — state-of-the-art performance in each of these different domains.

The first application we consider, procedural content generation, presents results testing each of the graph predicates described in Chapter 5, demonstrating MONOSAT's performance across a diverse set of content generation tasks. The second and third applications we consider rely on the maximum flow predicate, and demonstrate MONOSAT's effectiveness on two industrial applications: circuit layout and data center allocation. These latter two application scenarios will show that MONOSAT can effectively solve real-world instances over graphs with hundreds of thousands of nodes and edges — in some cases, even instances with more than 1 million nodes and edges.

6.1 Procedural Content Generation

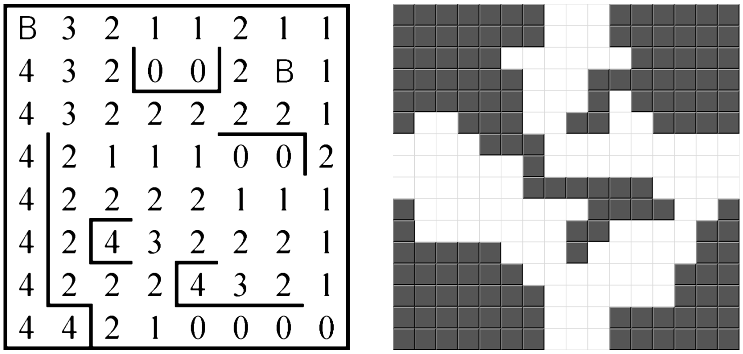
The first application we consider is procedural content generation, in which artistic objects, such as landscapes or mazes are designed algorithmically, rather than by hand. Many popular video games include procedurally generated content, leading to a recent interest in content generation using declarative specifications (see, e.g., [37, 183]), in which the artifact to be generated is specified as the solution to a logic formula.

Many procedural content generation tasks are really graph generation tasks. In maze generation, the goal is typically to select a set of edges to include in a graph (from some set of possible edges that may or may not form a complete graph) such that there exists a path from the start to the finish, while also ensuring that when the graph is laid out in a grid, the path is non-obvious. Similarly, in terrain generation, the goal may be to create a landscape which combines some maze-like properties with other geographic or aesthetic constraints.

For example, the open-source terrain generation tool Diorama[1] considers a set of undirected, planar edges arranged in a grid. Each position on the grid is associated with a height; Diorama searches for a height map that realizes a complex combination of desirable characteristics of this terrain, such as the positions of mountains, water, cliffs, and players' bases, while also ensuring that all positions in the map are reachable. Diorama expresses its constraints using answer set programming (ASP) [25]. As we described in Section 5.2.1, ASP solvers are closely related to SAT solvers, but unlike SAT solvers can encode reachability constraints in linear space. Partly for this reason, ASP solvers are more commonly used than SAT solvers in declarative procedural content generation applications. For instance, Diorama, Refraction [183], and Variations Forever [182] all use ASP.

Below, we provide comparisons of our SMT solver MONOSAT against the state-of-the-art ASP solver CLASP 3.04 [104] (and, where practical, also to MINISAT 2.2 [84]) on several procedural content generation problems. These experiments demonstrate the effectiveness of our reachability, shortest paths, connected components, maximum flow, and minimum spanning tree predicates. All experiments were conducted on Ubuntu 14.04, on an Intel i7-2600K CPU, at 3.4 GHz (8MB L3 cache), limited to 900 seconds and 16 GB of RAM. Reported runtimes for CLASP do not include the cost of grounding (which varies between instantaneous and hundreds of seconds, but in procedural content generation applications

[1] http://warzone2100.org.uk



B	3	2	1	1	2	1	1
4	3	2	0	0	2	B	1
4	3	2	2	2	2	2	1
4	2	1	1	1	0	0	2
4	2	2	2	2	1	1	1
4	2	4	3	2	2	2	1
4	2	2	2	4	3	2	1
4	4	2	1	0	0	0	0

Figure 6.1: Generated terrain. Left, a height map generated by Diorama, and right, a cave (seen from the side, with gravity pointing to the bottom of the image) generated in the style of 2D platformers. Numbers in the height map correspond to elevations (bases are marked as 'B'), with a difference greater than one between adjacent blocks creating an impassable cliff. Right, an example Platformer room, in which players must traverse the room by walking and jumping — complex movement dynamics that are modeled as directed edges in a graph.

is typically a sunk cost that can be amortized over many runs of the solver).

Reachability: We consider two applications for the theory of graph reachability. The first is a subset of the *cliff-layout* constraints from the terrain generator Diorama.[2]

The second example is derived from a 2D side-scrolling video game. This game generates rooms in the style of traditional *Metroidvania* platformers. Reachability constraints are used in two ways: first, to ensure that the air and ground blocks in the map are contiguous, and secondly, to ensure that the player's on-screen character is capable of reaching each exit from any reachable position on the map. This ensures not only that there are no unreachable exits, but also that there are no traps (i.e., reachable locations that the player cannot escape from, such as a steep pit) in the room. In this instance, although there are many reachabil-

[2] Because we had to manually translate these constraints from ASP into our SMT format, we use only a subset of these cliff-layout constraints. Specifically, we support the **undulate**, **sunkenBase**, **geographicFeatures**, and **everythingReachable** options from cliff-layout, with near=1, depth=5, and 2 bases (except where otherwise noted).

Reachability	MONOSAT	CLASP	MINISAT
Platformer 16×16	0.8s	1.5s	Timeout
Platformer 24×24	277s	Timeout	n/a
Diorama 16×16	6s	< 0.1s	Timeout
Diorama 32×32	58.9s	0.2s	n/a
Diorama 48×48	602.6s	7.9s	n/a

Table 6.1: Runtime results on reachability constraints in terrain generation tasks using MONOSAT and CLASP. We can see that for the Platformer instances, which are dominated by reachability constraints from just four source nodes, MONOSAT is greatly more scalable than CLASP; however, for the Diorama constraints, which contain many reachability predicates that do not share common source or destination nodes, CLASP greatly outperforms MONOSAT.

ity predicates, there are only four distinct source nodes among them (one source node to ensure the player can reach the exit; one source node to ensure the player cannot get trapped, and one source node each to enforce the 'air' and 'ground' connectedness constraints). As a result, the reachability predicates in this instance collapse down to just four distinct reachability theory solver instances in MONOSAT, and so can be handled very efficiently. In contrast, the Diorama constraints contain a large number of reachability predicates with distinct source (and destination) nodes, and so MONOSAT must employ a large number of distinct theory solvers to enforce them. Example solutions to small instances of the Diorama and platformer constraints are shown in Figure 6.1.

Runtime results in Table 6.1 show that both MONOSAT and CLASP can solve much larger instances than MINISAT (for which the larger instances are not even practical to encode, indicated as 'n/a' in the table). The comparison between MONOSAT and CLASP is mixed: On the one hand, CLASP is much faster than MONOSAT on the undirected Diorama instances. On the other hand, MONOSAT outperforms CLASP on the directed Platformer constraints.

Given that ASP supports linear time encodings for reachability constraints and is widely used for that purpose, CLASP's strong performance on reachability constraints is not surprising. Below, we combine the Diorama constraints with additional graph constraints for which the encoding into ASP (as well as CNF) is

Size	Range	MONOSAT	CLASP	MINISAT
8×8	8-16	<0.1s	<0.1s	9s
16×16	16-32	4s	7s	>3600s
16×16	32-48	4s	23s	2096s
16×16	32-64	4s	65s	>3600s
16×16	32-96	4s	>3600s	>3600s
16×16	32-128	4s	>3600s	>3600s
24×24	48-64	46s	30s	>3600s
24×24	48-96	61s	1125s	>3600s
32×32	64-128	196s	>3600s	>3600s

Table 6.2: Runtime results for shortest paths constraints in Diorama. Here, we can see that both as the size of the map is increased, and as the lengths of the shortest path constraints are increased, MONOSAT outperforms CLASP.

non-linear, and in each case MONOSAT outperforms CLASP, as well as MINISAT dramatically.

Shortest Paths: We consider a modified version of the Diorama terrain generator, replacing the reachability constraint with the constraint that the shortest path between the two bases must fall within a certain range ('Range', in Figure 6.2). We tested this constraint while enforcing an increasingly large set of ranges, and also while testing larger Diorama graph sizes $(8 \times 8, 16 \times 16, 24 \times 24, 32 \times 32)$. One can see that while ASP is competitive with MONOSAT in smaller graphs and with smaller shortest-path range constraints, both as the size of the shortest path range constraint increases, and also as the size of the graph itself increases, the encodings of shortest path constraints in both SAT and ASP scale poorly.

The two-sided encoding for shortest paths into CNF that we compare to here is to symbolically unroll the Floyd-Warshall algorithm (similar to encoding two-sided reachability constraints). A shortest path predicate $shortestPath_{s,t}(G) \leq L$, which evaluates to TRUE iff the shortest path in G is less than or equal to constant L, can be encoded into CNF using $\mathcal{O}(L \cdot |E| \cdot |V|)$ clauses (with E the set of edges, and V the set of vertices, in G). This encoding is practical only for very small graphs, or for small, constant values of L (as can be seen in the performance of

Components	MONOSAT	CLASP
8 Components	6s	98s
10 Components	6s	Timeout
12 Components	4s	Timeout
14 Components	0.82s	Timeout
16 Components	0.2s	Timeout

Table 6.3: Runtime results for connected components constraints in Diorama. Here CLASP can solve only the smallest instances.

MINISAT in Table 6.2).

Whereas ASP solvers have linear space encodings for reachability, the encodings of shortest path constraints into ASP are the same as for SAT solvers. There are also $\mathcal{O}(|E| \cdot |V| \cdot \log |V|)$ encodings of shortest paths into the theory of bitvectors (see, e.g., [90]), and $\mathcal{O}(|E| \cdot |V|)$ encodings into the theories of linear arithmetic or integer arithmetic, using comparison constraints. However, while these SMT encodings are concise, they essentially force the solver to non-deterministically guess the minimum distances to each node in the graph, and perform very poorly in practice (as shown in [90]).

Table 6.2 shows large performance improvements over CLASP and MINISAT.

Connected Components: We modify the Diorama constraints such that the generated map must consist of exactly k different terrain 'regions', where a region is a set of contiguous terrain positions of the same height. This produces terrain with a small number of large, natural-looking, contiguous ocean, plains, hills, and mountain regions. We tested this constraint for the 16×16 size Diorama instance, with $k = \{8, 10, 12, 14, 16\}$. In MONOSAT, this encoding uses two connected component count predicates: $components_{\geq}(E,k) \wedge \neg components_{\geq}(E,k+1)$, over an additional undirected graph in which adjacent grid positions with the same terrain height are connected.

The Diorama instances with connected component constraints are significantly harder for both solvers, and in fact neither solver could solve these instances in reasonable time for Diorama instances larger than 8×8. Additionally, for these instances, we disabled the 'undulate' constraint, as well as the reachability constraint,

Maximum Flow	MONOSAT	CLASP	MINISAT
8×8, max-flow=16	2s	2s	1s
16×16, max-flow=8	9s	483s	>3600s
16×16, max-flow=16	8s	27s	>3600s
16×16, max-flow=24	14s	26s	>3600s
24×24, max-flow=16	81s	>3600s	>3600s
32×32, max-flow=16	450s	>3600s	>3600s

Table 6.4: Runtime results for maximum flow constraints in Diorama. We can see that as the instance size increases, MONOSAT greatly outperforms CLASP.

as neither CLASP nor MONOSAT could solve the problem with these constraints combined with the connected components constraint. Results are presented in Table 6.3, showing that MONOSAT scales well as the number of required connected components is increased, whereas for CLASP, the constraints are only practical when the total number of constrained components is small.[3]

Maximum Flow: We modify the Diorama constraints such that each edge has a capacity of 4, and then enforce that the maximum flow between the top nodes and the bottom nodes of the terrain must be 8,16, or 24. This constraint prevents chokepoints between the top and bottom of the map.

Maximum flow constraints can be encoded into ASP using the built-in unary arithmetic support in $\mathcal{O}(|E| \cdot |V|^2)$ constraints. As discussed in Section 5.3.1, maximum flow constraints can also be encoded via the theory of bitvectors into pure CNF using $\mathcal{O}(|E| \cdot \log|V| + |V| \cdot \log|V|)$ constraints. However, neither of these encodings performs well in practice. In Table 6.4, we show that MONOSAT can handle maximum flow constraints on much larger graphs that CLASP or MINISAT. In fact, MONOSAT's maximum flow predicate is highly scalable; we will have more to say about this in Sections 6.2 and 6.3.

Minimum Spanning Trees: A common approach to generating random, traditional, 2D pen-and-paper mazes, is to find the minimum spanning tree of a ran-

[3] The entries begin at 8, as forcing smaller numbers of connected components than 8 was unsatisfiable.

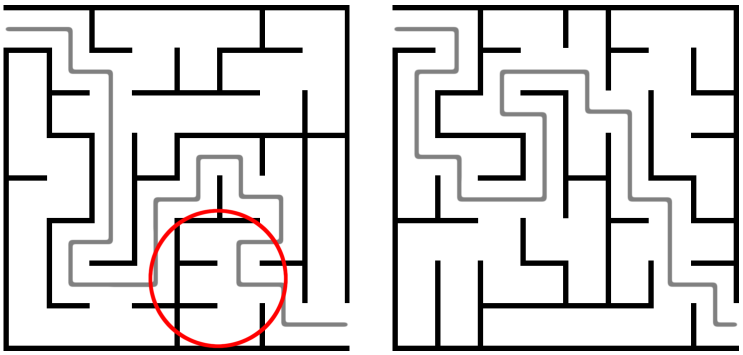


Figure 6.2: Random Mazes. Mazes generated through a combination of minimum spanning tree edge and weight constraints, and a constraint on the length of the path from start to finish. On the left, an un-optimized maze, with awkward, comb-structured walls (circled). On the right, an optimized maze, generated in seconds by MONOSAT.

domly weighted graph. Here, we consider a related problem: generating a random maze with a shortest start-to-finish path of a certain length.

We model this problem with two graphs, G_1 and G_2. In the first graph, we have randomly weighted edges arranged in a grid. The random edge weights make it likely that a minimum spanning tree of this graph will form a visually complex maze. In the second graph we have all the same edges, but unweighted; these unweighted edges will be used to constrain the length of the shortest path through the maze. Edges in G_2 are constrained to be enabled if and only if the corresponding edges in G_1 are elements of the minimum spanning tree of G_1. We then enforce that the shortest path in G_2 between the start and end nodes is within some specified range. Since the only edges enabled in G_2 are the edges of the minimum spanning tree of G_1, this condition constrains that the path length between the start and end node in the minimum spanning tree of G_1 be within these bounds. Finally, we constrain the graph to be connected. Together, the combined constraints on these two graphs will produce a maze with guaranteed bounds on the length of the shortest path solution.[4]

[4] While one could use the connected component count predicate to enforce connectedness, as we are already computing the minimum spanning tree for this graph in the solver, we can more efficiently just enforce the constraint that the minimum spanning tree has weight less than infinity.

Spanning Tree	MONOSAT	CLASP
Maze 5×5	< 0.1s	15s
Maze 8×8	1.5s	Timeout
Maze 16×16	32s	Timeout

Table 6.5: Runtime results for maze generation using minimum spanning tree constraints, comparing MONOSAT and CLASP. In these instances, the shortest path through the maze was constrained to a length between 3 and 4 times the width of the maze.

The solver must select a set of edges in G_1 to enable and disable such that the minimum spanning tree of the resulting graph is a) connected and b) results in a maze with a shortest start-to-finish path within the requested bounds. By itself, these constraints can result in poor-quality mazes (see Figure 6.2, left, and notice the unnatural wall formations circled in red); by allowing the edges in G_1 to be enabled or disabled freely, any tree can become the minimum spanning tree, effectively eliminating the effect of the random edge weight constraints.

Instead we convert this into an optimization problem, by combining it with an additional constraint: that the minimum spanning tree of G_1 must be $\leq$ to some constant, which we then lower repeatedly until it cannot be lowered any further without making the instance unsatisfiable.[5] This produces plausible mazes (see Figure 6.2, right), while also satisfying the shortest path constraints, and can be solved in reasonable time using MONOSAT (Figure 6.5).

[5] MONOSAT has built-in support for optimization problems via linear or binary search, and CLASP supports minimization via the "#minimize" statement.

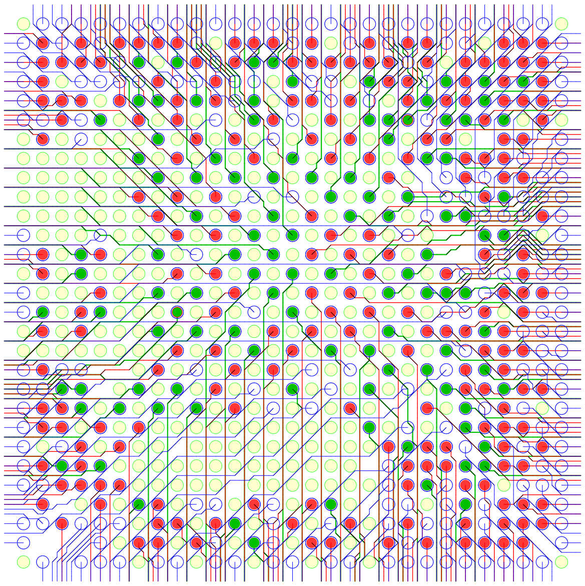


Figure 6.3: A multi-layer escape routing produced by MONOSAT for the TI TMS320C, an IC with a 29 × 29 ball grid array.

6.2 Escape Routing

The procedural content generation examples provide a good overview of the performance of many of the graph predicates described in Chapter 5 on graph-based procedural content generation tasks. We next turn to a real-world, industrial application for our theory of graphs: escape routing for Printed Circuit Board (PCB) layout.

In order to connect an integrated circuit (IC) to a PCB, traces must be routed on the PCB to connect each pin or pad on the package of the IC to its appropriate destination. PCB routing is a challenging problem that has given rise to a large body of research (e.g., [127, 137, 211]). However, high-density packages with large pin-counts, such as ball grid arrays, can be too difficult to route globally in a single step. Instead, initially an *escape routing* is found for the package, and only afterward is that escape routing connected to the rest of the PCB. Escape routing arises in particular when finding layouts for ball grid arrays (BGAs), which are ICs with dense grids of pins or pads covering an entire face of the IC (Figure 6.3).

In escape routing, the goal of connecting each signal pin on the package to its intended destination is (temporarily) relaxed. Instead, an easier initial problem is considered: find a path from each signal pin to *any* location of the PCB that is on the perimeter of the IC (and may be on any layer of the PCB). Once such an escape routing has been found, each of those escaped traces is routed to its intended destination in a subsequent step. That subsequent routing is not typically considered part of the escape routing process.

Many variants of the escape routing problem have been considered in the literature. Single-layer escape routing can be solved efficiently using maximum-flow algorithms, and has been explored in many studies [52, 93, 94, 201, 207, 210]; a good survey of these can be found in [208]. [143] uses a SAT solver to perform single-layer escape routing under additional 'ordering constraints' on some of the traces. SAT and SMT solvers have also been applied to many other aspects of circuit layout (e.g., [59, 82, 88, 89, 152] applied SAT, SMT, and ASP solvers to rectilinear or VLSI wire routing, and [90] applied an SMT solver to clock-routing). To the best of our knowledge, we are the first to apply SAT or SMT solvers to multi-layer escape routing.

For our purposes, a printed circuit board consists of one or more layers, with the BGA connected to the top-most layer. Some layers of the PCB are reserved just for ground or for power connections, while the remaining layers, sometimes including the top-most layer that the package connects to, are routable layers. Signals along an individual layer are conducted by metal traces, while signals crossing layers are conducted by vias. Typically, vias have substantially wider diameters than traces, such that the placement of a via prevents the placement of neighbouring traces. Different manufacturing processes support traces or vias with different diameters; denser printing capabilities can allow for multiple traces to fit between adjacent BGA pads (or, conversely, can support tighter spacing between adjacent BGA pads).

However, because the placement of vias between layers occludes the placement of nearby traces on those layers, multi-layer escape routing cannot be modeled correctly as a maximum flow problem. Instead, multi-layer escape routing has typically been solved using a greedy, layer-by-layer approach(e.g., [202]). Below, we show how multi-layer escape routing can be modeled correctly by combining

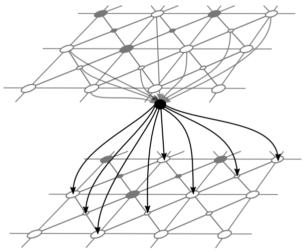


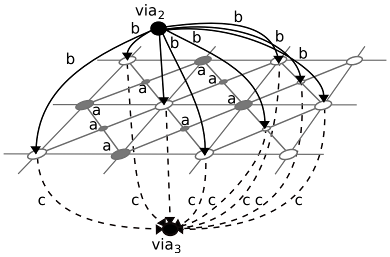
Figure 6.4: Multi-layer escape routing with simultaneous via-placement. On-grid positions are shown as large nodes, while 45-degree traces pass through the small nodes. This is a *symbolic* graph, in which some of the nodes or edges in this graph are included only if corresponding Boolean variables in an associated formula ϕ are assigned TRUE. We construct ϕ such that nodes connecting adjacent layers (the central black node, representing a via) are included in the graph only if the nodes surrounding the via (marked in gray) are disabled. The via node (center, black) is connected to all nodes around the periphery of the gray nodes, as well as to the central node (interior to the gray nodes).

the maximum flow predicate of Chapter 5 with additional Boolean constraints, and solved efficiently using MONOSAT.

6.2.1 Multi-Layer Escape Routing in MONOSAT

Figure 6.4 illustrates the symbolic flow graph we use to model multi-layer escape routing. Each layer of this flow-graph is similar to typical single-layer network flow-based escape routing solutions (see, e.g., [207]), except that in our graph all positions are potentially routable, with no spaces reserved for vias or pads. Each node in the graph has a node capacity of 1 (with node capacities enforced by introducing pairs of nodes connected by a single edge of capacity 1).

We also include *potential* vias in the graph, spaced at regular intervals, that the solver may choose to include or exclude from the graph. The potential via is shown as a black node in Figure 6.4, with neighbouring nodes shown in gray, indicating that they would be blocked by that via. In Figure 6.4, we show in gray the nodes that would be blocked by a via with a radius roughly 1.5 times the width of a trace. However, different via widths can be easily supported by simply altering the pattern of gray nodes to be blocked by the via.



$$via_2 \rightarrow \neg a_i$$

$$via_3 \rightarrow \neg a_i$$

$$(via_2 \wedge \neg via_3) \leftrightarrow b_i$$

$$(\neg via_2 \wedge via_3) \leftrightarrow c_i$$

Figure 6.5: Detail from layer 2 of Fig. 6.4, showing some symbolic nodes and edges controlled by Boolean variables $(a_i, b_i, c_i, via_2, via_3)$ in formula ϕ. To avoid clutter, the picture shows multiple edges with labels a, b, and c, but formally, each edge will have its own Boolean variable a_i, b_i, and c_i. All nodes and edges have capacity 1, however, nodes and edges with associated variables are only included in the graph if their variable is assigned TRUE. Two via nodes are shown, one connecting from the layer above, and one connecting to the layer below. Nodes in the layer that are occluded if a via is placed in this position are shown in gray (in this case, the via has a diameter twice the width of a trace, but any width of via can be modeled simply by adjusting the pattern of nodes blocked by the via). The first two constraints shown enforce that if either via node is included in G, then the nodes in the layer that would be occluded by the via (in gray) must be disabled. The remaining constraints allow the nodes surrounding the blocked nodes to connect to the via if and only if the via begins or ends at this layer (rather than passing through from an upper to a lower layer). These constraints are included in ϕ for each potential via location at each layer in G.

Each via node is connected to the nodes surrounding the gray nodes that are blocked by the via, as well as the central node interior to the gray nodes (see Figure 6.5). These represent routable connections on the layer if the via is placed, allowing traces to be routed from the via nodes to the nodes surrounding the blocked nodes, or allowing the via to route through the layer and down to the next layer below. Each via node is associated with a Boolean variable (via_2 in Figure 6.5), such that the via node is included in the graph if and only if via_2 is TRUE in ϕ. The potentially blocked nodes around each via are also associated with variables (for clarity, drawn as a in Figure 6.5, however in ϕ each edge will actually have a unique variable a_i). For each via, we include constraints $via \rightarrow \neg a_i$ in ϕ, disabling all the immediate neighbouring nodes if the via is included in the graph. Any satisfiable assignment to these constraints in ϕ selects a subset of the nodes of G representing a compatible, non-overlapping set of vias and traces.

Four configurations are possible for the two via nodes shown in Figure 6.5: (a) neither via node is enabled, allowing traces to be routed through the gray nodes on this layer, (b) the via enters from above, and connects to this layer, (c) the via begins at this layer, connecting to a layer below, and (d) the via passes through this layer, connecting the layer above to the layer below. By adding constraints restricting the allowable configurations of vias, as described in Figure 6.6, our approach can model all the commonly used via types: through-hole vias, buried vias, blind vias, or any-layer micro-vias. With minor adjustments to the constraints, these different via types can be combined into a single model or can be restricted to specific layers of the PCB, allowing a wide variety of PCB manufacturing processes to be supported.

We add an additional source node s and sink node t to the graph, with directed, capacity-1 edges connecting the s to each signal in the graph, and directed, capacity-1 edges connecting all nodes on the perimeter of each layer to t. Finally, a single flow constraint $maxflow_{s,t}(E) \geq |signals|$ is added to ϕ, ensuring that in any satisfying assignment, the subset of edges included in the graph must admit a flow corresponding to a valid escape routing for all of the signal pins.

A solution to this formula corresponds to a feasible multi-layer escape routing, including via and trace placement. However, as MONOSAT only supports maximum flow constraints, and not minimum-cost maximum flow constraints, the trace routing in this solution is typically far from optimal (with traces making completely unnecessary detours, for example). For this reason, once MONOSAT has produced a feasible escape routing, including a placement of each via, we then apply an off-the-shelf minimum-cost maximum flow solver [165] to find a corresponding locally optimal trace routing for each individual layer of the feasible routing. This can be solved using completely standard linear-programming encodings of minimum-cost maximum flow, as the vias are already placed and the layer that each signal is to be routed on is already known.

Via Type	Constraint
Through-hole	$via_j \rightarrow (\neg a_i^1 \wedge \neg a_i^2 \wedge \ldots \neg a_i^n)$
Blind	$via_j \rightarrow (\neg a_i^1 \wedge \neg a_i^2 \wedge \ldots \neg a_i^j)$
Buried	$via_j \rightarrow (\neg a_i^s \wedge \neg a_i^{s+1} \wedge \ldots \neg a_i^t)$
Micro	—

Figure 6.6: Constraints enforcing different via models in ϕ, for $via_1 \ldots via_n$, where variables a_i^k control the potentially blocked nodes of via_j. (For variable a_i^k, the index k indicates the layer number, and the constraint is enforced for all values of i of neighbouring nodes to the via.) Through-hole vias are holes drilled through all layers of the PCB; the corresponding constraints block the placement of traces at the position of the through-hole on all layers of the PCB. Buried and blind vias drill through a span of adjacent layers, with blind vias always drilling all the way to either the topmost or bottom layer of the PCB (we show the case for blind vias starting at the top layer, and continuing to layer j, above). In the constraints for buried vias, s and t are the allowable start and end layers for the buried via, determined by the type of buried via. Micro-vias allow any two adjacent layers to be connected and require no additional constraints (the default behaviour); if only a subset of the layers support micro-vias, then this can be easily enforced. Each of these via types can also be combined together in one routing or (excepting through-holes) restricted to a subset of the layers.

6.2.2 Evaluation

We evaluate our procedure on a wide variety of dense ball grid arrays from four different companies, ranging in size from a 28×28 pad ARM processor with 382 routable signal pins to a 54×54 pad FPGA with 1755 routable signal pins. These parts, listed in Tables 6.6 and 6.7, include 32-bit and 64-bit processors, FPGAs, and SoCs. The first seven of these packages use 0.8mm pitch pads, while the remainder use 1mm pitch pads. Each part has, in addition to the signal pins to be escaped, a roughly similar number of power and ground pins (for example, the 54×54 Xilinx FPGA has 1137 power and ground pins, in addition to the 1755 signal pins). Most parts also have a small number of disconnected pins, which are not routed at all. Typically, power and ground pins are routed to a number of dedicated power and ground layers in the PCB, separately from the signal traces; we assume that the bottom-most layers of the PCB contain the power and ground layers, and route all power and ground pins to those layers with through-hole vias. This leaves only the routable signal pins to be escaped in each part on the remaining layers.

For comparison, we implemented a simple network-flow-based single-layer escape routing algorithm, similar to the one described in [207]. We then implemented a greedy, layer-by-layer router by routing as many signals as possible on the top-most layer (using maximum flow), and, while unrouted signals remain, adding a new layer with vias connecting to each unrouted signal. This process repeats until no unrouted signals remain. As can be seen in Table 6.6, this layer-by-layer routing strategy is simple but effective, and has been previously suggested for multi-layer escape routing in several works in the literature (for example, [202] combines this strategy with their single-layer routing heuristic to create a multi-layer escape routing method).

In Table 6.6, we compare our approach to the layer-by-layer strategy using blind vias, and, in Table 6.7, using through-hole vias. All experiments were run on a 2.67GHz Intel x5650 CPU (12Mb L3, 96 Gb RAM), in Ubuntu 12.04. Although our approach supports buried and micro-vias, we found that all of these instances could be solved by MONOSAT with just 2 or 3 signal layers, even when using the more restrictive through-hole and blind via models, and so we omit evaluations for these less restrictive models (which, in two or three layer PCBs, are nearly equivalent to blind vias).

In Tables 6.6 and 6.7, MONOSAT finds many solutions requiring fewer layers than the layer-by-layer strategy (and in no case requires more layers than the layer-by-layer approach). For example, in Table 6.6, MONOSAT finds a solution using blind vias (for the TI AM5K2E04 processor, packaged in a dense, 33 × 33 pad, 0.8mm pitch BGA) which requires only 3 signal layers, whereas the layer-by-layer approach requires 4 signal layers for the same part. In this case, MONOSAT was also able to prove that no escape routing using 2 or fewer layers was possible for this circuit (assuming the same grid-model is used). In Table 6.7, using more restrictive through-vias, there are several examples where MONOSAT finds solutions using 1 or even 2 fewer signal layers than the layer-by-layer approach.

The runtimes required by MONOSAT to solve these instances are reasonable, spanning from a few seconds to a little less than 2 hours for the largest instance

Part	Size	Layer-by-Layer		MONOSAT	
		Layers	Time (s)	Layers	Time (s)
TI AM5716	28 × 28	3	5.4s + 63.6s	**2***	54.4s + 81.2s
TI AM5718	28 × 28	3	5.4s + 70.4s	**2***	47.4s + 118.4s
TI AM5726	28 × 28	3	5.3s + 49.7s	3	53.3s + 492.8s
TI AM5728	28 × 28	3	5.3s + 48.6s	3	53.8s + 387.2s
TI TMS320C	29 × 29	4	7.7s + 81.5s	**3**	75.0s + 497.7s
TI AM52E02	33 × 33	4	9.5s + 107.7s	**3**	103.8 + 921.8
TI AM5K2E04	33 × 33	4	9.2s + 96.8s	**3***	114.7s + 962.0s
TI 66AK2H1	39 × 39	3	24.0s + 508.0s	**2***	338.4s + 878.1s
Lattice M25	32 × 32	2*	10.4s + 160.5s	2*	140.1s + 306.7s
Lattice M40	32 × 32	3	114.6s + 205.6s	**2***	194.0s + 364.4s
Lattice M40	34 × 34	3	18.5s + 300.0s	**2***	254.3s + 425.2s
Lattice M80	34 × 34	3	17.8s + 266.5s	**2***	411.3s + 505.8s
Lattice M80	42 × 42	3	27.0s + 499.0s	3	810.3s + 882.4s
Lattice M115	34 × 34	3	16.9s + 274.1s	**2***	392.9s + 578.2s
Lattice M115	42 × 42	3	27.4s + 461.4s	3	242.5s + 254.7s
Altera 10AX048	28 × 28	2*	8.0s + 109.3s	2*	85.1s + 183.4s
Altera 10AX066	34 × 34	2*	13.1s + 218.3s	2*	151.1s + 371.2s
Altera 10AX115	34 × 34	2*	13.9s + 286.8s	2*	168.5s + 501.9s
Altera 10AT115	44 × 44	3	29.6s + 579.5s	3	384.8s + 928.8s
Altera EP4S100	44 × 44	3	31.0s + 698.6s	3	401.8s + 1154.6s
Xilinx XCVU160	46 × 46	3	34.3s + 617.9s	3	414.3s + 977.5s
Xilinx XCVU440	49 × 49	4	52.2s + 1167.5s	**3***	1246.9s + 2133.7s
Xilinx XCVU440	54 × 54	4	60.9s + 1438.1s	**3***	1597.3s + 2726.9s

Table 6.6: Multi-layer escape routing with blind vias. *Run-times are reported as $a + b$, where a is the time to find a feasible multi-layer routing, and b is the time to post-process that feasible solution using minimum-cost maximum flow routing. Boldface highlights when our approach required fewer layers; solutions that use a provably minimal number of layers are marked with *. Length shows the average trace length in mm. These solutions ignore differential pair constraints (routing differential signals as if they were normal signals).*

considered. These runtimes are all the more impressive when considering that the graphs encoded in some of these instances are very large by SAT solver standards, with more than 1,000,000 nodes (and nearly 4,000,000 edges) in the formula for the 54 × 54 Xilinx FPGA.

Part	Size	Layer-by-Layer		MonoSAT	
		Layers	Time (s)	Layers	Time (s)
TI AM5716	28×28	3	5.4s + 58.9s	**2***	35.8s + 62.9s
TI AM5718	28×28	3	5.1s + 61.8s	**2***	46.0s + 69.1s
TI AM5726	28×28	4	7.7s + 63.1s	**3**	97.6s + 80.2s
TI AM5728	28×28	4	6.8s + 56.7s	**3**	110.1s + 85.4s
TI TMS320C	29×29	5	10.9s + 106.1s	**3**	109.6s + 128.4s
TI AM52E02	33×33	5	13.4s + 133.3s	**3**	203.2s + 180.4s
TI AM5K2E04	33×33	5	12.7s + 125.4s	**3***	291.3s + 187.8s
TI 66AK2H1	39×39	3	24.9s + 495.1s	**2***	347.9s + 510.7s
Lattice M25	32×32	2*	10.7s + 143.7s	2*	132.3s + 278.7s
Lattice M40	32×32	3	15.3s + 183.1s	**2***	161.5s + 311.3s
Lattice M40	34×34	3	18.0s + 270.2s	**2***	183.9s + 405.6s
Lattice M80	34×34	3	17.7s + 238.4s	3	304.8s + 638.2s
Lattice M80	42×42	3	26.2s + 478.4s	3	810.3s + 882.4s
Lattice M115	34×34	3	16.8s + 227.1s	3	364.2s + 358.7s
Lattice M115	42×42	3	27.8s + 457.3s	3	945.9s + 1500.3s
Altera 10AX048	28×28	2*	8.2s + 98.2s	2*	109.2s + 115.8s
Altera 10AX066	34×34	2*	14.0s + 212.3s	2*	203.5s + 282.9s
Altera 10AX115	34×34	2*	13.3s + 235.1s	2*	198.2s + 293.7s
Altera 10AT115	44×44	3	28.9s + 455.2s	3	616.1s + 992.9s
Altera EP4S100	44×44	3	28.5s + 589.2s	3	733.5s + 834.5s
Xilinx XCVU160	46×46	3	32.5s + 538.3s	3	646.6s + 1216.4s
Xilinx XCVU440	49×49	4	53.7s + 1051.1s	**3**	3457.5s + 1284.5s
Xilinx XCVU440	54×54	4	600s + 1373.6s	**3**	6176.9s + 1861.9s

Table 6.7: Multi-layer escape routing with through-vias. *Run-times are reported as $a+b$, where a is the time to find a feasible multi-layer routing, and b is the time to post-process that feasible solution using minimum-cost maximum flow routing. Boldface highlights when our approach required fewer layers; solutions that use a provably minimal number of layers are marked with *. Length shows the average trace length in mm. These solutions ignore differential pair constraints (routing differential signals as if they were normal signals).*

6.3 Virtual Data Center Allocation

The final application we consider is *virtual data center* allocation. Virtual data center allocation [26, 113] is a challenging network resource allocation problem, in which instead of allocating a single virtual machine to the cloud, a connected virtual data center (VDC) consisting of several individual virtual machines must be allocated simultaneously, with guaranteed bandwidth between some or all of the virtual machines.

While allocating individual VMs to a data center is a well-studied problem, allocating virtual data centers remains an open research problem, with current commercial approaches lacking end-to-end bandwidth guarantees [1–3]. Solutions that do provide end-to-end bandwidth guarantees lack scalability [212], are restricted to data centers with limited or artificial topologies [22, 176, 212], or are *incomplete* [113], meaning that they may fail to find allocations even when feasible allocations exist, especially as load increases, resulting in under-utilized data center resources.

As we will show, we can formulate the VDC allocation problem as a multi-commodity flow problem, and solve it efficiently using a conjunction of maximum flow predicates from our theory of graphs. Using this approach, MONOSAT can allocate VDCs of up to 15 VMs to physical data centers with thousands of servers, even when those data centers are nearly saturated. In many cases, MONOSAT can allocate 150% – 300% as many total VDCs to the same physical data center as previous methods.

6.3.1 Problem Formulation

Formally, the VDC allocation problem[6] is to find an allocation of VMs to servers, and links in the virtual network to links in the physical network, that satisfies the compute, memory and network bandwidth requirements of each VM across the entire data center infrastructure, including servers, top-of-rack (ToR) switches and aggregation switches.

The physical network consists of a set of servers S, switches N, and a di-

[6] There are a number of closely related formalizations of the VDC allocation problem in the literature; here we follow the definition in [212].

rected graph $(S \cup N, L)$, with capacities $c(u,v)$ for each link $(u,v) \in L$. The virtual data center consists of a set of virtual machines VM and a set of directed bandwidth requirements $R \subseteq VM \times VM \times \mathbb{Z}^+$. For each server $s \in S$, we are given CPU core, RAM, and storage capacities $cpu(s), ram(s), storage(s)$, and each virtual machine $v \in VM$ has corresponding core, RAM, and storage requirements $cpu(v), ram(v), storage(v)$.

Given a PN and VDC defined as above, the multi-path VDC allocation problem is to find an assignment $A : VM \mapsto S$ of virtual machines $v \in VM$ to servers $s \in S$, and, for each bandwidth requirement $(u,v,b) \in R$, an assignment of non-negative bandwidth $B_{u,v}(l)$ to links $l \in L$, such that the following sets of constraints are satisfied:

(**L**) **Local VM allocation constraints.** These ensure that each virtual machine is assigned to exactly one server in the physical network (multiple VMs may be assigned to each server), and that each server has sufficient CPU core, RAM, and storage resources available to serve the sum total of requirements of the VMs allocated to it. Let $V(s) = \{v \in VM \mid A(v) = s\}$; then $\forall s \in S : \sum_{V(s)} cpu(v) \leq cpu(s) \wedge \sum_{V(s)} ram(v) \leq ram(s) \wedge \sum_{V(s)} storage(v) \leq storage(s)$. We model resource requirements using integer values, and assume that no sharing of resources between allocated VMs is allowed.

(**G**) **Global bandwidth allocation constraints**. These ensure that sufficient bandwidth is available in the physical network to satisfy all VM to VM bandwidth requirements in R simultaneously. Formally, we require that $\forall (u,v,b) \in R$, the assignments $B_{u,v}(l)$ form a valid $A(u) - A(v)$ network flow greater or equal to b, and that we respect the capacities of each link l in the physical network: $\forall l \in L : \sum_{(u,v,b) \in R} B_{u,v}(l) \leq c(l)$. We model bandwidths using integer values and assume that communication bandwidth between VMs allocated to the same server is unlimited.

Prior studies [54, 114, 193, 209] observed that if path-splitting is allowed, then the global bandwidth allocation constraints correspond to a multi-commodity flow problem, which is NP-complete even for undirected integral flows [92], but has poly-time solutions via linear programming if real-valued flows are allowed.[7] Since multi-commodity flow can be reduced to the multi-path VDC allocation problem, for the case of integer-valued flows, the multi-path VDC allocation problem is NP-hard [54].

Next, we will show how multi-commodity integral flow problems can be encoded as a conjunction of maximum flow constraints over graphs with symbolic edge weights. We will then provide a solution to the full multi-path VDC allocation problem by combining our multi-commodity flow encoding for global constraints **G** with a pseudo-Boolean encoding of local constraints **L**.

6.3.2 Multi-Commodity Flow in MONOSAT

We model multi-path, end-to-end bandwidth guarantees in MONOSAT as a multi-commodity flow problem. In this subsection, we describe how we model integer-value multi-commodity flow in terms of the built-in maximum flow predicates that MONOSAT supports; in the next subsection, we show how to use these multi-commodity flow constraints to express VDC allocation.

Before we introduce our encoding for integer-value multi-commodity flow, it is helpful to provide some context. SMT solvers have not traditionally been applied to large multi-commodity flow problems; rather multi-commodity flow problems are usually solved using integer-arithmetic solvers, or are approximated using linear programming. MONOSAT does not directly provide support for multi-commodity flows, but as we will show below, by expressing multi-commodity flows as a conjunction of single-commodity maximum flow predicates (which MONOSAT does support), we can use MONOSAT to solve large multi-commodity flow problems – a first for SMT solvers.

We consider this formulation of integer-value multi-commodity flows in terms of combinations of maximum flow predicates to be a key contribution of this section. While there are many obvious ways to encode multi-commodity flows in SMT solvers, the one we present here is, to the best of our knowledge, the only SMT en-

[7] Note that while linear programming supports global bandwidth constraints, it does not support the local server constraints. Therefore, approaches that model the global constraints as a linear program either include additional steps to perform local server allocation [209], or use mixed integer programming [54].

coding to scale to multi-commodity flow problems with thousands of nodes. As there are many applications to which SMT solvers are better suited than integer arithmetic solvers (and vice-versa), this SMT formulation has many potential applications beyond virtual data center allocation.

Given a directed graph $G = (V,E)$, an integer capacity $c(u,v)$ for each edge $(u,v) \in E$, and a set of commodity demands K, where a commodity demand $i \in K$ is a tuple (s_i,t_i,d_i), representing an integer flow demand of d_i between source $s_i \in V$ and target $t_i \in V$. The integral multi-commodity flow problem is to find a feasible flow such that each demand d_i is satisfied, while for each edge (u,v) the total flow of all capacities (summed) is at most $c(u,v)$:

$$f_i(u,v) \geq 0, \quad \forall (u,v) \in E, i \in K$$

$$\sum_{i\in K} f_i(u,v) \leq c(u,v), \quad \forall (u,v) \in E$$

$$\sum_{v\in V} f_i(u,v) - \sum_{v\in V} f_i(v,u) = \begin{cases} 0, \text{if } u \notin \{s_i,t_i\} \\ d, \text{if } u = s_i \\ -d, \text{if } u = t_i \end{cases} , \forall i \in K$$

To encode multi-commodity integral flow constraints in MONOSAT, we instantiate directed graphs $G_{1..|K|}$ with the same topology as G. For each edge $(u,v)_i \in G_i$, we set its capacity to be a fresh bitvector $c(u,v)_i$, subject to the constraint $0 \leq c(u,v)_i \leq c(u,v)$. We then assert for each edge (u,v) that $\sum_i c(u,v)_i \leq c(u,v)$ – that is, the capacities of each edge (u,v) in each commodity graph G_i must sum to at most the original capacity of edge (u,v). Finally, for each commodity demand (s_i,t_i,d_i), we assert that the maximum s_i–t_i flow in G_i is $\geq d_i$, using MONOSAT's built-in maximum flow constraints.

If the multi-commodity flow is feasible, the solver will find a partitioning of the capacities among the graphs G_i such that the maximum s_i–t_i flow in G_i is *at least* d_i for each commodity constraint i. We can then force each commodity flow in the solution to be *exactly* d_i by adding an extra node n_i to each graph G_i, an edge (t_i,n_i) with capacity d_i, and replacing the commodity demand (s_i,t_i,d_i) with (s_i,n_i,d_i).

6.3.3 Encoding Multi-Path VDC Allocation

The local constraints of the multi-path VDC allocation problem can be modeled as a set of pseudo-Boolean constraints, for which many efficient and direct encodings into propositional satisfiability (SAT) are known [21, 85]. The part of constraint set **L** (see Section 6.3.2) enforcing that each VM is assigned to at most one server is a special case of a so-called ‘at-most-one’ pseudo-Boolean constraint [53], which can be handled even more efficiently (in fact, MonoSAT has built-in theory support for large ‘at-most-one’ constraints). Constraint set **G** can be encoded as a multi-commodity flow as described above, with up to $|VM|^2$ commodity demands (one for each bandwidth tuple $(u, v, bandwidth) \in R$). However, we can greatly improve on this by grouping together bandwidth constraints that share a common source, and merging them into a single commodity demand: Given a set of bandwidth constraints $(u, v_i, bandwidth_i) \in R$ with the same source u, we can convert these into a single commodity demand by adding an extra node $w \notin VM$, along with edges (v_i, w) with capacity $bandwidth_i$. The commodity demands $(u, v_i, bandwidth_i)$ can then be replaced by a single commodity demand $(u, w, \sum_i bandwidth_i)$.

Since there are at most $|VM|$ distinct sources in R, this reduces the number of commodity demands from $|VM|^2$ in the worst case to $|VM|$. In cases where the VDC is undirected, we can improve on this further, by swapping sources and sinks in communication requirements so as to maximize the number of requirements with common sources. To do so, we construct the undirected graph of communication requirements, with an undirected edge of weight $(u, v) = bandwidth$ for each bandwidth requirement, and find an approximate minimum-cost vertex cover (using the 2-opt approximation from [24]). This can be done efficiently (in polynomial time) even for large networks. Necessarily, each edge, and hence each communication requirement, will have at least one covering vertex. For each requirement $(u, v, bandwidth)$, if v is a covering vertex and u is not, we replace the requirement with $(v, u, bandwidth)$, swapping u and v. After swapping all un-covered source vertices in this way, we then proceed to merge requirements with common sources as above. For cases where the VDC is directed, we skip this cover-finding optimization, and only merge together connection requirements that happen to have the same (directed) source in the input description.

Given this optimized set of commodity demands, we construct a directed graph G consisting of the physical network $(S \cup N, L)$, and one node for each virtual machine in VM. If any VDC communication requirements $(u, v_i, bandwidth_i)$ have been merged into combined requirements $(u, w, \sum bandwidth_i)$ as above, we add additional, directed edges (v_i, w) with capacity $bandwidth_i$ to G.

For each $v \in VM$ and each server $s \in S$, we add a directed symbolic edge e_{vs} from v to s with unlimited capacity to G; this edge controls the server to which each VM is allocated. Note that only the VM allocation edges e_{vs} have to be symbolic; all remaining edges in G have known constant capacities and can be asserted to be in G.

We assert (using MONOSAT's theory of pseudo-Boolean constraints, as described in Algorithm 5 of Chapter 4) that for each VM v, exactly one edge e_{vs} is enabled, so that the VM is allocated to exactly one server: $\forall v \in VM : \sum_s e_{vs} = 1$. For each server s, we assert $\sum_v cpu(v) \leq cpu(s) \wedge \sum_v RAM(v) \leq RAM(s) \wedge \sum_v storage(v) \leq storage(s)$, i.e. that the set of VMs allocated to each server (which may be more than one VM per server) will have sufficient CPU core, RAM, and storage resources available on that server. Together these assertions enforce constraint set **L** from our problem definition.

Finally, using the multi-commodity flow encoding described above, we assert that the multi-commodity flow in G satisfies $(u, v, bandwidth)$ for each optimized commodity requirement. When constructing the individual commodity flow graphs G_i from G, we add an assertion that each VM-server allocation edge e_{vs} is contained in G_i if, and only if, that edge is included in G; this ensures that the same VM-server allocation edges are enabled in each commodity flow graph.

6.3.4 Evaluation

We compare the performance of MONOSAT to that of two previous VDC tools: SecondNet's VDCAlloc algorithm [113] — the seminal VDC allocation tool with sound, end-to-end bandwidth guarantees — and the Z3-based abstraction-refinement technique from [212], the tool most similar to our own contribution.

SecondNet's VDCAlloc algorithm [113]. SecondNet's VDCAlloc algorithm ('SecondNet', except where ambiguous) is an incomplete, heuristic-driven algorithm based on bipartite matching, that is fast (much faster than MONOSAT) and scales well to physical networks with even hundreds of thousands of servers (whereas MONOSAT scales only to a few thousand servers).

However, while SecondNet scales very well, it also has major limitations. As it is based on bipartite matching, it fundamentally cannot allocate more than one VM in each VDC to any given server. SecondNet also performs allocation in an incomplete, greedy fashion: it commits to a node allocation before attempting link mapping, and it maps links from the virtual network one at a time. In heavily utilized networks, this greedy process can fail to find a feasible allocation of a virtual data center, even when a feasible allocation exists. Below we will show that in many realistic circumstances, SecondNet allocates less than half of the total feasible allocations, and sometimes less than a third.

Abstraction-refinement technique based on Z3 [212]. The authors of [212] introduced two approaches for performing single-path VDC allocation with bandwidth guarantees that use the SMT solver Z3 [70]. Unlike the SMT solver MONOSAT used by MONOSAT, Z3 has no built-in support for graph predicates. Therefore, a major challenge tackled by [212] was to efficiently represent the global bandwidth and connectivity constraints in the low-level logic of Z3. For the special case of physical data-centers with proper tree topologies, the authors introduced an efficient encoding of these constraints that can be represented in Z3 using only a linear number of constraints in the number of servers, which they show performs well enough to scale to several hundred servers (but can only be applied to tree topologies).

The first approach from [212](which we call Z3-generic) uses an encoding that can handle any data center topology, but as was shown in [212], scales extremely poorly. The second approach (which we call Z3-AR) is an optimized abstraction-refinement technique with Z3 as its back-end solver. This approach is more scalable than the generic encoding, but is restricted to data centers with tree topologies. In our experiments we found that Z3-generic performed poorly, often failing to find any allocations within a 1-hour timeout. For brevity we do not report results for Z3-generic.

	VDC Instance Structure											
	vn3.1		vn3.2		vn3.3		vn5.1		vn5.2		vn5.3	
	#VDC	Time	#VDC	Time	#VDC	Time	#VDC	Time	#VDC	Time	#VDC	Time
Tree Physical Data Center, 200 servers with 4 cores each												
SecondNet†	**88**	< 1	**88**	< 1	87	< 1	48	2.1	51	2.2	52	2.2
[212]-AR	**88**	17.9	**88**	33.5	**88**	34.7	**53**	61.9	**53**	59.9	**53**	55.1
MonoSAT	**88**	4.7	**88**	6.8	**88**	6.3	**53**	7.3	**53**	7.1	**53**	9.0
Tree Physical Data Center, 200 servers with 16 cores each												
SecondNet†	313	< 1	171	< 1	129	< 1	56	2.6	59	2.9	57	2.7
[212]-AR	**355**	76.0	301	(3600)	300	(3600)	88	(3600)	50	(3600)	24	(3600)
MonoSAT	**355**	15.1	**353**	22.4	**352**	20.45	**201**	22.1	**201**	22.6	**202**	25.6
Tree Physical Data Center, 400 servers with 16 cores each												
SecondNet†	628	2.1	342	1.0	257	1.2	109	18.6	117	21.5	114	18.7
[212]-AR	**711**	209.9	678	(3600)	691	3547.9	72	(3600)	43	(3600)	77	(3600)
MonoSAT	**711**	55.4	**709**	86.44	**705**	78.6	**404**	90.3	**405**	85.4	**405**	99.8
Tree Physical Data Center, 2000 servers with 16 cores each												
SecondNet†	3140	48.3	1712	23.8	1286	30.7	539	2487.7	582	2679.0	567	2515.5
[212]-AR	**3555**	2803.7	741	(3600)	660	(3600)	76	(3600)	86	(3600)	204	(3600)
MonoSAT	3554	1558.2	**3541**	2495.7	**3528**	2375.1	**958**	(3600)	**1668**	(3600)	**1889**	(3600)

Table 6.8: Total Number of Consecutive VDCs Allocated on Data Centers with Tree Topologies. The six major columns under "VDC Instance Structure" give results for serial allocation of the six different VDC types from [212]. '#VDC' is the number of VDCs allocated until failure to find an allocation, or until the 1h timeout; 'Time' is the total runtime for all allocations, in seconds; 'Mdn' is the median runtime per allocation, in seconds. '(3600)' indicates that allocations were stopped at the timeout. Largest allocations are in **boldface**. In this table, 'SecondNet†' is an implementation of SecondNet's VDCAlloc algorithm [113], modified to handle tree topologies by the authors of [212]. MonoSAT is much faster than [212]-AR but slower than SecondNet. In most cases where it does not time out, MonoSAT is able to allocate several times as many VDCs as SecondNet, in a reasonable amount of time per VDC (usually less than one second per VDC, except on the largest instances).

To ensure a fair comparison, we used the original implementations of these tools, with minor bug fixes and latest enhancements, obtained from the original authors.

Comparison on Trees from [212] Our first experiment reproduces and extends an experiment from [212], in which a series of identically structured VDCs are allocated one-by-one to tree-structured data centers until the solver is unable to make further allocations (or a timeout of 1 CPU hour is reached). We obtained the original implementation Z3-AR from the authors for this experiment, along with a version of SecondNet they implemented with support for the tree-structured data centers considered here. In this experiment, the VDCs being allocated always have identical structure; this is a limitation introduced here for compatibility with the solvers from [212]. In our subsequent experiments, below, we will consider more realistic allocation scenarios. Except where noted, all experiments were conducted on a 2.66GHz (12Mb L3 cache) Intel x5650 processor, running Ubuntu 12.04, and limited to 16GB RAM.

We started with the 200-server/4-cores-per-server physical data center from [212], but then considered larger versions of the original benchmark to study performance scaling. The larger data centers are also more representative of current data centers, using 16-core servers rather than the 4-core servers from the original paper.

Table 6.8 summarizes our results. SecondNet, being heuristic and incomplete, is much faster than MONOSAT, but MONOSAT is much faster and more scalable than Z3-AR. Importantly, we see that MONOSAT is able to scale to thousands of servers, with typical per-instance allocation times of a few seconds or less per VDC. Furthermore, on these tree-structured data centers, MONOSAT is typically able to allocate two or even three times as many VDCs as SecondNet onto the same infrastructure.

Comparison on BCube and FatTree from [113] The second experiment we conducted is a direct comparison against the original SecondNet implementation, with the latest updates and bug fixes, obtained from its original authors [113] (this is also the version of SecondNet we use for all subsequent comparisons in this section). Note that the implementation of Z3-generic, and both the theory and implementation of Z3-AR, are restricted to tree topologies, so they could not be included in these experiments.

The SecondNet benchmark instances are extremely large — in one case ex-

	VDC Instance Structure											
	vn3.1		vn3.2		vn3.3		vn5.1		vn5.2		vn5.3	
	#VDC	Time (s)	#VDC	Time	#VDC	Time	#VDC	Time	#VDC	Time	#VDC	Time
FatTree Physical Data Center, 432 servers with 16 cores each												
SecondNet	624	< 0.1	360	< 0.1	252	< 0.1	132	< 0.1	132	< 0.1	120	< 0.1
MonoSAT	**768**	148.3	**762**	257.9	**760**	247.1	**448**	317.3	**438**	315.9	**436**	323.26
BCube Physical Data Center, 512 Servers with 16 cores each												
SecondNet	833	< 0.1	486	< 0.1	386	< 0.1	157	< 0.1	195	< 0.1	144	< 0.1
MonoSAT	**909**	201.0	**881**	347.3	**869**	310.4	**473**	502.7	**440**	423.3	**460**	412.8
FatTree Physical Data Center, 1024 servers with 16 cores each												
SecondNet	1520	< 0.1	848	< 0.1	608	< 0.1	272	< 0.1	288	< 0.1	278	< 0.1
MonoSAT	**1820**	787.6	**1812**	1435.2	**1806**	1437.5	**1064**	1941.5	**1031**	1925.2	**963**	(3600)
BCube Data Center, 1000 Servers with 16 cores each												
SecondNet	1774	< 0.1	1559	1.4	1188	2.0	302	< 0.1	356	< 0.1	293	< 0.1
MonoSAT	**1775**	746.6	**1713**	1350.4	**1677**	1513.8	**938**	1741.8	**884**	1747.1	912	1697.0

Table 6.9: Total Number of Consecutive VDCs Allocated on Data Centers with FatTree and BCube Topologies. Table labels are the same as in Table 6.8, but in this table, 'SecondNet' refers to the original implementation. As before, SecondNet is much faster, but MonoSAT scales well up to hundreds of servers, typically allocating VDCs in less than a second. And in most cases, MonoSAT allocated many more VDCs to the same physical data center. (An interesting exception is the lower-left corner. Although MonoSAT is complete for any specific allocation, it is allocating VDCs one-at-a-time in an online manner, as do other tools, so the overall allocation could end up being suboptimal.)

ceeding 100 000 servers — but also extremely easy to allocate: the available bandwidth per link is typically $\geq 50\times$ the requested communication bandwidths in the VDC, so with only 16 cores per server, the bandwidth constraints are mostly irrelevant. For such easy allocations, the fast, incomplete approach that SecondNet uses is the better solution. Accordingly, we scaled the SecondNet instances down to 432–1024 servers, a realistic size for many real-world data centers. For these experiments, we generated sets of 10 VDCs each of several sizes (6, 9, 12 and 15 VMs), following the methodology described in [212]. These VDCs have proportionally greater bandwidth requirements than those originally considered by SecondNet, requiring 5–10% of the smallest link-level capacities. The resulting VDC instances are large enough to be representative of many real-world use cases, while also exhibiting non-trivial bandwidth constraints. For each of these sets of VDCs, we then repeatedly allocate instances (in random order) until the data center is saturated.

Table 6.9 shows allocations made by SecondNet and MONOSAT on two data centers, one with a BCube topology with 512 servers, and one with a FatTree topology with 432 servers. As in our previous experiment, SecondNet is much faster than MONOSAT, but MONOSAT is fast enough to be practical for data centers with hundreds of servers, with typical allocation times of a few seconds per VDC (however, in a minority of cases, MONOSAT did require tens or even hundreds of seconds for individual allocations). In many cases, MONOSAT was able to allocate more than twice as many VDCs as SecondNet on these data centers — a substantial improvement in data center utilization.

Comparison on commercial networks The above comparisons consider how MONOSAT compares to existing VDC allocation tools on several artificial (but representative) network topologies from the VDC literature. To address the question of whether there are actual real-world VDC applications where MONOSAT performs not only better than existing tools, but is also fast enough to be used in practice, we also considered a deployment of a standard Hadoop virtual cluster, on a set of actual data center topologies. We collaborated with the private cloud provider ZeroStack [215] to devise an optimal virtual Hadoop cluster to run Terasort [60]. Each Hadoop virtual network consists of a single master VM connected to 3–11 slave VMs. We consider 5 different sizes of VMs, ranging from 1 CPU and 1GB RAM, to 8 CPUs and 16GB of RAM; for our experiments, the slave VMs are selected randomly from this set, with the master VM selected randomly but always at least as large as the largest slave VM. The Hadoop master has tree connectivity with all slaves, with either a 1 or 2 Gbps network link between the master and each slave.

The physical data center topology was provided by another company, which requested to remain anonymous. This company uses a private cloud deployed across four data centers in two geographic availability zones (AZs): *us-west* and *us-middle*. Each data center contains between 280 and 1200 servers, spread across 1 to 4 clusters with 14 and 40 racks. Each server has 16 cores, 32 GB RAM, 20 Gbps network bandwidth (over two 10 Gbps links). The network in each data center has a leaf-spine topology, where all ToR switches connect to two distinct *aggregation switch*es over two 20 Gbps links each (a total of 4 links with 80 Gbps; two on each aggregation switch) and aggregation switches are interconnected with four 40 Gbps links each. For each cluster, there is a *gateway switch* with a 240 Gbps link connected to each aggregation switch. All data centers use equal-cost multi-path (ECMP) to take advantage of multiple paths.

A VDC is allocated inside one AZ: VMs in one VDC can be split across two clusters in an AZ, but not across two AZs. Table 6.10 shows VDC allocation results per AZ.

We applied SecondNet and MONOSAT in this setting, consecutively allocating random Hadoop master-slave VDCs of several sizes, ranging from 4 to 12 VMs, until no further allocations could be made. Note that, as with the previous experiment, Z3-AR is unable to run in this setting as it is restricted to tree-topology data centers.

In Figure 6.10 we show the results for the largest of these data centers (results for the smaller data centers were similar). As with the previous experiments on Tree, FatTree, and BCube topology data centers, although SecondNet is much faster than MONOSAT, MONOSAT's per-instance allocation time is typically just a few seconds, which is realistically useful for many practical applications. As with our previous experiments, MONOSAT is able to allocate many more VDCs than SecondNet - in these examples, allocating between 1.5 and 2 times as many total VDCs as SecondNet, across a range of data center and VDC sizes, including a commercial data center with more than 1000 servers.

MONOSAT is not only able to find many more allocations than SecondNet in this realistic setting, but MONOSAT's median allocation time, 1-30 seconds, shows that it can be practically useful in a real, commercial setting, for data centers and VDCs of this size. This provides strong evidence that MONOSAT can find practical use in realistic settings where large or bandwidth-hungry VDCs need to be allocated. It also demonstrates the practical advantage of a (fast) complete algorithm like MONOSAT over a much faster but incomplete algorithm like SecondNet: for bandwidth-heavy VDCs, even with arbitrary running time, SecondNet's VDCAlloc is unable to find the majority of the feasible allocations.

	VDC Instance Structure											
	1G-4 VMs		2G-4 VMs		1G-10 VMs		2G-10 VMs		1G-15 VMs		2G-15 VMs	
	#VDC	Time	#VDC	Time	#VDC	Time	#VDC	Time	#VDC	Time	#VDC	Time
US West 1: 2 clusters, 60 racks, 1200 servers with 32 cores each												
SecondNet	**2400**	0.6	1740	0.4	786	0.2	240	< 0.1	480	0.1	0	< 0.1
MonoSAT	2399	261.7	**2399**	261.7	**943**	244.1	**932**	265.7	**613**	554.5	**600**	1205.3
US West 2: 1 clusters, 14 racks, 280 servers with 32 cores each												
SecondNet	**560**	0.1	406	< 0.1	184	< 0.1	56	0.1	112	< 0.1	0	< 0.1
MonoSAT	**560**	14.5	**560**	14.2	**221**	11.3	**217**	11.3	**146**	14.9	**143**	44.7
US Mid 1: 4 clusters, 24 racks, 384 servers with 32 cores each												
SecondNet	**768**	0.2	553	0.1	244	< 0.1	73	0.1	191	< 0.1	0	< 0.1
MonoSAT	767	28.0	**765**	27.3	**303**	21.8	**301**	20.5	**200**	30.5	**191**	79.4
US Mid 2: 1 cluster, 40 racks, 800 servers with 32 cores each												
SecondNet	**1600**	0.3	1160	0.3	524	0.12	160	< 0.1	320	0.1	0	< 0.1
MonoSAT	1597	111.7	**1597**	114.9	**634**	82.8	**621**	97.9	**413**	263.7	**402**	373.3

Table 6.10: Total number of consecutive Hadoop VDCs allocated on data centers. Here we consider several variations of a virtual network deployment for a standard Hadoop cluster, on 4 real-world, full-scale commercial network topologies. *1G-4 VMs* stands for Hadoop virtual cluster consisting of 4 VMs (one master and 3 slaves as described Section 6.3.4 commercial networks subsection), where master connects with all slaves over 1 Gbps link. As before, SecondNet is much faster than MonoSAT (though MonoSAT is also fast enough for real-world usage, requiring typically < 1 second per allocation). However, as the virtual network becomes even moderately large, MonoSAT is able to allocate many more virtual machines while respecting end-to-end bandwidth constraints - often allocating several times as many machines as SecondNet, and in extreme cases, finding hundreds of allocations in cases where SecondNet cannot make any allocations at all. Similarly, keeping the virtual network the same size but doubling the bandwidth requirements of each virtual machine greatly decreases the allocations that SecondNet can make, while MonoSAT is much more robust to these more congested settings.

This reinforces our observations from our earlier experiments with artificial topologies: MonoSAT improves greatly on state-of-the-art VDC allocation, for bandwidth-constrained data centers with as many as 1000 servers.

6.4 Conclusion

In this chapter, we have presented comprehensive experimental results from three diverse fields: procedural content generation, circuit layout, and data center allocation. The experiments demonstrate that our graph theory, as implemented in the SMT solver MONOSAT using the techniques described in Chapter 4, can extend the state-of-the-art. Our results in particular highlight the scalability and effectiveness of our support for maximum flow predicates (which are key to the circuit layout and data center allocation encodings).

At the same time, these experiments also give a picture of some of the cases where MONOSAT does *not* perform as well as other approaches. For example, we saw that for reachability constraints, there are cases where MONOSAT is greatly more scalable, and also cases where it is greatly less scalable, than the ASP solver CLASP.

However, while there do exist cases where MONOSAT's graph theory may not outperform existing approaches, we have also shown that in many important cases — and across diverse fields — MONOSAT's graph theory does in fact attain state-of-the-art performance that greatly improves upon existing SAT, SMT, or ASP solvers.

Chapter 7

Monotonic Theory of Geometry

The next monotonic theory that we consider is a departure from the other theories we have explored. This is a theory of predicates concerning the convex hulls of finite, symbolic point sets (Figure 7.1). There are many common geometric properties of the convex hulls of point sets that monotonically increase (or decrease) as additional points are added to the set. For example, the area covered by the convex hull of a point set may increase (but cannot decrease) as points are added to the set.

Similarly, given a fixed query point q, adding additional points to the point set can cause the convex hull of that point set to grow large enough to contain q, but cannot cause a convex hull that previously contained q to no longer contain q. Similarly, given the convex hulls of two point sets, adding a point to either set can cause the two hulls to overlap, but cannot cause overlapping hulls to no longer overlap.

Many other common geometric properties of point sets are also monotonic (examples include the minimum distance between two point sets, the geometric span (i.e., the maximum diameter) of a point set, and the weight of the minimum steiner tree of a point set), but we restrict our attention to convex hulls in this chapter. Our current implementation is limited to 2 dimensional point sets; however, it should be straightforward to generalize our solver to 3 or more dimensions.

We implemented a theory solver supporting several predicates of convex hulls of finite point sets, using the SMMT framework, in our SMT solver MONOSAT (described in Appendix A). Our implementation operates over one or more

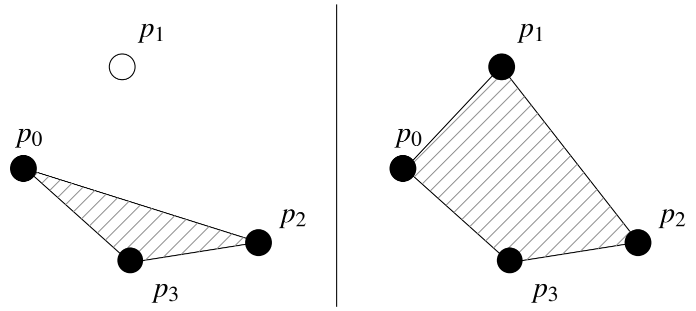


Figure 7.1: Left: Convex hull (shaded) of a symbolic point set $S \subseteq \{p_0, p_1, p_2, p_3\}$, with $S = \{p_0, p_2, p_3\}$. *Right*: Convex hull of $S = \{p_0, p_1, p_2, p_3\}$. Each point p_i has a fixed, constant position; the solver must chose which of these points to include in S. Many properties of the convex hulls of a point set S are monotonic with respect to S. For example, adding a point to S may increase the area of the convex hull, but cannot decrease it.

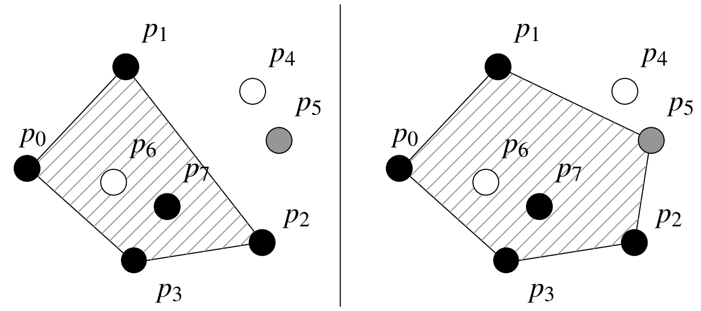


Figure 7.2: *Left:* Under-approximative convex hull H^- (shaded) of a symbolic point set $S \subseteq \{p_0 \dots p_7\}$, under the partial assignment $\mathcal{M} = \{(p_0 \in S), (p_1 \in S), (p_2 \in S), (p_3 \in S), (p_4 \notin S), (p_6 \notin S), (p_7 \in S)\}$, with p_5 unassigned (shown in grey). *Right:* Over-approximate convex hull H^+ of the same assignment.

symbolic sets of points $S_0, S_1, \ldots$, with their elements represented in Boolean form as a set of element atoms $(p_0 \in S_0), (p_1 \in S_0), \ldots$. Given a partial assignment $\mathcal{M} = \{(p_0 \in S), (p_1 \in S), (p_2 \in S), (p_3 \in S), (p_4 \notin S), (p_6 \notin S), (p_7 \in S)\}$ to the element atoms, the theory solver first forms concrete over- and under-approximations of each point set, S^+, S^-, where S^- contains only the elements for which $(p_i \in S_i) \in \mathcal{M}$, and S_i^+ all elements in S_i^- along with any points that are unassigned in $\mathcal{M}$. For each set, the theory solver then computes an under-approximative convex hull, H_i^-, from S_i^-, and an over-approximative convex hull, H_i^+, from S_i^+ (see Figure 7.2). Because S_i^- and S_i^+ are both concrete sets (containing tuples representing 2D points), we can use any standard convex hull algorithm to compute H_i^- and H_i^+; we found that Andrew's monotone chain algorithm [7] worked well in practice.

Having computed H_i^- and H_i^+ for each set S_i, we then iterate through each predicate atom $p(S_i)$ in the theory and individually compute whether they hold in H_i^-, or fail to hold in H_i^+. These individual checks are computed for each predicate as described in Section 7.1.

One important consideration of these geometric properties is that numerical accuracy plays a greater role than in the graph properties we considered above. So long as the coordinates of the points themselves are rationals, the area of the convex hull, along with point containment and intersection queries can be computed precisely using arbitrary precision rational arithmetic by computing determinants (see e.g. chapters 3.1.6 and 11.5 of [91] for a discussion of using determinants for precise computation of geometric properties).

Our implementation of the geometric theory solver is very close to the graph solver we described in Chapter 5, computing concrete under- and over-approximations H^-, H^+ of the convex hull of each point set in the same manner as we did for G^- and G^+. We also include two additional improvements. First, after computing each H_i^-, H_i^+, we compute axis-aligned bounding boxes for each hull ($bound_i^-, bound_i^+$). These bounding boxes are very inexpensive to compute, and allow us to cheaply eliminate many collision detections with the underlying convex hulls (this is especially important when using expensive arbitrary precision arithmetic).

Secondly, once an intersection (or point containment) is detected, we find a small (but not necessarily minimal) set of points which are sufficient to produce that collision. For example, when a point is found to be contained in a convex hull, we can find three points from that hull that together form a triangle that contains that point. So long as the points composing that triangle remain in the hull, even if other points are removed from the hull the point will remain contained. These points can typically be arranged to be computed as a side effect of collision detection, and their presence in S^- of S^+ can be checked very cheaply, allowing us to skip many future collision checks while those points remain in the relevant set. This is particularly useful for the geometric properties we consider here, as a) a fixed, small number of points typically constitute a proof of containment (usually 2, 3 or 4), in contrast to potentially very long paths in the graph theory solver of Chapter 5, and b) the geometric properties we are computing may be much more expensive than the graph properties being computed in the previous section.

Many techniques exist for speeding up the computation of dynamic geometric properties, especially with regards to collision detection, and we have only implemented the most basic of these (bounding boxes); a more efficient implementation could make use of more efficient data structures (such as hierarchical bounding volumes or trapezoidal maps [69]) to greatly speed up or obviate many of the computations in our solver. As before, because these computations are performed on the concrete under- and over-approximation point sets, standard algorithms or off-the-shelf collision detection libraries may be used.

7.1 Geometric Predicates of Convex Hulls

In this section, we describe the predicates of convex hulls supported by our geometry theory (each of which is a finite, monotonic predicate). For each predicate, we describe the algorithms used to check the predicates on H^- and H^+ (as computed above), and to perform conflict analysis when a conflict is detected by theory propagation.

7.1.1 Areas of Convex Hulls

Given a set of points S, the area of the convex hull of that set of points (shaded area of Figure 7.1) can increase as more points are added to S, but cannot decrease.

This predicate allows us to constrain the area of the convex hull of a point set. In principle, this predicate could also be set up as a monotonic function (returning the area of the hull as a rational-value), however as our implementation does not yet support linear arithmetic, we will treat the area constraints as predicates comparing the area to fixed, constant values (expressed as arbitrary precision rationals).

Monotonic Predicate: $\mathit{hullArea}_{\geq x}(S)$, $\mathit{hullArea}_{>x}(S)$, with S a finite set $S \subseteq S'$, true iff the convex hull of the points in S has an area $\geq$ (resp. $>$) than x (where x is a constant). This predicate is positive monotonic with respect to the set S.

Algorithm: Initially, compute $\mathit{area}(\mathit{bound}^-), \mathit{area}(\mathit{bound}^+)$. If $\mathit{area}(\mathit{bound}^-) > x$, compute $\mathit{area}(H^-)$; if $\mathit{area}(\mathit{bound}^+) < x$, compute $\mathit{area}(H^+)$. The areas can be computed explicitly, using arbitrary precision rational arithmetic.

Conflict set for $\mathit{hullArea}_{\geq x}(S)$**:** The area of H^- is greater or equal to x. Let $p_0, p_1, \ldots$ be the points of S^- that form the vertices of the under-approximate hull H^- (e.g., points p_0, p_1, p_2, p_3 in Figure 7.2), with $\mathit{area}(H^-) \geq x$. Then at least one of the points must be disabled for the area of the hull to decrease below x. The conflict set is $\{(p_0 \in S), (p_1 \in S), \ldots, \neg \mathit{hullArea}_{\geq x}(S)\}$.

Conflict set for $\neg \mathit{hullArea}_{\geq x}(S)$**:** The area of H^+ is less than x; then at least one point $p_i \notin S^+$ that is not contained in H^+ must be added to the point set to increase the area of H^+ (e.g., only point p_4 in Figure 7.2). Let points $p_0, p_1, \ldots$ be the points of S' not contained in H^+. The conflict set is $\{(p_0 \notin S^+), (p_1 \notin S^+), \ldots, \mathit{hullArea}_{\geq x}(S)\}$, where $p_0, p_1, \ldots$ are the (possibly empty) set of points $p_i \notin S^+$.

7.1.2 Point Containment for Convex Hulls

Given a set of points S, and a fixed point q (Figure 7.3), adding a point to S can cause the convex hull of S to grow to contain p, but cannot cause a convex hull that previously contained q to no longer contain q.

Note that this predicate is monotonic with respect to the set of elements in S, but that it is not monotonic with respect to translating the position of the point q

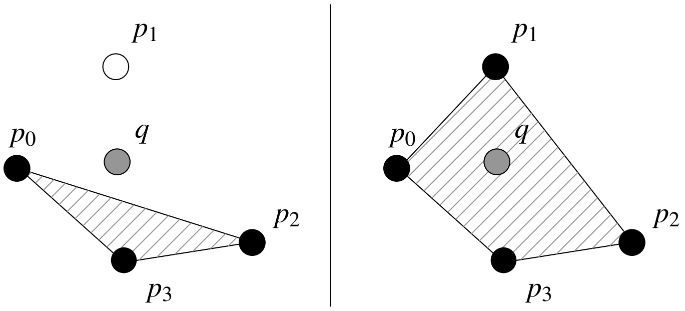


Figure 7.3: *Left:* The convex hull (shaded) of a symbolic point set $S \subseteq \{p_0, p_1, p_2, p_3\}$, along with a query point q. *Right:* The same as the left, but with point p_1 added to set S. After adding this point, the convex hull of S contains point q.

(which is why q must be a constant). The points that can be contained in S, along with the query point q, each have constant (2D) positions.

There are a few different reasonable definitions of point containment (all monotonic), depending on whether the convex hull is treated as a closed set (including both the area inside the hull and also the edges of the hull) or an open set (only the interior of the hull is included). We consider the closed variant here.

Monotonic Predicate: $hullContains_q(S)$, true iff the convex hull of the 2D points in S contains the (fixed point) q.

Algorithm: First, check whether q is contained in the under-approximative bounding box $bound^-$. If it is, then check if q is contained in H^-. We use the PNPOLY [95] point inclusion test to perform this check, using arbitrary precision rational arithmetic, which takes time linear in the number of vertices of H^-. In the same way, only if q is contained in $bound^+$, check if q is contained in H^+ using PNPOLY.

Conflict set for $hullContains_q(S)$**:** Convex hull H^- contains p. Let p_0, p_1, p_2 be three points from H^- that form a triangle containing p (such a triangle must exist, as H^- contains p, and we can triangulate H^-, so one of those triangles must contain p; this follows from Carathodory's theorem for convex hulls [51]). So long as those three points are enabled in S, H^- must contain

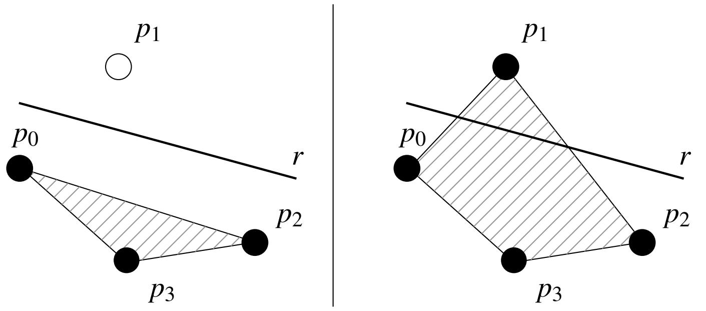


Figure 7.4: *Left:* The convex hull (shaded) of a symbolic point set $S \subseteq \{p_0, p_1, p_2, p_3\}$, along with a line-segment r. *Right:* The same as the left, but with point p_1 added to set S. After adding this point, the convex hull of S intersects line-segment r.

them, and as they contain p, p must be contained in H^-. The conflict set is $\{(p_0 \in S), (p_1 \in S), (p_2 \in S), \neg hullContains_q(S)\}$.

Conflict set for $\neg hullContains_q(S)$**:** p is outside of H^+. If H^+ is empty, then the conflict set is the default monotonic conflict set. Otherwise, we will employ the separating axis theorem to produce a conflict set. The separating axis theorem [91] states that given any two convex volumes, either those volumes intersect, or there exists a separating axis on which the volumes projections do not overlap. As we gave two non-intersecting convex hulls, by the separating axis theorem, there exists a separating axis between H^+ and p. Let $p_0, p_1, \ldots$ be the (disabled) points of S whose projection onto that axis is $\geq$ the projection of p onto that axis[1]. At least one of those points must be enabled in S in order for H^+ to grow to contain p. The conflict set is $\{(p_0 \notin S), (p_1 \notin S), \ldots, hullContains_q(S)\}$.

7.1.3 Line-Segment Intersection for Convex Hulls

Given a set of points S, and a fixed line-segment r (Figure 7.4), adding a point to S can cause the convex hull of S to intersect r, but cannot cause a convex hull that previously intersected r to no longer intersect r.

This predicate directly generalizes the previous predicate (point containment), and we will utilize some of the methods described above in computing it.

Monotonic Predicate: $\mathit{hullIntersects}_r(S)$, true iff the convex hull of the 2D points in S intersects the (fixed) line-segment r.

Algorithm: First, check whether line-segment r intersects bound^-. If it does, check if r intersects H^-. If H^- is empty, is a point, or is itself a line-segment, this is trivial (and can be checked precisely in arbitrary precision arithmetic by computing cross products following [110]). Otherwise, we check if either end-point of r is contained in H^-, using PNPOLY as above for point containment. If neither end-point is contained, we check whether the line-segment intersects H^-, by testing each edge of H^- for intersection with r (as before, by computing cross-products). If r does not intersect the under-approximation, repeat the above on bound^+ and H^+.

Conflict set for $\mathit{hullIntersects}_r(S)$: Convex hull H^- intersects line-segment r. If either end-point or r was contained in H^-, then proceed as for the point containment predicate. Otherwise, the line-segment r intersects with at least one edge (p_i, p_j) of H^-. Conflict set is $\{(p_i \in S), (p_j \in S), \neg \mathit{hullIntersects}_r(S)\}$.

Conflict set for $\neg \mathit{hullIntersects}_r(S)$: r is outside of H^+. If H^+ is empty, then the conflict set is the naïve monotonic conflict set. Otherwise, by the separating axis theorem, there exists a separating axis between H^+ and line-segment r. Let $p_0, p_1, \ldots$ be the (disabled) points of S whose projection onto that axis is $\geq$ the projection of the nearest endpoint of r onto that axis. At least one of

[1] We will make use of the separating axis theorem several times in this chapter. The standard presentation of the separating axis theorem involves normalizing the separating axis (which we would not be able to do using rational arithmetic). This normalization is required if one wishes to compute the minimum distance between the projected point sets; however, if we are only interested in *comparing* distances (and testing collisions), we can skip the normalization step, allowing the separating axis to be found and applied using only precise rational arithmetic.

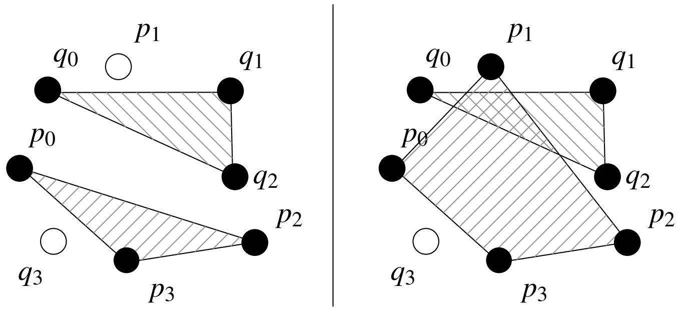


Figure 7.5: *Left:* Two symbolic point sets $S_0 \subseteq \{p_0, p_1, p_2, p_3\}$ and $S_1 \subseteq \{q_0, q_1, q_2, q_3\}$, and corresponding convex hulls (shaded areas). Hulls are shown for $S_0 = \{p_0, p_2, p_3\}$, $S_1 = \{q_0, q_1, q_2\}$. These convex hulls do not intersect. *Right:* Same as the left, but showing hulls after adding point p_1 into S_0. With this additional point in S_0, the convex hulls of S_0, S_1 now intersect.

those points must be enabled in S in order for H^+ to grow to contain p. The conflict set is $\{(p_0 \notin S), (p_1 \notin S), \ldots, hullIntersects_r(S)\}$.

7.1.4 Intersection of Convex Hulls

The above predicates can be considered a special-case of a more general predicate that tests the intersections of the convex hulls of two point sets: Given two sets of points, S_0 and S_1, with two corresponding convex hulls $hull_0$ and $hull_1$ (see Figure 7.5), adding a point to either point set can cause the corresponding hull to grow such that the two hulls intersect; however, if the two hulls already intersect, adding additional points to either set cannot cause previously intersecting hulls to no longer intersect.

This convex hull intersection predicate can be used to simulate the previous predicates ($hullContains_q(S)$ and $hullIntersects_r(S)$), by asserting all of the points of one of the two point sets to constant assignments. For efficiency, we handle point containment, and intersection with fixed line segments or polygons, as special cases in our implementation.

As with the other above predicates, there are variations of this predicate depending on whether or not convex hulls with overlapping edges or vertices are considered to intersect (that is, whether the convex hulls are open or closed sets).

We will consider only the case where both hulls are closed (i.e., overlapping edges or vertices are considered to intersect).

Monotonic Predicate: $hullsIntersect(S_0, S_1)$, true iff the convex hull of the points in S_0 intersects the convex hull of the points of S_1.

Algorithm: If the bounding box for convex hull H_0 intersects the bounding box for H_1, then there are two possible cases to check for:

1. 1: A vertex of one hull is contained in the other, or
2. 2: an edge of H_0 intersects an edge of H_1.

If neither of the above cases holds, then H_0 and H_1 do not intersect.

Each of the above cases can be tested in quadratic time (in the number of vertices of H_0 and H_1), using arbitrary precision arithmetic. In our implementation, we use PNPOLY, as described above, to test vertex containment, and test for pair-wise edge intersection using cross products.

Conflict set for $hullsIntersect(S_0, S_1)$**:** H_0^- intersects H_1^-. There are two (not mutually exclusive) cases to consider:

1. A vertex of one hull is contained within the other hull. Let the point be p_a; let the hull containing it be H_b^-. Then (as argued above) there must exist three vertices p_{b1}, p_{b2}, p_{b3} of H_b^- that form a triangle containing p_a. So long as those three points and p_a are enabled, the two hulls will overlap. Conflict set is $\{(p_a \in S_a), (p_{b1} \in S_b), (p_{b2} \in S_b), (p_{b3} \in S_b), \neg hullsIntersect(S_0, S_1)\}$.
2. An edge of H_0^- intersects an edge of H_1^-. Let p_{1a}, p_{1b} be points of H_0^-, and p_{2a}, p_{2b} points of H_1^-, such that line segments $\overline{p_{1a}, p_{1b}}$ and $\overline{p_{2a}, p_{2b}}$ intersect. So long as these points are enabled, the hulls of the two point sets must overlap. Conflict set is $\{(p_{0a} \in S_0), (p_{0b} \in S_0), (p_{1a} \in S_1), (p_{1b} \in S_1), \neg hullsIntersect(S_0, S_1)\}$.

Conflict set for $\neg hullsIntersect(S_0, S_1)$**:** H_0^+ do not intersect H_1^+. In this case, there must exist a separating axis (Figure 7.6) between H_0^+ and H_1^+. (Such

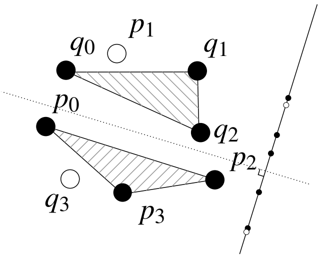


Figure 7.6: *A separating axis between two convex hulls.* Between any two disjoint convex polygons, there must exist a separating line (dotted line) parallel to one edge (in this case, parallel to edge (p_0, p_2)), and a separating axis (solid line) normal to that edge. Small circles show the positions of each point, projected onto the separating axis.

an axis can be discovered as a side effect of computing the cross products of each edge in step 2 above.) Project all disabled points of S_0 and S_1 onto that axis (white dots in Figure 7.6). Assume (without loss of generality) that the maximum projected point of H_1^+ is less than the minimum projected point of H_1^+. Let $p_{0a}, p_{0b}, \ldots$ be the disabled points of S_0 whose projections are on the far side of the maximum projected point of H_1^+. Let $p_{1a}, p_{1b}, \ldots$ be the disabled points of S_1 whose projections are near side of the minimum projected point of H_1^+. At least one of these disabled points must be enabled, or this axis will continue to separate the two hulls. The conflict set is $\{(p_{0a} \notin S_0), (p_{0b} \notin S_0), \ldots, (p_{1a} \notin S_1), (p_{1b} \notin S_1), \ldots, hullsIntersect(S_0, S_1)\}$.

Many other common geometric properties of point sets are also monotonic (examples include the minimum distance between two point sets, the geometric span (i.e., the maximum diameter) of a point set, and the weight of the minimum steiner tree of a point set), but we restrict our attention to convex hulls in this chapter. Our current implementation is limited to 2-dimensional point sets; however, it should be straightforward to generalize our solver to 3 or more dimensions.

7.2 Applications

Here we consider the problem of synthesizing art galleries, subject to constraints. This is a geometric synthesis problem, and we will use it to demonstrate the convex hull constraints introduced in this chapter. The Art Gallery problem is a classic NP-Hard problem [55, 124, 141], in which (in the decision version) one must determine whether it is possible to surveil the entire floor-plan of a multi-room building with a limited set of fixed cameras. There are many variations of this problem, and we will consider a common version in which cameras are restricted to being placed on vertices, and only every vertex of the room must be seen by a camera (as opposed to the floor or wall spaces between them). This variant is NP-Complete, by reduction from the dominating set problem [141].

Here, we introduce a related problem: Art Gallery Synthesis. Art Gallery Synthesis is the problem of designing an 'art gallery' (a non-intersecting set of simple polygons without holes) that can be completely surveilled by at most a given number of cameras, subject to some other constraints on the allowable design of the art gallery. The additional constraints may be aesthetic or logistic constraints (for example, that the area of the floor plan be within a certain range, or that there be a certain amount of space on the walls). There are many circumstances where one might want to design a building or room that can be guarded from a small number of sightlines. Obvious applications include aspects of video game design, but also prisons and actual art galleries.

Without constraints on the allowable geometry, the solution to the art gallery synthesis problem is trivial, as any convex polygon can be completely guarded by a single camera. For this reason, the choice of constraints has a great impact on the practical difficulty of solving this problem. There are many ways that one could constrain the art gallery synthesis problem; below, we will describe a set of constraints that are intended to exhibit the strengths of our convex hull theory solver, while also producing visually interesting and complex solutions.

We consider the following constrained art gallery synthesis problem: Given a box containing a fixed set of randomly placed 2-dimensional points, we must find N non-overlapping convex polygons with vertices selected from those points, such that a) the area of each polygon is greater than some fixed constant (to prevent lines or very small slivers from being created), b) the polygons may meet at an edge, but may not meet at just one vertex (to prevent forming wall segments of infinitesimal thickness), and c) all vertices of all the polygons, and all 4 corners of the room, can be seen by a set of at most M cameras (placed at those vertices). Figure 7.7 shows

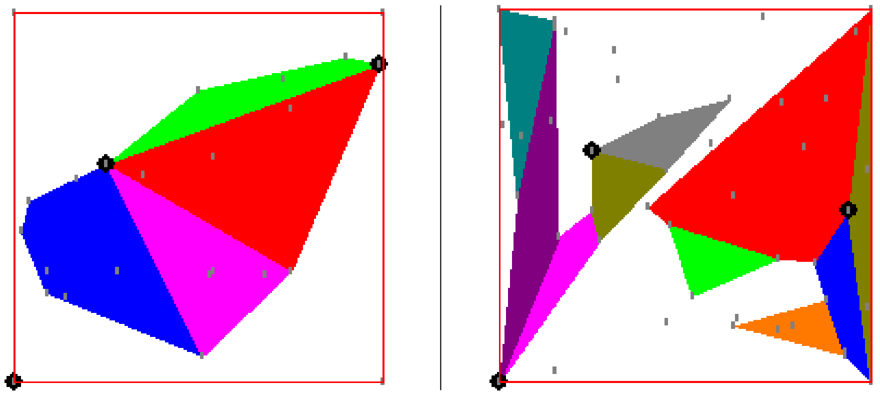


Figure 7.7: Two artificial 'art galleries' found by MonoSAT. Cameras are large black circles; potential vertices are gray dots. Each convex polygon is the convex hull of a subset of those gray dots, selected by the solver. Notice how the cameras have been placed such that the vertices of all hulls (and the corners of the room) can been seen, some along very tight angles, including one vertex completely embedded in three adjoining polygons. Notice also that some vertices can only be seen from one 'side', and that some edges cannot be fully observed by cameras (as only vertices are required to be observable in this variant of the problem).

Art Gallery Synthesis	MonoSAT	Z3
10 points, 3 polygons, $\leq$3 cameras	2s	7s
20 points, 4 polygons, $\leq$4 cameras	36s	433s
30 points, 5 polygons, $\leq$5 cameras	187s	$>$ 3600s
40 points, 6 polygons, $\leq$6 cameras	645s	$>$ 3600s
50 points, 7 polygons, $\leq$7 cameras	3531s	$>$ 3600s
20 points, 10 polygons, $\leq$5 cameras	3142s	Timeout
20 points, 10 polygons, $\leq$3 cameras	16242s	Memout

Table 7.1: Art gallery synthesis results.

two example solutions to these constraints, found by MonoSAT.

In Table 7.1, we compare MonoSAT to an encoding of the above constraints into the theory of linear arithmetic (as solved by Z3, version 4.3.1). Unfortunately, the encoding into linear arithmetic is very expensive, using a cubic number of comparisons to find the points making up the hull of each convex polygon. We describe the encoding in detail in Appendix C.3.

For each instance, we list the number of randomly placed points from which the vertices of the polygons can be chosen, the number of convex polygons to be placed, and the maximum number of cameras to be used to observe all the vertices of the room (with cameras constrained to be placed only on vertices of the placed polygons, or of the bounding box). These experiments were conducted on an Intel i7-2600K CPU, at 3.4 GHz (8MB L3 cache), limited to 10 hours of runtime and 16 GB of RAM.

The version of art gallery synthesis that we have considered here is an artificial one that is likely far removed from a real-world application. Nonetheless, these examples show that, as compared to a straightforward linear arithmetic encoding, the geometric predicates supported by MONOSAT can be greatly more scalable, and show promise for more realistic applications.

Chapter 8

Monotonic Theory of CTL

Computation Tree Logic (CTL) synthesis [56] is a long-standing problem with applications to synthesizing synchronization protocols and concurrent programs. We show how to formulate CTL model checking as a monotonic theory, enabling us to use the SMMT framework of Chapter 4 to build a Satisfiability Modulo CTL solver (implemented as part of our solver MONOSAT, described in Appendix A). This yields a powerful procedure for CTL synthesis, which is not only faster than previous techniques from the literature, but also scales to larger and more difficult formulas. Moreover, our approach is efficient at producing *minimal* Kripke structures on common CTL synthesis benchmarks.

Computation tree logic is widely used in the context of model checking, where a CTL formula specifying a temporal property, such as safety or liveness, is checked for validity in a program or algorithm (represented by a Kripke structure). Both the branching time logic CTL and its application to model checking were first proposed by Clarke and Emerson [56]. In that work, they also introduced a decision procedure for CTL satisfiability, which they applied to the synthesis of *synchronization skeletons*, abstractions of concurrent programs which are notoriously difficult to construct manually. Though CTL model checking has found wide success, there have been fewer advances in the field of CTL synthesis due to its high complexity.

In CTL synthesis, a system is specified by a CTL formula, and the goal is to find a model of the formula — a Kripke structure in the form of a transition system

$\texttt{AG}(\texttt{EF}(q \land \neg p))$

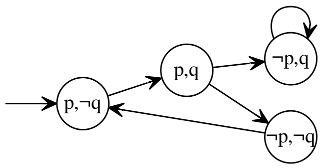


Figure 8.1: An example CTL formula (left), and a Kripke structure (right) that is a model for that formula. The CTL formula can be paraphrased as asserting that *"On all paths starting from the initial state (A) in all states on those paths (G) there must exist a path starting from that state (E) which contains at least one state (F) in which q holds and p does not."* The Kripke structure consists of a finite set of states connected with unlabelled, directed edges, and a finite set of state properties. One state is distinguished as the initial state in which the CTL formula will be evaluated, and each state is labelled with a truth value for each atomic proposition $\{p,q\}$ in the formula.

in which states are annotated with sets of atomic propositions (we will refer to these propositions as *state properties*). The most common motivation for CTL synthesis remains the synthesis of synchronization for concurrent programs, such as mutual exclusion protocols. In this setting, the Kripke structure is interpreted as a global state machine in which each global state contains every process's internal local state. The CTL specification in this setting consists of both structural intra-process constraints on local structures, and inter-process behavioral constraints on the global structure (for instance, starvation freedom). If a Kripke structure is found which satisfies the CTL specification, then one can derive from it the guarded commands that make up the corresponding synchronization skeleton [13, 56].

We introduce a theory of CTL model checking, supporting a predicate $Model(\phi)$, which evaluates to TRUE iff an associated Kripke structure $K = (T,P)$, with transition system T and state property mapping P, is a model for the CTL formula ϕ. We will then show that this predicate can be used to perform bounded CTL synthesis, by specifying a space of possible Kripke structures of bounded size and solving the resulting formula.

Due to the CTL small model property [87], in principle a bounded CTL-SAT procedure yields a complete decision procedure for unbounded CTL-SAT, but in practice, neither bounded approaches, nor classical tableau approaches, have been scalable enough for completeness to be a practical concern. Rather, our approach (like similar constraint-solver based techniques for CTL [68, 121] and LTL [132, 178]) is appropriate for the case where a formula is expected to be satisfiable by a Kripke structure with a modest number of states (~ 100). Nevertheless, we will show that our approach solves larger and more complex satisfiable CTL formulas, including ones with a larger numbers of states, much faster than existing bounded and unbounded synthesis techniques. This makes our approach particularly appropriate for CTL synthesis.

In addition to being more efficient than existing techniques, our approach is also capable of synthesizing minimal models. As we will discuss below, previous CTL synthesis approaches were either incapable of finding minimal models [12, 56], or could not do so with comparable scalability to our technique [68, 121].

We begin with a review of related work in Section 8.1. In Sections 8.2 and 8.3, we show how to apply the SMMT framework to a theory of CTL model checking. Sections 8.4 and 8.5 explain the most important implementation details and optimizations. In Section 8.6, we provide experimental comparisons to state-of-the-art techniques showing that our SMMT-approach to CTL synthesis can find solutions to larger and more complex CTL formulas than comparable techniques in two families of CTL synthesis benchmarks: one derived from mutual exclusion protocols, and the other derived from readers-writers protocols. Further, our approach does so without the limitations and extra expert knowledge that previous approaches require.

8.1 Background

The original 1981 Clarke and Emerson paper introducing CTL synthesis [56] proposed a tableau-based synthesis algorithm, and used this algorithm to construct a 2-process mutex in which each process was guaranteed mutually exclusive access to the critical section, with starvation freedom.

Subsequently, although there has been steady progress on the general CTL synthesis problem, the most dramatic gains have been with techniques that are structurally-constrained, taking a CTL formula along with some additional ‘structural' information about the desired Kripke structure, not specified in CTL, which is then leveraged to achieve greater scalability than generic CTL synthesis techniques. For example, in 1998, Attie and Emerson [11, 12] introduced a CTL synthesis technique for the case where the Kripke structure is known to be composed of multiple similar communicating processes. They used this technique to synthesize a Kripke structure for a specially constructed 2-process version of the CTL formula (a 'pair-program') in such a way that the produced Kripke structure could be safely generalized into an N-process solution. This allowed them to produce a synchronization skeleton for a mutex with 1000 or more processes, far larger than other techniques. However, while this process scales very well, only certain CTL properties can be guaranteed to be preserved in the resulting Kripke structure, and in general the Kripke structure produced this way may be much larger than the minimal solution to the instance. In particular, EX and AX properties are not preserved in this process [11].

Additionally, the similar-process synthesis techniques of Attie and Emerson rely on a generic CTL synthesis method to synthesize these pair-programs. As such, improvements to the scalability or expressiveness of generic CTL synthesis methods can be directly applied to improving this pair-program synthesis technique. Their use of the synthesis method from [56] yields an initially large Kripke structure that they minimize in an intermediate step. We note that our approach is particularly suited for synthesizing such pair-programs, not merely for performance reasons, but also because it is able to synthesize minimal models directly.

On the topic of finding minimal models, Bustan and Grumberg [49] introduced a technique for minimizing Kripke structures. However, the minimal models that our technique produces can in general be smaller than what can be achieved by starting with a large Kripke structure and subsequently minimizing it. This is because minimization techniques which are applied on an existing Kripke structure *after* its synthesis only yield a structure minimal with respect to equivalent structures (for some definition of equivalence, e.g. strong or weak bisimulation). This does not necessarily result in a structure that is the overall minimal model of the original CTL formula. For this reason, techniques supporting the direct synthesis of minimal models, such as ours, have an advantage over post-synthesis minimization techniques.

In 2005, Heymans et al. [121] introduced a novel, constraint-based approach to the general CTL synthesis problem. They created an extension of answer set programming (ASP) that they called ‘preferential ASP’ and used it to generate a 2-process mutex with the added property of being ‘maximally parallel’, meaning that each state has a (locally) maximal number of outgoing transitions (without violating the CTL specification). They argued that this formalized a property that was implicit in the heuristics of the original 1981 CTL synthesis algorithm, and that it could result in Kripke structures that were easier to implement as efficient concurrent programs. As the formulation in their paper does not require additional structural constraints (though it can support them), it is a general CTL synthesis method. Furthermore, being a constraint-based method, one can flexibly add structural or other constraints to guide the synthesis. However, the scalability of their method was poor.

Subsequently, high performance ASP solvers [104] built on techniques from Boolean satisfiability solvers were introduced, allowing ASP solvers to solve much larger and much more difficult ASP formulas. In 2012, De Angelis, Pettorossi, and Proietti [68] showed that (unextended) ASP solvers could also be used to perform efficient bounded CTL synthesis, allowing them to use the high performance ASP solver Clasp [104]. Similar to [12], they introduced a formulation for doing CTL synthesis via ASP in the case where the desired Kripke structure is composed of multiple similar processes. Using this approach, they synthesized 2-process and 3-process mutexes with properties at least as strong as the original CTL specification from [12]. The work we introduce in this chapter is also a constraint-solver based, bounded CTL-synthesis technique. However, in Section 8.6, we will show that our approach scales to larger and more complex specifications than previous work, while simultaneously avoiding the limitations that prevent those approaches from finding minimal models.

Our focus in this section has been on this history of the development of CTL synthesis techniques; for a gentle introduction to CTL semantics (and CTL model checking), we refer readers to [128].

8.2 CTL Operators as Monotonic Functions

A Kripke structure is composed of a finite state transition system (T) over states S, and a set of Boolean state properties (P), such that each state property $p \in P$ has a truth value $p(s)$ in every state $s \in S$. One state of the Kripke structure is uniquely identified as the starting state — below, we will assume that the starting state is always state s_0 unless otherwise stated. A Kripke structure $K = (T, P)$ is said to be a *model* of a CTL formula ϕ if ϕ evaluates to TRUE in the starting state of K.

The grammar of a CTL formula ϕ in existential normal form (ENF) is typically defined recursively as

$$\phi ::= \text{TRUE}|a|\neg\phi|\phi \wedge \phi|EX\phi|EG\phi|E(\phi U \phi)$$

with a being a state property.[1]

However, we will consider a slightly different presentation of CTL that is more convenient to represent as a theory of CTL model checking. We represent Kripke structures as a tuple (T, P), with T a transition system and P a set of state property vectors. As in the graph theory in Chapter 5, we represent the transition system T as a set of potential transitions $T \subseteq S \times S$, and introduce, for each transition t in T, a theory atom $(t \in T)$, such that the transition t is enabled in T if and only if $(t \in T)$ is assigned TRUE. We will also introduce, for each property vector $p \in P$ and each state $s \in S$, a theory atom $(p(s))$, TRUE iff property p holds in state s. Together, these transition atoms and property atoms define a bounded Kripke structure, $K = (T, P)$. P is the set of all state properties in the Kripke structure; an individual state property $p \in P$ can also be represented as a vector of Booleans $A = [p(s_1), p(s_2), \ldots p(s_k)]$ for the k states of the system. This formulation of CTL model checking as a predicate over a symbolic Kripke structure with a fixed set of states and a variable set of transitions and property assignments is similar to the ones introduced in [68, 121].

[1] CTL formulas are also often expressed over a larger set of operators: $AG, EG, AF, EF, AX, EX, EU, AU$, along with the standard propositional connectives. However, it is sufficient to consider the smaller set of existentially quantified CTL operators EX, EG and EU, along with the propositional operators $(\neg, \wedge)$, and TRUE, which are known to form an adequate set. Any CTL formula can be efficiently converted into a logically equivalent existential normal form (ENF) in terms of these operators, linear in the size of the original formula [147].

We introduce for each unary CTL operator op an evaluation function $op(T,A)$, taking as arguments a set of edges specifying the transition system, T, and a 1-dimensional vector of Booleans, A, of size $|S|$ which indicates the truth value of property a in each state. Element s of A, written $A[s]$, is interpreted as representing the truth value of state property a in state s (we assume that the states are uniquely addressable in such a way that they can be used as indices into this property vector). For each of the binary operators EU and $\wedge$, we introduce an evaluation function $op(T,A,B)$, which takes a single transition system as above, and two state property vectors, A and B, representing the truth value of the arguments of op in each state.

Each evaluation function returns a vector of Booleans C, of size $|S|$, representing a fresh state property storing, for each state s, the truth value of evaluating CTL operator op from initial state s in the Kripke structure $K = (T, \{A\})$ (or $K = (T, \{A,B\})$ if op takes two arguments). This is a standard interpretation of CTL (and how explicit-state CTL model checking is often implemented), and we refer to the literature for common ways to compute each operator (see, e.g., [128]).

We support the following CTL functions (with T a transition system, and A,B,C vectors of Booleans representing state properties):

- $\neg(T,A) \mapsto C$
- $\wedge(T,A,B) \mapsto C$
- $\texttt{EX}(T,A) \mapsto C$
- $\texttt{EG}(T,A) \mapsto C$
- $\texttt{EU}(T,A,B) \mapsto C$

Notice that there is no operator for specifying state properties; rather, a state property is specified in the formula directly by passing a vector of state property atoms as an argument of an operator. For example, consider the following CTL formula in ENF form:

$$EG(\neg a \wedge EX(b))$$

This formula has two state properties (‘a’ and ‘b’). Given a transition system T, this formula can be encoded into our theory of CTL model checking as

$$EG(T, \wedge(T, \neg(T,a), EX(T,b)))$$

where a,b are vectors of Booleans representing the truth values of the state properties a and b at each state in the Kripke structure.[2]

Before describing our CTL theory solver, we will briefly establish that each CTL operator as defined above is monotonic. First, we show that the operators $\texttt{EX}(T,A)$, $\texttt{EG}(T,A)$ or $\texttt{EU}(T,A,B)$ are each monotonic with respect to the state property vectors A and B. Let $solve_s(\phi(T,A))$ be a predicate that denotes whether or not the formula ϕ holds in state s of the Kripke structure determined by the vector of Booleans T (transitions) and A (state properties).

Lemma 8.2.1. *$solve_s(\texttt{EX}(T,A))$, $solve_s(\texttt{EG}(T,A))$, and $solve_s(\texttt{EU}(T,A,B))$ are each positive monotonic in both the transition system T and state property A (and state property B, for* EU*).*

Proof. Take any T,A,B that determine a structure K for which the predicate holds. Let K' be a structure determined by some T',A',B' such that K' has the same states, state properties and transitions as K, except for one transition that is enabled in K' but not in K, or one state property which holds in K' but not in K. Formally, there is exactly one argument in either T', A', or B' that is 0 in T (or A or B respectively) and 1 in T' (or A' or B' respectively). Then either (a) one of the states satisfies one of the atomic propositions in K', but not in K, or (b) there is a transition in K', but not in K.

We assume $solve_s(\phi(T,A))$ holds (for EU, we assume that $solve_s(\phi(T,A,B))$ holds). Then, there must exist a witnessing infinite sequence starting from s in K. If (b), the exact same sequence must exist in K', since it has a superset of the transitions in K. Thus we can conclude $solve_s(\phi(T',A'))$ holds (respectively,

[2] Notice that the transition system T has to be passed as an argument to each function. We can ask whether a formula $EG(T_1, EX(T_2,a))$, with $T_1 \neq T_2$, is well-formed? In fact T_1 does not need to equal T_2 in order to have meaningful semantics, however, if the transition systems are not the same, then the semantics of the formula will not match the expected semantics of CTL. We will assume that all formulas of interest have the same transition system passed as each argument of the formula.

$solve_s(\phi(T',A',B'))$ holds). If (a), then the sequence will only differ in at most one state, where p holds instead of $\neg p$ (or q instead of $\neg q$). We note that for each of the three CTL operators, this sequence will be a witness for K', if the original sequence was a witness for K. Thus, $solve_s(\phi(T',A'))$ holds as well (respectively, $solve_s(\phi(T',A',B'))$ holds). □

It is easy to see that $\wedge$ and $\vee$ are positive monotonic in the same way, and $\neg$ is negative monotonic. Excluding negation, then, all the CTL operators needed to express formulas in ENF have positive monotonic $solve_s$ predicates, while negation alone has a negative monotonic $solve_s$ predicate.

Each CTL operator $op(T,A)$ returns a vector of Booleans C, where $C_i \leftarrow solve_i(T,A)$ (or $C_i \leftarrow solve_i(T,A,B)$ for the binary operators). It directly follows that for each $op(T,A) \mapsto C$, each entry of C_i is monotonic with respect to arguments T,A (and also with respect to B, for binary operators).

Finally, we also introduce a predicate $Model(A)$, which takes a Boolean state property vector A and returns TRUE iff the property A holds in the initial state. For simplicity, we will further assume (without loss of generality) that the initial state is always state s_0; predicate $Model(A)$ then simply returns the Boolean at index s_0 of the vector A: $Model(A) \mapsto A[s_0]$, and as a result is trivially positive monotonic with respect to the property state vector A.

8.3 Theory Propagation for CTL Model Checking

As described in Section 8.2 each CTL operator is monotonic, as is the predicate $Model(A)$. In Section 4.3, we described an approach to supporting theory propagation for compositions of positive and negative monotonic functions, relying on a function, $approx(\phi, \mathcal{M}_A^+, \mathcal{M}_A^-)$. In Algorithm 15 we describe our implementation of Algorithm 8 for the theory of CTL model checking, and in Algorithm 16, we describe our implementation of $approx$ for the theory of CTL model checking (with the over- and under-approximative models represented as two Kripke structures, K^+, K^-).[3] This approximation function in turn calls a function $evaluate(op,T,A)$, which takes a CTL operator, transition system, and a state property vector and

[3] Similar recursive algorithms for evaluating CTL formulas on partial Kripke structures, can be found in [43, 115, 142], applied to CTL model checking, rather than CTL synthesis.

returns a vector of Booleans containing, for each state s in T, the evaluation of op starting from state s. Our implementation of *evaluate* is simply the standard implementation of CTL model checking semantics (e.g., as described in [128]).

While our theory propagation implementation for CTL model checking closely follows that of Section 4.3, our implementation of conflict analysis differs from (and improves on) the general case described in Section 4.3. We describe our implementation of conflict analysis in the next section.

Algorithm 15 Theory propagation for CTL model checking. *T-Propagate* takes a (partial) assignment,$\mathcal{M}$. It returns a tuple (FALSE, conflict) if $\mathcal{M}$ is found to be unsatisfiable, and returns a tuple (TRUE, $\mathcal{M}$) otherwise. This implementation creates an over-approximative Kripke structure K^+ and an under-approximative Kripke structure K^-. These Kripke structures are then used to safely over- and under-approximate the evaluation of the CTL formula, in procedure APPROXCTL, and also during conflict analysis (procedure ANALYZECTL). The function NNF returns a negation-normal-form formula, discussed in Section 8.4.

```
function TheoryPropagate($\mathcal{M}$)
    $T^- \leftarrow \{\}, T^+ \leftarrow \{\}$
    for each transition atom $t_i$ do
        if $(t_i \notin T) \notin \mathcal{M}$ then
            $T^+ \leftarrow T^+ \cup \{t_i\}$
        if $(t_i \in T) \in \mathcal{M}$ then
            $T^- \leftarrow T^- \cup \{t_i\}$
    $P^- \leftarrow \{\}, P^+ \leftarrow \{\}$
    for each state property atom $p(s)$ do
        if $\neg(p(s)) \notin \mathcal{M}$ then
            $P^+ \leftarrow P^+ \cup \{p(s)\}$
        if $(p(s) \in \mathcal{M}$ then
            $P^- \leftarrow P^- \cup \{p(s)\}$
    $K^+ \leftarrow (T^+, P^+)$
    $K^- \leftarrow (T^-, P^-)$
    for each predicate atom $Model(\phi)$ do
        if $\neg Model(\phi) \in \mathcal{M}$ then
            if approxCTL$(Model(\phi), K^-, K^+) \mapsto$ True then
                return False, AnalyzeCTL$(NNF(\phi), s_0)$)
        else if $Model(\phi) \in \mathcal{M}$ then
            if approxCTL$(Model(\phi), K^+, K^-) \mapsto$ False then
                return False, AnalyzeCTL$(NNF(\phi), s_0)$)
        else
            if approxCTL$(Model(\phi), K^-, K^+) \mapsto$ True then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{Model(\phi)\}$
            else if approxCTL$(Model(\phi), K^+, K^-) \mapsto$ False then
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{\neg Model(\phi)\}$
    return True, $\mathcal{M}$
```

Algorithm 16 APPROXCTL(ϕ,K^+,K^-) takes a formula ϕ and two Kripke structures, K^+ and K^-. If ϕ is the predicate *Model*, it returns a truth value; otherwise it returns a vector of Booleans, representing the value of ϕ starting from each state of the Kripke structure.

```
Let $(T^+,P^+) = K^+$
Let $(T^-,P^-) = K^-$
if $\phi$ is atomic state vector $p$ then
    Lookup the state property vector p in the vector of vectors $P^+$
    return $P^+[p]$
else if $\phi$ is predicate $Model(\psi)$ then
    $A \leftarrow$ APPROXCTL($\psi$,$K^+$,$K^-$)
    Lookup the value of starting state $s_0$ in the vector of Booleans A
    return $A[s_0]$
else if $\phi$ is a unary operator $op$ with argument $\psi$ then
    if $op$ is $\neg$ then
        ($op$ is negative monotonic)
        $A$ := APPROXCTL($\psi$, $K^-$, $K^+$)
        return $evaluate(\neg,T^-,A)$
    else if op $\in \{$EX, EG$\}$ then
        $A$ := APPROXCTL($\psi$, $K^+$, $K^-$)
        return $evaluate(op,T^+,A)$
else $\phi$ is binary $op \in \{$EU$,\wedge\}$ with arguments $\psi_1$, $\psi_2$
    $A_1$ := APPROXCTL($\psi_1$, $K^+$, $K^-$)
    $A_2$ := APPROXCTL($\psi_2$, $K^+$, $K^-$)
    return $evaluate(op,A_1,A_2,T^+)$
```

8.4 Conflict Analysis for the Theory of CTL

Above we described our support for theory propagation for the theory of CTL model checking, relying on the function composition support described in Section 4.3. Here, we describe our implementation of conflict analysis for the theory of CTL model checking.

Unlike our theory propagation implementation, which operates on formulas in existential normal form, to perform conflict analysis we convert the CTL formula into *negation normal form* (NNF), pushing any negation operators down to the innermost terms of the formula. To obtain an adequate set, the formula may now also include universally quantified CTL operators and Weak Until. In Algorithm 15 above, the function $NNF(\phi)$ takes a formula in existential normal form and returns an equivalent formula in negation normal form. This translation to negation normal form takes time linear in the size of the input formula; the translation can be performed once, during preprocessing, and cached for future calls.

Our procedure $\text{ANALYZECTL}(\phi, s, K^+, K^-, \mathcal{M})$ operates recursively on the NNF formula, handling each CTL operator with a separate case, and returns a conflict set of theory literals. In Algorithm 17 below, we show only the cases handling operators EX, AX, EF, AF; the full algorithm including support for the remaining cases can be found in Appendix D.4.

Algorithm 17 ANALYZECTL$(\phi, s, K^+, K^-, \mathcal{M})$ analyzes ϕ recursively, and returns a conflict set. ϕ is a CTL formula in negation normal form, s is a state, K^+ and K^- are over-approximate and under-approximate Kripke structures, and $\mathcal{M}$ is a conflicting assignment. Notice that for the existentially quantified operators, the over-approximation transition system K^+ is analyzed, while for the universally quantified operators, the under-approximation transition system K^- is analyzed.

```
function ANALYZECTL($\phi, s, K^+, K^-, \mathcal{M}$)
    Let $(T^+, P^+) = K^+$
    Let $(T^-, P^-) = K^-$
    $c \leftarrow \{\}$
    if $\phi$ is $EX(\psi)$ then
        for each transition $t$ outgoing from $s$ do
            if $(t \notin T) \in \mathcal{M}$ then
                $c \leftarrow c \cup \{(t \in T)\}$.
        for each transition $t = (s, u)$ in $T^+$ do
            if $evaluate(\psi, u, K^+) \mapsto$ FALSE then
                $c \leftarrow c \cup$ ANALYZECTL$(\psi, n, K^+, K^-, \mathcal{M})$.
    else if $\phi$ is $AX(\psi)$ then
        Let $t = (s, u)$ be a transition in $T^-$, with $evaluate(\psi, u, K^+) \mapsto$ FALSE.
        (At least one such state must exist)
        $c \leftarrow c \cup \{(t \notin T)\}$
        $c \leftarrow c \cup$ ANALYZECTL$(\psi, u, K^+, K^-, \mathcal{M})$.
    else if $\phi$ is $EF(\psi)$ then
        Let $R$ be the set of all states reachable from $s$ in $T^+$.
        for each state $r \in R$ do
            for each transition $t$ outgoing from $r$ do
                if $(t \notin T) \in \mathcal{M}$ then
                    $c \leftarrow c \cup \{(t \in T)\}$
            $c \leftarrow c \cup$ ANALYZECTL$(\psi, r, K^+, K^-, \mathcal{M})$.
    else if $\phi$ is $AF(\psi)$ then
        Let $L$ be a set of states reachable from $s$ in $T^-$, such that $L$ forms a lasso from $s$, and such that for $\forall u \in L, evaluate(\psi, r, K^+) \mapsto$ FALSE.
        for each transition $t \in lasso$ do
            $c \leftarrow c \cup \{(t \notin T)\}$
        for each state $u \in L$ do
            $c \leftarrow c \cup$ ANALYZECTL$(\psi, u, K^+, K^-, \mathcal{M})$
    else if $\phi$ has operator $op \in \{EG, AG, EW, AW, EU, AU, \vee, \wedge, \neg p, p\}$ then
        See full description of ANALYZECTL in Appendix D.4.
    return $c$
```

8.5 Implementation and Optimizations

In Section 8.2, we showed how CTL model checking can be posed as a monotonic theory. In Sections 8.3 and 8.4, we then described our implementation of the theory propagation and conflict analysis procedures of a lazy SMT theory solver, following the techniques for supporting compositions of positive and negative monotonic functions and predicates described in Section 4.3. We have also implemented some additional optimizations which greatly improve the performance of our CTL theory solver. One basic optimization that we implement is pure literal filtering (see, e.g. [179]): For the case where $Model(\phi)$ is assigned TRUE (resp. FALSE), we need to check only whether $Model(\phi)$ is falsified (resp., made true) during theory propagation. In all of the instances we will examine in this chapter, $Model(\phi)$ is asserted TRUE in the input formula, and so this optimization greatly simplifies theory propagation. We describe several further improvements below.

In Section 8.5.1 we describe symmetry breaking constraints, which can greatly reduce the search space of the solver, and in Section 8.5.2 we show how several common types of CTL constraints can be cheaply converted into CNF, reducing the size of the formula the theory solver must handle. Finally, in Section 8.5.3, we discuss how in the common case of a CTL formula describing multiple communicating processes, we can (optionally) add support for additional *structural constraints*, similarly to the approach described in [68]. These structural constraints allow our solver even greater scalability, at the cost of adding more states into the smallest solution that can be synthesized.[4]

8.5.1 Symmetry Breaking

Due to the way we expose atomic propositions and transitions to the SAT solver with theory atoms, the SAT solver may end up exploring large numbers of isomorphic Kripke structures. We address this by enforcing extra symmetry-breaking constraints which prevent the solver from considering (some) redundant configurations of the Kripke structure. Symmetry reduction is especially helpful to prove instances UNSAT, which aids the search for suitable bounds.

[4] Thus, if structural constraints are used, iteratively decreasing the bound may no longer yield a minimal structure.

Let $label(s_i)$ be the binary representation of the atomic propositions of state s_i, and let $out(s_i)$ be the set of outgoing edges of state s_i. Let s_0 be the initial state. The following constraint enforces an order on the allowable assignments of state properties and transitions in the Kripke structure:

$$\begin{aligned}&\forall i, j :[i < j \wedge i \neq 0 \wedge j \neq 0] \rightarrow \\ &\quad \Big[\big(label(s_i) \leq label(s_j)\big) \wedge \Big(\big(label(s_i) = label(s_j)\big) \rightarrow \big(|out(s_i)| \leq |out(s_j)|\big)\Big)\Big]\end{aligned}$$

8.5.2 Preprocessing

Given a CTL specification ϕ, we identify certain common sub-expressions which can be cheaply converted directly into CNF, which is efficiently handled by the SAT solver at the core of MONOSAT. We do so if ϕ matches $\bigwedge_i \phi_i$, as is commonly the case when multiple properties are part of the specification. If ϕ_i is purely propositional, or of the form $\texttt{AG}p$, with p purely propositional, we eliminate ϕ_i from the formula and convert ϕ_i into a logically equivalent CNF expression over the state property assignment atoms of the theory.[5] This requires a linear number of clauses in the number of states in K. We also convert formulas of the form $\texttt{AG}\psi$, with ψ containing only propositional logic and at most a single Next-operator (EX or AX). Both of these are frequent sub-expressions in the CTL formulas that we have seen.

8.5.3 Wildcard Encoding for Concurrent Programs

As will be further explained later, the synthesis problem for synchronization skeletons assumes a given number of processes, which each have a local transition system. The state transitions in the full Kripke structure then represent the possible interleavings of executing the local transition system of each process. This local transition system is normally encoded into the CTL specification.

Both [12] and [68] explored strategies to take advantage of the case where the local transition systems of these processes are made explicit. The authors of [68] found they could greatly improve the scalability of their answer-set-programming

[5] Since $\texttt{AG}p$ only specifies reachable states, the clause is for each state s, a disjunction of p being satisfied in s, or s having no enabled incoming transitions. This changes the semantics of CTL for unreachable states, but not for reachable states.

based CTL synthesis procedure by deriving additional 'structural' constraints for such concurrent processes. As our approach is also constraint-based, we can (optionally) support similar structural constraints. In experiments below, we show that even though our approach already scales better than existing approaches without these additional structural constraints, we also benefit from such constraints.

Firstly, we can exclude any global states with state properties that are an illegal encoding of multiple processes. If the local state of each process is identified by a unique atomic proposition, then we can enforce that each global state must make true exactly one of the atomic propositions for each process. For every remaining combination of state property assignments, excluding those determined to be illegal above, we add a single state into the Kripke structure, with a pre-determined assignment of atomic propositions, such that only the transitions between these states are free for the SAT solver to assign. This is in contrast to performing synthesis without structural constraints, in which case all states are completely undetermined (but typically fewer are required).

Secondly, since we are interested in interleavings of concurrent programs, on each transition in the global Kripke structure, we enforce that only a single process may change its local state, and it may change its local state only in a way that is consistent with its local transition system.

The above two constraints greatly reduce the space of transitions in the global Kripke structure that are left free for the SAT solver to assign (and completely eliminate the space of atomic propositions to assign in each state). However these constraints make our procedure incomplete, since in general more than a single state with the same atomic propositions (but different behavior) need to be distinguished. To allow multiple states with equivalent atomic propositions, we also add a small number of 'wildcard' states into the Kripke structure, whose state properties and transitions (incoming and outgoing) are not set in advance. In the examples we consider in this chapter, we have found that a small number of such wildcard states (between 3 and 20) are sufficient to allow for a Kripke structure that satisfies the CTL formula, while still greatly restricting the total space of Kripke structures that must be explored by the SAT solver.

We disable symmetry breaking when using the wildcard encoding, as the wildcard encoding is incompatible with the constraint in Section 8.5.1.

8.6 Experimental Results

There are few CTL synthesis implementations available for comparison. Indeed, the original CTL synthesis/model-checking paper [56] presents an implementation of CTL model checking, but the synthesis examples were simulated by hand. The only publicly available, unbounded CTL synthesis tool we could find is Prezza's open-source CTLSAT tool[6], which is a modern implementation of the classic tableau-based CTL synthesis algorithm [56].

We also compare to De Angelis et al.'s encoding of bounded CTL synthesis into ASP [68]. De Angelis et al. provide encodings[7] specific to the *n*-process mutual exclusion example, which exploit structural assumptions about the synthesized model (for example, that it is the composition of *n* identical processes). We label this encoding "ASP-structural" in the tables below. For ASP-structural, we have only the instances originally considered in [68].

To handle the general version of CTL synthesis (without added structural information), we also created ASP encodings using the methods from De Angelis et al.'s paper, but without problem-specific structural assumptions and optimizations. We label those results "ASP-generic". For both encodings, we use the latest version (4.5.4) of Clingo [104], and for each instance we report the best performance over the included Clasp configurations.[8]

We compare these tools to two versions of MONOSAT: MONOSAT-structural, which uses the wildcard optimization presented in Section 8.5.3, and MONOSAT-generic, without the wildcard optimization.

With the exception of CTLSAT, the tools we consider are bounded synthesis tools, which take as input both a CTL formula and a maximum number of states. For ASP-structural, the state bounds follow [68]. For the remaining tools, we selected the state bound manually, by repeatedly testing each tool with different bounds, and reporting for each tool the smallest bound for which it found a satisfying solution. In cases where a tool could not find any satisfying solution within our time or memory bounds, we report out-of-time or out-of-memory.

[6] https://github.com/nicolaprezza/CTLSAT

[7] http://www.sci.unich.it/~{}deangelis/papers/mutex_FI.tar.gz

[8] These are: "auto", "crafty", "frumpy", "handy", "jumpy", "trendy", and "tweety".

8.6.1 The Original Clarke-Emerson Mutex

The mutex problem assumes that there are *n* processes that run concurrently and on occasion access a single shared resource. Instead of synthesizing entire programs, the original Clarke-Emerson example [56] considers an abstraction of the programs called *synchronization skeletons*. In the instance of a mutex algorithm, it is assumed that each process is in one of three states: *non-critical section* (**NCS**), the *try section* (**TRY**) or the *critical section* (**CS**). A process starts in the non-critical section in which it remains until it requests to access the resource, and changes to the try section. When it finally enters the critical section it has access to the resource, and eventually loops back to the non-critical section. The synthesis problem is to find a global Kripke structure for the composition of the *n* processes, such that the specifications are met. Our first set of benchmarks is based on the Clarke and Emerson specification given in [56], that includes mutual exclusion and starvation freedom for all processes.

	# of Processes				
Approach	**2**	**3**	**4**	**5**	**6**
CTLSAT	TO	TO	TO	TO	TO
ASP-generic	3.6 (7*)	1263.7 (14)	TO	MEM	MEM
ASP-structural	0.0 (12)	1.2 (36)	-	-	-
MonoSAT-gen	0.0 (7*)	1.4 (13*)	438.6 (23*)	1744.9 (42)	TO
MonoSAT-str	0.2 (7)	0.5 (13)	4.5 (23)	166.7 (41)	1190.5 (75)

Table 8.1: Results on the original Clarke-Emerson mutual exclusion example. Table entries are in the format *time(states)*, where *states* is the number of states in the synthesized model, and *time* is the run time in seconds. For ASP-structural, we only have the manually encoded instances provided by the authors. An asterisk indicates that the tool was able to prove minimality, by proving the instance is UNSAT at the next lower bound. TO denotes exceeding the 3hr timeout. MEM denotes exceeding 16GB of RAM. All experiments were run on a 2.67GHz Intel Xeon x5650 processor.

Table 8.1 presents our results on the mutex formulation from [56]. Both versions of MonoSAT scale to much larger instances than the other approaches, finding solutions for 5 and 6 processes, respectively. CTLSAT, implementing the classical tableau approach, times out on all instances.[9] Only the -generic versions can guarantee minimal solutions, and MONOSAT-generic is able to prove minimal models for several cases.

As expected, structural constraints greatly improve efficiency for both ASP-structural and MONOSAT-structural relative to their generic counterparts.

8.6.2 Mutex with Additional Properties

As noted in [121], the original Clarke-Emerson specification permits Kripke structures that are not *maximally parallel*, or even practically reasonable. For instance, our methods synthesize a structure in which one process being in NCS will block another process in TRY from getting the resource — the only transition such a global state has is to a state in which both processes are in the TRY section. In addition to the original formula, we present results for an augmented version in which we eliminate that solution[10] by introducing the "Non-Blocking" property, which states that a process may always remain in the NCS:

$$\texttt{AG}\ (\mathrm{NCS}_i \rightarrow\ \texttt{EX}\ \mathrm{NCS}_i) \tag{NB}$$

In addition, in the original paper there are structural properties implicit in the given local transition system, preventing jumping from NCS to CS, or from CS to TRY. We encode these properties into CTL as "No Jump" properties.

$$\texttt{AG}\ (\mathrm{NCS}_i \rightarrow\ \texttt{AX}\ \neg\mathrm{CS}_i)\ \wedge\ \texttt{AG}\ (\mathrm{CS}_i \rightarrow\ \texttt{AX}\ \neg\mathrm{TRY}_i) \tag{NJ}$$

We also consider two properties from [68]: Bounded Overtaking (BO), which guarantees that when a process is waiting for the critical section, each other process can only access the critical section at most once before the first process enters the

[9] Notably, CTLSAT times out even when synthesizing the original 2-process mutex from [56], which Clarke and Emerson originally synthesized by hand. This may be because in that work, the local transition system was specified implicitly in the algorithm, instead of in the CTL specification as it is here.

[10] While the properties that we introduce in this chapter mitigate some of the effects of underspecification, we have observed that the formulas of many instances in our benchmarks are not strong enough to guarantee a sensible solution. We are mainly interested in establishing benchmarks for synthesis performance, which is orthogonal to the task of finding suitable CTL specifications, which resolve these problems.

	# of Processes					
Approach	**2**	**3**	**4**	**5**	**6**	**7**
Property: ORIG $\land$ BO						
ASP-generic	3.4 (7*)	1442.0 (14)	TO/MEM	MEM	MEM	MEM
ASP-structural	0.0 (12)	2.3 (36)	-	-	-	-
MonoSAT-gen	0.0 (7*)	11.1 (13*)	438.3 (23*)	1286.6 (42)	TO	TO
MonoSAT-str	0.1 (7)	0.6 (13)	5.3 (23)	59.5 (41)	375.3 (75)	10739.5 (141)
Property: ORIG $\land$ BO $\land$ MR						
ASP-generic	10.1 (9*)	TO	MEM	MEM	MEM	MEM
ASP-structural	0.8 (10)	950.9 (27)	-	-	-	-
MonoSAT-gen	0.0 (9*)	6.0 (25*)	TO	TO	TO	TO
MonoSAT-str	0.1 (10)	8.7 (26)	TO	TO	TO	TO
Property: ORIG $\land$ NB $\land$ NJ						
ASP-generic	34.8 (9*)	TO	MEM	MEM	MEM	MEM
ASP-structural	0.1 (10)	7326.1 (27)	-	-	-	-
MonoSAT-gen	0.0 (9*)	1275.7 (22*)	TO	TO	TO	TO
MonoSAT-str	0.2 (10)	1.6 (26)	5314.7 (51)	TO	TO	TO
Property: ORIG $\land$ NB $\land$ NJ $\land$ BO						
ASP-generic	15.4 (9*)	TO	MEM	MEM	MEM	MEM
ASP-structural	0.1 (10)	TO	-	-	-	-
MonoSAT-gen	0.0 (9*)	127.7 (22*)	TO	TO	TO	TO
MonoSAT-str	0.1 (10)	1.3 (24)	TO	TO	TO	TO
Property: ORIG $\land$ NB $\land$ NJ $\land$ BO $\land$ MR						
ASP-generic	10.7 (9*)	TO	MEM	MEM	MEM	MEM
ASP-structural	0.1 (10)	1917.6 (27)	-	-	-	-
MonoSAT-gen	0.0 (9*)	4.4 (25*)	TO	TO	TO	TO
MonoSAT-str	0.1 (10)	2.7 (26)	TO	TO	TO	TO

Table 8.2: Results on the mutual exclusion example with additional properties (described in Section 8.6.2). As with Table 8.1, entries are in the format *time(states)*. ORIG denotes the original mutual exclusion properties from Section 8.6.1. As before, although problem-specific structural constraints improve efficiency, MonoSAT-generic is comparably fast to ASP-structural on small instances, and scales to larger numbers of processes. MonoSAT-structural performs even better.

critical section, and Maximal Reactivity (MR), which guarantees that if exactly one process is waiting for the critical section, then that process can enter the critical section in the next step.

In Table 8.2, we repeat our experimental procedure from Section 8.6.1, adding various combinations of additional properties. This provides a richer set of benchmarks, most of which are harder than the original. As before, the -structural constraints greatly improve efficiency, but nevertheless, MONOSAT-generic outperforms ASP-structural. MONOSAT-generic is able to prove minimality on several benchmarks, and on one benchmark, MONOSAT-structural scales to 7 processes.

8.6.3 Readers-Writers

To provide even more benchmarks, we present instances of the related Readers-Writers problem [63]. Whereas the Mutex problem assumes that all processes require exclusive access to a resource, the Readers-Writers problem permits some simultaneous access. Two types of processes are distinguished: writers, which require exclusive access, and readers, which can share their access with other readers. This is a typical scenario for concurrent access to shared memory, in which write permissions and read permissions are to be distinguished. The local states of each process are as in the Mutex instances.

We use Attie's [11] CTL specification. We note however that this specification allows for models which are not maximally parallel, and in particular disallows concurrent access by two readers. In addition to this original formula, we also consider one augmented with the Multiple Readers Eventually Critical (MREC) property. This ensures that there is a way for all readers, if they are in TRY, to simultaneously enter the critical section, if no writer requests the resource.

$$\mathtt{AG}\ \left(\bigwedge_{w_i} \mathrm{NCS}_{w_i} \rightarrow \left(\bigwedge_{r_i} \mathrm{TRY}_{r_i} \rightarrow\ \ \mathtt{EF}\ \ \bigwedge_{r_i} CS_{r_i}\right)\right) \qquad \text{(RW-MREC)}$$

This property turns out not to be strong enough to enforce that concurrent access for readers must always be possible. We introduce the following property, which we call Multiple Readers Critical. It states that if a reader is in TRY, and all other readers are in CS, it is possible to enter the CS in a next state – as long as all writers are in NCS, since they have priority access over readers.

	# of Processes (# of readers, # of writers)					
Approach	**2 (1, 1)**	**3 (2, 1)**	**4 (2, 2)**	**5 (3, 2)**	**6 (3, 3)**	**7 (4, 3)**
Property: RW						
CTLSAT	TO	TO	TO	TO	TO	TO
ASP-generic	0.6 (5*)	9.5 (9*)	TO	MEM	MEM	MEM
MonoSAT-gen	0.0 (5*)	0.0 (9*)	2.8 (19*)	30.0 (35*)	5312.7 (74)	TO
MonoSAT-str	0.1 (5)	0.5 (9)	0.7 (19)	2.9 (35)	98.8 (74)	384.4 (142)
Property: RW ∧ NB ∧ NJ						
ASP-generic	6.8 (8*)	2865.5 (16)	MEM	MEM	MEM	MEM
MonoSAT-gen	0.0 (8*)	1.4 (16*)	110.4 (27*)	843.8 (46*)	TO	TO
MonoSAT-str	0.1 (9)	0.2 (16)	3.4 (27)	35.9 (54)		TO
Property: RW ∧ NB ∧ NJ ∧ RW-MREC						
ASP-generic	2.4 (8*)	120.6 (22)	MEM	MEM	MEM	MEM
MonoSAT-gen	0.0 (8*)	238.4 (22*)	TO	TO	TO	TO
MonoSAT-str	0.1 (9)	0.25 (23)	5.3 (52)	159.1 (127)	TO	TO
Property: RW ∧ NB ∧ NJ ∧ RW-MRC						
ASP-generic	2.4 (8*)	TO	MEM	MEM	MEM	MEM
MonoSAT-gen	0.0 (8*)	1114.1 (22)	18.1 (27*)	251.6 (46*)	TO	TO
MonoSAT-str	0.1 (9)	0.2 (23)	2.5 (28)	28.0 (47)	TO	TO

Table 8.3: Results on the readers-writers instances. Property (RW) is Attie's specification [11]. Data is presented as in Table 8.1, in the format *time(states)*.

$$\texttt{AG}\ (\bigwedge_{w_i} \mathrm{NCS}_{w_i} \rightarrow (\mathrm{TRY}_{r_i} \bigwedge_{r_j \neq r_i} \mathrm{CS}_{r_j} \rightarrow\ \texttt{EX}\ \bigwedge_{r_i} CS_{r_i})) \tag{RW-MRC}$$

Using this property, we are able to synthesize a structure for two readers and a single writer, in which both readers can enter the critical section concurrently, independently of who enters it first, without blocking each other.

We ran benchmarks on problem instances of various numbers of readers and writers, and various combinations of the CTL properties. Since ASP-structural has identical process constraints, which make it unsuitable to solve an asymmetric problem such as Readers-Writers, we exclude it from these experiments. As with the Mutex problem, as CTLSAT is unable to solve even the simplest problem instances, we do not include benchmarks for the more complex instances.

Our experiments on each variation of the Readers-Writer problem are presented in Table 8.3. We observe that in general, Readers-Writers instances are easier to solve than Mutex instances with the same number of processes. At the same time, the additional properties introduced by us restrict the problem further, and make the instances harder to solve than the original Readers-Writers formulation. Taken together with the results from Tables 8.1 and 8.2, this comparison further strengthens our argument that MONOSAT-generic scales better than ASP-generic. The results also confirm that the structural MONOSAT solver making use of the wildcard encoding performs much better than MONOSAT-generic.

These experiments demonstrate that MONOSAT greatly outperforms existing tools for CTL synthesis. Further, MONOSAT has the ability to solve CTL formulas under additional constraints (e.g., about the structure of the desired solution), and can do so without sacrificing generality (by e.g., assuming identical processes). In many cases, we are also able to compute a provably minimal satisfying Kripke structure.

Chapter 9

Conclusions and Future Work

In this thesis, we have introduced a general framework for building lazy SMT solvers for finite, monotonic theories. Using this *SAT Modulo Monotonic Theories* framework, we have built a lazy SMT solver (MONOSAT) supporting important properties from graph theory, geometry, and automata theory, including many that previously had only poor support in SAT, SMT, or similar constraint solvers.

Furthermore, we have provided experimental evidence that SMT solvers built using our framework are efficient in practice. In fact, our solvers have significantly advanced the state of the art across several important problem domains, demonstrating real-world applications to circuit layout, data center management, and protocol synthesis.

For near-term future work, the monotonic theories already implemented in MONOSAT include many of the most commonly used properties of graph theory. Many important, real-world problems could benefit from our graph theory solver, and we have only just begin to explore them. To name just a few: product line configuration and optimization (e.g., [6, 149, 150]), many problems arising in software-defined networks and data centers (e.g., SDN verification [36], configuration [120, 155], and repair [206]), road/traffic design [139], and many problems arising in circuit routing (in addition to escape-routing, which we considered in Chapter 6.2).

In the longer term, in addition to the monotonic properties we describe in this thesis, there are many other important properties that can be modeled as finite monotonic theories, and which are likely amenable to our approach. For example, many important graph properties beyond the ones we already support are monotonic, and could be easily supported by our approach (e.g., Hamiltonian cycle detection, minimum Steiner tree weight, and many variations of network flow problems, are all monotonic in the edges of a graph). Alternatively, the geometry theory we describe in Chapter 7 can likely be generalized into a more comprehensive theory of constructive solid geometry, with applications to computer-aided design (CAD) tools.

A particularly promising monotonic theory, which we have not yet discussed in this thesis, is a theory of non-deterministic finite state machine string acceptance. Just as our CTL model checking theory is used to synthesize Kripke structures for which certain CTL properties do or do not hold, an FSM string acceptance theory can be used to synthesize finite state machines (of bounded size) that do or do not accept certain strings. This is a classic $\mathcal{NP}$-hard machine learning problem (see, e.g., [8, 196]), with potential applications to program synthesis.

In fact, many properties of non-deterministic state machines, including not only finite state machines but also push-down automata, Lindenmayer-systems [170], and even non-deterministic Turing machines, are monotonic with respect to the transition system of the state machine.

SAT and SMT solvers have proven themselves to be powerful constraint solvers, with great potential. The techniques in this thesis provide an easy and effective way to extend their reach to many new domains.

Bibliography

[1] IBM Platform Resouce Scheduler, http://www.ibm.com/support/knowledgecenter/ss8mu9_2.4.0/prs_kc/prs_kc_administering.html, (accessed 01-09-2016). URL http://www.ibm.com/support/knowledgecenter/SS8MU9_2.4.0/prs_kc/prs_kc_administering.html. → pages 94

[2] OpenStack Nova scheduler, http://docs.openstack.org/juno/config-reference/content/section_compute-scheduler.html, (accessed 01-09-2016). URL http://docs.openstack.org/juno/config-reference/content/section_compute-scheduler.html. → pages

[3] VMware Distributed Resource Management: Design, Implementation, and Lessons Learned, labs.vmware.com/vmtj/vmware-distributed-resource-management-design-implementation-and-lessons-learned, (accessed 05-10-2016). URL labs.vmware.com/vmtj/vmware-distributed-resource-management-design-implementation-and-lessons-learned. → pages 94

[4] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *Proceedings of the 39th annual Design Automation Conference*, pages 731–736. ACM, 2002. → pages 69

[5] F. A. Aloul, B. Al-Rawi, and M. Aboelaze. Routing and wavelength assignment in optical networks using Boolean satisfiability. In *Proceedings of the Consumer Communications and Networking Conference*, pages 185–189. IEEE, 2008. → pages 62, 69

[6] N. Andersen, K. Czarnecki, S. She, and A. Wasowski. Efficient synthesis of feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 106–115. ACM, 2012. → pages 62, 146

[7] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979. → pages 110

[8] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. → pages 147

[9] A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Recent Advances in AI Planning*, pages 97–108. Springer, 2000. → pages 17

[10] A. Armando, C. Castellini, E. Giunchiglia, and M. Maratea. A SAT-based decision procedure for the Boolean combination of difference constraints. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 16–29. Springer, 2005. → pages 20, 178

[11] P. C. Attie. Synthesis of large concurrent programs via pairwise composition. In *Proceedings of the International Conference on Concurrency Theory*, pages 130–145. Springer, 1999. → pages 125, 143, 144

[12] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):51–115, 1998. → pages 124, 125, 126, 137

[13] P. C. Attie and E. A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *Transactions on Programming Languages and Systems (TOPLAS)*, 23(2):187–242, 2001. → pages 123

[14] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 9, pages 399–404, 2009. → pages 12, 64, 67

[15] G. Audemard and L. Simon. Refining restarts strategies for SAT and UNSAT. In *Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012. → pages 12

[16] G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In *Proceedings of the International Conference on Automated Deduction*, pages 195–210. Springer, 2002. → pages 17, 20

[17] G. Audemard, J.-M. Lagniez, B. Mazure, and L. Saïs. Boosting local search thanks to CDCL. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 474–488. Springer, 2010. → pages 7

[18] Autodesk. AutoCAD Version 1.0. → pages 1

[19] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003. → pages 62

[20] G. J. Badros, A. Borning, and P. J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001. → pages 1

[21] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming*. Springer, 2003. → pages 98

[22] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM Computer Communication Review*, 2011. → pages 94

[23] K. Bansal, A. Reynolds, T. King, C. Barrett, and T. Wies. Deciding local theory extensions via e-matching. In *International Conference on Computer Aided Verification*, pages 87–105. Springer, 2015. → pages 23

[24] R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. *North-Holland Mathematics Studies*, 1985. → pages 98

[25] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003. → pages 63, 77

[26] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani. Data center network virtualization: A survey. *Communications Surveys & Tutorials*, 15(2):909–928, 2013. → pages 94

[27] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 187–201. Springer, 1996. → pages 17

[28] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 512–526. Springer, 2006. → pages 35

[29] C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007. → pages 1

[30] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the International Conference on Computer-Aided Verification*, 2011. → pages 173

[31] C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proceedings of the International Conference on Computer-Aided Verification*, pages 236–249. Springer, 2002. → pages 17

[32] R. J. Bayardo Jr and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 203–208, 1997. → pages 11, 12

[33] S. Bayless, N. Bayless, H. Hoos, and A. Hu. SAT Modulo Monotonic Theories. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, 2015. → pages 29, 68

[34] A. Biere. Lingeling, Plingeling and Treengeling. *Proceedings of SAT Competition*, 2013. → pages 64, 67

[35] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. *Symbolic model checking without BDDs*. Springer, 1999. → pages 1, 60

[36] N. Bjørner and K. Jayaraman. Checking cloud contracts in Microsoft Azure. In *International Conference on Distributed Computing and Internet Technology*, pages 21–32. Springer, 2015. → pages 146

[37] G. Boenn, M. Brain, M. De Vos, et al. Automatic composition of melodic and harmonic music by answer set programming. In *Logic Programming*, pages 160–174. Springer, 2008. → pages 77

[38] J. F. Botero, X. Hesselbach, M. Duelli, D. Schlosser, A. Fischer, and H. De Meer. Energy efficient virtual network embedding. *IEEE Communications Letters*, 16(5):756–759, 2012. → pages 69

[39] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Van Rossum, S. Schulz, and R. Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *Journal of Automated Reasoning*, 35 (1-3):265–293, 2005. → pages 20

[40] A. R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011. → pages 1

[41] A. R. Bradley and Z. Manna. Property-directed incremental invariant generation. *Formal Aspects of Computing*, 20(4-5):379–405, 2008. → pages 26, 60

[42] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009. → pages 18, 173

[43] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Proceedings of the International Conference on Computer-Aided Verification*. Springer, 1999. → pages 23, 130

[44] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT solver. In *Proceedings of the International Conference on Computer-Aided Verification*, pages 299–303. Springer, 2008. → pages 1

[45] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986. → pages 7

[46] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proceedings of the International Conference on Computer-Aided Verification*, pages 78–92. Springer, 2002. → pages 17

[47] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2): 142–170, 1992. → pages 60

[48] L. S. Buriol, M. G. Resende, and M. Thorup. Speeding up dynamic shortest-path algorithms. *INFORMS Journal on Computing*, 20(2): 191–204, 2008. → pages 61, 187

[49] D. Bustan and O. Grumberg. Simulation-based minimization. *Transactions on Computational Logic*, 2003. → pages 125

[50] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, volume 8, pages 209–224, 2008. → pages 1

[51] C. Carathéodory. Über den variabilitätsbereich der fourierschen konstanten von positiven harmonischen funktionen. *Rendiconti del Circolo Matematico di Palermo (1884-1940)*, 32(1):193–217, 1911. → pages 113, 194

[52] W.-T. Chan, F. Y. Chin, and H.-F. Ting. Escaping a grid by edge-disjoint paths. *Algorithmica*, 2003. → pages 86

[53] J. Chen. A new SAT encoding of the at-most-one constraint. *Constraint Modelling and Reformulation*, 2010. → pages 98

[54] N. Chowdhury, M. R. Rahman, and R. Boutaba. Virtual network embedding with coordinated node and link mapping. In *Proceedings of the International Conference on Computer Communications*, pages 783–791. IEEE, 2009. → pages 95, 96

[55] V. Chvatal. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory, Series B*, 18(1):39–41, 1975. → pages 119

[56] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of Programs*, 1982. → pages 122, 123, 124, 125, 139, 140, 141

[57] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004. → pages 1

[58] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981. → pages 60

[59] E. Coban, E. Erdem, and F. Ture. Comparing ASP, CP, ILP on two challenging applications: Wire routing and haplotype inference. *Proceedings of the International Workshop on Logic and Search*, 2008. → pages 86

[60] T. S. B. committee. Sort benchmark. http://sortbenchmark.org/, (accessed 26-01-2016). → pages 104

[61] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of Computing*, pages 151–158. ACM, 1971. → pages 3

[62] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, volume 6. MIT press Cambridge, 2001. → pages 71

[63] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 1971. → pages 143

[64] I. Cplex. Cplex 11.0 users manual. *ILOG SA, Gentilly, France*, page 32, 2007. → pages 69

[65] J. Cussens. Bayesian network learning by compiling to weighted MAX-SAT. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 2008. → pages 66

[66] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960. → pages 6

[67] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. → pages 6

[68] E. De Angelis, A. Pettorossi, and M. Proietti. Synthesizing concurrent programs using answer set programming. *Fundamenta Informaticae*, 120 (3-4):205–229, 2012. → pages 124, 126, 127, 136, 137, 139, 141

[69] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. *Computational Geometry*. Springer, 2000. → pages 111

[70] L. De Moura and N. Bjorner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2008. → pages 1, 21, 30, 64, 100, 170, 173

[71] L. de Moura and N. Bjørner. Satisfiability modulo theories: An appetizer. In *Formal Methods: Foundations and Applications*, pages 23–36. Springer, 2009. → pages 17

[72] L. De Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proceedings of the International Conference on Automated Deduction*, pages 438–455. Springer, 2002. → pages 17

[73] L. De Moura, H. Rueß, and M. Sorea. Lemmas on demand for satisfiability solvers. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 2002. → pages 17

[74] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990. → pages 11

[75] C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science*, pages 260–267. IEEE, 2001. → pages 66

[76] G. Dequen and O. Dubois. Kcnfs: An efficient solver for random k-SAT formulae. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 486–501. Springer, 2004. → pages 6

[77] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005. → pages 20

[78] B. L. Dietrich and A. J. Hoffman. On greedy algorithms, partially ordered sets, and submodular functions. *IBM Journal of Research and Development*, 47(1):25, 2003. → pages 49

[79] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. → pages 61, 187

[80] B. Dutertre and L. De Moura. A fast linear-arithmetic solver for DPLL (T). In *Proceedings of the International Conference on Computer-Aided Verification*, pages 81–94. Springer, 2006. → pages 1, 33

[81] B. Dutertre and L. De Moura. The Yices SMT solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006. → pages 1

[82] D. East and M. Truszczynski. More on wire routing with ASP. In *Workshop on ASP*, 2001. → pages 86

[83] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 61–75. Springer, 2005. → pages 12, 169

[84] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 333–336. Springer, 2004. → pages 11, 64, 67, 77, 169

[85] N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2 (1-4):1–26, 2006. → pages 98

[86] P. Elias, A. Feinstein, and C. Shannon. A note on the maximum flow through a network. *Transactions on Information Theory*, 2(4):117–119, 1956. → pages 72

[87] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Symposium on Theory of Computing*, pages 169–180. ACM, 1982. → pages 123

[88] E. Erdem and M. D. Wong. Rectilinear Steiner tree construction using answer set programming. In *Proceedings of the International Conference on Logic Programming*, 2004. → pages 86

[89] E. Erdem, V. Lifschitz, and M. D. Wong. Wire routing and satisfiability planning. In *Computational Logic*. 2000. → pages 86

[90] A. Erez and A. Nadel. Finding bounded path in graph using SMT for automatic clock routing. In *International Conference on Computer Aided Verification*, pages 20–36. Springer, 2015. → pages 56, 62, 67, 81, 86

[91] C. Ericson. *Real-time collision detection*. Elsevier Amsterdam/Boston, 2005. → pages 110, 114

[92] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, 1975. → pages 95

[93] J.-W. Fang, I.-J. Lin, P.-H. Yuh, Y.-W. Chang, and J.-H. Wang. A routing algorithm for flip-chip design. In *Proceedings of the International Conference on ComputerAided Design*, 2005. → pages 69, 86

[94] J.-W. Fang, I.-J. Lin, Y.-W. Chang, and J.-H. Wang. A network-flow-based RDL routing algorithms for flip-chip design. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2007. → pages 86

[95] W. R. Franklin. Pnpoly-point inclusion in polygon test. *Web site: http://www. ecse. rpi. edu/Homepages/wrf/Research/Short_Notes/pnpoly. html*, 2006. → pages 113, 194

[96] M. Fränzle and C. Herde. Hysat: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30 (3):179–198, 2007. → pages 23

[97] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Citeseer, 1995. → pages 6

[98] A. Fröhlich, A. Biere, C. M. Wintersteiger, and Y. Hamadi. Stochastic local search for satisfiability modulo theories. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1136–1143, 2015. → pages 18

[99] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2): 209–221, 1985. → pages 193

[100] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the International Conference on Computer-Aided Verification*, pages 519–531. Springer, 2007. → pages 1, 17, 173

[101] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL (T): Fast decision procedures. In *Proceedings of the International Conference on Computer-Aided Verification*, pages 175–188. Springer, 2004. → pages 17, 18, 19, 22

[102] M. N. V. P. Gao. Efficient pseudo-Boolean satisfiability encodings for routing and wavelength assignment in optical networks. In *Proceedings of the Ninth Symposium on Abstraction, Reformulation and Approximation*, 2011. → pages 62, 69

[103] C. Ge, F. Ma, J. Huang, and J. Zhang. SMT solving for the theory of ordering constraints. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 287–302. Springer, 2015. → pages 66

[104] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning*, pages 260–265. Springer, 2007. → pages 63, 77, 126, 139

[105] M. Gebser, T. Janhunen, and J. Rintanen. Answer set programming as SAT modulo acyclicity. In *Proceedings of the Twenty-first European Conference on Artificial Intelligence (ECAI14)*, 2014. → pages 66

[106] M. Gebser, T. Janhunen, and J. Rintanen. SAT modulo graphs: Acyclicity. In *Logics in Artificial Intelligence*, pages 137–151. Springer, 2014. → pages 56, 63, 66

[107] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision proceduresthe case study of modal K. In *International Conference on Automated Deduction*, pages 583–597. Springer, 1996. → pages 17

[108] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. 1996. → pages 20

[109] E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007. → pages 9

[110] R. Goldman. Intersection of two lines in three-space. In *Graphics Gems*, page 304. Academic Press Professional, 1990. → pages 115, 195

[111] D. Goldwasser, O. Strichman, and S. Fine. A theory-based decision heuristic for DPLL (T). In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, page 13. IEEE Press, 2008. → pages 22

[112] A. Griggio, Q.-S. Phan, R. Sebastiani, and S. Tomasi. Stochastic local search for SMT: combining theory solvers with walkSAT. In *Frontiers of Combining Systems*, pages 163–178. Springer, 2011. → pages 18

[113] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference (CONEXT)*, page 15. ACM, 2010. → pages 94, 99, 100, 101, 102

[114] A. Gupta, J. Kleinberg, A. Kumar, R. Rastogi, and B. Yener. Provisioning a virtual private network: a network design problem for multicommodity flow. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, pages 389–398, 2001. → pages 95

[115] A. Gupta, A. E. Casavant, P. Ashar, A. Mukaiyama, K. Wakabayashi, and X. Liu. Property-specific testbench generation for guided simulation. In

Proceedings of the 2002 Asia and South Pacific Design Automation Conference, page 524. IEEE Computer Society, 2002. → pages 23, 130

[116] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 664–672. IEEE, 1995. → pages 66

[117] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997. → pages 66

[118] M. Heule, M. Dufour, J. Van Zwieten, and H. Van Maaren. March_eq: implementing additional reasoning into an efficient look-ahead SAT solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 345–359. Springer, 2005. → pages 6

[119] M. J. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Proceedings of the Haifa Verification Conference*, pages 50–65. Springer, 2011. → pages 7

[120] J. A. Hewson, P. Anderson, and A. D. Gordon. A declarative approach to automated configuration. In *Large Installation System Administration Conference*, volume 12, pages 51–66, 2012. → pages 146

[121] S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Synthesis from temporal specifications using preferred answer set programming. In *Theoretical Computer Science*, pages 280–294. Springer, 2005. → pages 124, 126, 127, 141

[122] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM (JACM)*, 48(5): 1038–1068, 2001. → pages 23

[123] Y.-K. Ho, H.-C. Lee, and Y.-W. Chang. Escape routing for staggered-pin-array PCBs. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011. → pages 69

[124] F. Hoffmann. On the rectilinear art gallery problem. In *International Colloquium on Automata, Languages, and Programming*, pages 717–728. Springer, 1990. → pages 119

[125] I. D. Horswill and L. Foged. Fast procedural level population with playability constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2012. → pages 63

[126] I. Houidi, W. Louati, W. B. Ameur, and D. Zeghlache. Virtual network provisioning across multiple substrate networks. *Computer Networks*, 55 (4):1011–1023, 2011. → pages 69

[127] C.-H. Hsu, H.-Y. Chen, and Y.-W. Chang. Multi-layer global routing considering via and wire capacities. In *Proceedings of the International Conference on ComputerAided Design*, 2008. → pages 85

[128] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. 1999. → pages 126, 128, 131

[129] D. Jackson. Automating first-order relational logic. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 130–139. ACM, 2000. → pages 62

[130] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2): 256–290, 2002. → pages 62

[131] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proceedings of the International Conference on Software Engineering*, pages 730–733. IEEE, 2000. → pages 62

[132] S. Jacobs and R. Bloem. Parameterized synthesis. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 362–376. Springer, 2012. → pages 124

[133] T. Janhunen, S. Tasharrofi, and E. Ternovska. SAT-TO-SAT: Declarative extension of SAT solvers with new propagators. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016. → pages 23, 56, 62

[134] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–92, 2012. → pages 6

[135] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990. → pages 9

[136] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010. → pages 1

[137] R. M. Karp, F. T. Leighton, R. L. Rivest, C. D. Thompson, U. V. Vazirani, and V. V. Vazirani. Global wire routing in two-dimensional arrays. *Algorithmica*, 1987. → pages 85

[138] H. A. Kautz, B. Selman, et al. Planning as satisfiability. In *Proceedings of the European Conference on Artificial Intelligence*, volume 92, pages 359–363, 1992. → pages 1

[139] B. Kim, A. Jarandikar, J. Shum, S. Shiraishi, and M. Yamaura. The SMT-based automatic road network generation in vehicle simulation environment. In *Proceedings of the 13th International Conference on Embedded Software*, page 18. ACM, 2016. → pages 146

[140] P. Kohli and P. H. Torr. Efficiently solving dynamic Markov random fields using graph cuts. In *Proceedings of the International Conference on Computer Vision*, volume 2, pages 922–929. IEEE, 2005. → pages 61, 70, 188, 190

[141] D.-T. Lee and A. Lin. Computational complexity of art gallery problems. *Information Theory, IEEE Transactions on*, 32(2):276–282, 1986. → pages 119

[142] W. Lee, A. Pardo, J.-Y. Jang, G. Hachtel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 76–81. IEEE Computer Society, 1997. → pages 23, 130

[143] L. Luo and M. D. Wong. Ordered escape routing based on Boolean satisfiability. In *Proceedings of the Asia and South Pacific Design Automation Conference*, 2008. → pages 86

[144] M. Mahfoudh, P. Niebert, E. Asarin, and O. Maler. A satisfiability checker for difference logic. *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 2:222–230, 2002. → pages 33, 178

[145] J. Marques-Silva, M. Janota, and A. Belov. Minimal sets over monotone predicates in Boolean formulae. In *Proceedings of the International Conference on Computer-Aided Verification*, pages 592–607. Springer, 2013. → pages 26

[146] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5): 506–521, 1999. → pages 4, 6, 8, 11

[147] A. Martin. Adequate sets of temporal connectives in CTL. *Electronic Notes in Theoretical Computer Science*, 2002. EXPRESS'01, 8th International Workshop on Expressiveness in Concurrency. → pages 127

[148] K. L. McMillan. Interpolation and SAT-based model checking. In *Proceedings of the International Conference on Computer-Aided Verification*, pages 1–13. Springer, 2003. → pages 1

[149] M. Mendonça. *Efficient reasoning techniques for large scale feature models*. PhD thesis, University of Waterloo, 2009. → pages 62, 146

[150] M. Mendonca, A. Wasowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, pages 231–240. Carnegie Mellon University, 2009. → pages 62, 146

[151] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001. → pages 4, 6, 8

[152] A. Nadel. Routing under constraints. 2016. → pages 67, 86

[153] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar. A new FPGA detailed routing approach via search-based Boolean satisfiability. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(6): 674–684, 2002. → pages 62

[154] G.-J. Nam, F. Aloul, K. A. Sakallah, and R. A. Rutenbar. A comparative study of two Boolean formulations of FPGA detailed routing constraints. *Transactions on Computers*, 53(6):688–696, 2004. → pages 62

[155] S. Narain, G. Levin, S. Malik, and V. Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management*, 16(3):235–258, 2008. → pages 146

[156] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979. → pages 52

[157] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980. → pages 62

[158] A. Niemetz, M. Preiner, and A. Biere. Precise and complete propagation based local search for satisfiability modulo theories. In *International Conference on Computer Aided Verification*, pages 199–217. Springer, 2016. → pages 18

[159] R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications*, pages 453–468. Springer, 2005. → pages 62

[160] R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, 2007. → pages 62

[161] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 36–50. Springer, 2005. → pages 17, 18, 22

[162] G. Optimization et al. Gurobi optimizer reference manual. *URL: http://www.gurobi.com*, 2:1–3, 2012. → pages 69

[163] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation. In *Proceedings of the International Conference on Computer Design*, pages 459–464. IEEE, 2000. → pages 6

[164] D. J. Pearce and P. H. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithmics (JEA)*, 11: 1–7, 2007. → pages 65, 188

[165] L. Perron. Operations research and constraint programming at Google. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*. 2011. → pages 89

[166] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 294–299. Springer, 2007. → pages 9

[167] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986. → pages 6

[168] M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005. → pages 7

[169] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993. → pages 12

[170] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012. → pages 147

[171] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, 1996. → pages 61, 187, 190

[172] D. Ranjan, D. Tang, and S. Malik. A comparative study of 2QBF algorithms. In *The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 292–305, 2004. → pages 23

[173] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002. → pages 1

[174] J. Rintanen. Madagascar: Efficient planning with SAT. *The 2011 International Planning Competition*, page 61, 2011. → pages 1

[175] J. Rintanen, K. Heljanko, and I. Niemelä. Parallel encodings of classical planning as satisfiability. In *European Workshop on Logics in Artificial Intelligence*, pages 307–319. Springer, 2004. → pages 66

[176] M. Rost, C. Fuerst, and S. Schmid. Beyond the stars: Revisiting virtual cluster embeddings. In *SIGCOMM Computer Communication Review*, 2015. → pages 94

[177] N. Ryzhenko and S. Burns. Standard cell routing via Boolean satisfiability. In *Proceedings of the 49th Annual Design Automation Conference*, pages 603–612. ACM, 2012. → pages 62

[178] S. Schewe and B. Finkbeiner. Bounded synthesis. In *Automated Technology for Verification and Analysis*, pages 474–488. Springer, 2007. → pages 124

[179] R. Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007. → pages 18, 21, 30, 136, 169

[180] B. Selman, H. Kautz, B. Cohen, et al. Local search strategies for satisfiability testing. *Cliques, coloring, and satisfiability: Second DIMACS implementation challenge*, 26:521–532, 1993. → pages 4, 6

[181] A. Smith. Personal communication, 2014. → pages 73

[182] A. M. Smith and M. Mateas. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Symposium on Computational Intelligence and Games (CIG)*, pages 273–280. IEEE, 2010. → pages 63, 77

[183] A. M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *Transactions onComputational Intelligence and AI in Games*, 3(3):187–200, 2011. → pages 63, 77

[184] A. Solar-Lezama. The sketching approach to program synthesis. In *Programming Languages and Systems*, pages 4–13. Springer, 2009. → pages 1

[185] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review*, 40(5):404–415, 2006. → pages 23

[186] N. Sörensson and A. Biere. Minimizing learned clauses. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 237–243. Springer, 2009. → pages 11

[187] P. M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing*, 4(3):375–380, 1975. → pages 192

[188] G. M. Stalmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formula, 1994. US Patent 5,276,897. → pages 7

[189] W. Stein et al. Sage: Open source mathematical software. 2008. → pages 22

[190] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In *International Conference on Computer Aided Verification*, pages 209–222. Springer, 2002. → pages 1, 178

[191] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *Logic in Computer Science*, volume 1, pages 29–37. DTIC Document, 2001. → pages 1

[192] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A cooperating validity checker. In *International Conference on Computer Aided Verification*, pages 500–504. Springer, 2002. → pages 1, 17

[193] W. Szeto, Y. Iraqi, and R. Boutaba. A multi-commodity flow based approach to virtual network resource allocation. In *Global Telecommunications Conference and Exhibition (GLOBECOM)*, 2003. → pages 95

[194] R. Tamassia and I. G. Tollis. Dynamic reachability in planar digraphs with one source and one sink. *Theoretical Computer Science*, 119(2):331–343, 1993. → pages 66

[195] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. → pages 189

[196] M. Tomita. Learning of construction of finite automata from examples using hill-climbing. 1982. → pages 147

[197] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, pages 466–483. Springer, 1983. → pages 5

[198] H. Visser and P. Roling. Optimal airport surface traffic planning using mixed integer linear programming. In *Proceedings of the AIAA's 3rd Annual Aviation Technology, Integration, and Operations (ATIO) Technical Forum*, 2003. → pages 69

[199] T. Walsh. SAT v CSP. In *International Conference on Principles and Practice of Constraint Programming*, pages 441–456. Springer, 2000. → pages 7

[200] C. Wang, F. Ivančić, M. Ganai, and A. Gupta. Deciding separation logic formulae by SAT and incremental negative cycle elimination. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 322–336. Springer, 2005. → pages 33, 178

[201] D. Wang, P. Zhang, C.-K. Cheng, and A. Sen. A performance-driven I/O pin routing algorithm. In *Proceedings of the Asia and South Pacific Design Automation Conference*, 1999. → pages 86

[202] R. Wang, R. Shi, and C.-K. Cheng. Layer minimization of escape routing in area array packaging. In *Proceedings of the International Conference on ComputerAided Design*, 2006. → pages 86, 91

[203] R. G. Wood and R. A. Rutenbar. FPGA routing and routability estimation via Boolean satisfiability. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(2):222–231, 1998. → pages 62

[204] C.-A. Wu, T.-H. Lin, C.-C. Lee, and C.-Y. R. Huang. QuteSAT: a robust circuit-based SAT solver for complex circuit structure. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1313–1318. EDA Consortium, 2007. → pages 6

[205] Y. Wu, L. Chiaraviglio, M. Mellia, and F. Neri. Power-aware routing and wavelength assignment in optical networks. *Proceedings of teh European Conference on Optical Communication (ECOC)*, 2009. → pages 69

[206] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated network repair with meta provenance. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 26. ACM, 2015. → pages 146

[207] T. Yan and M. D. Wong. A correct network flow model for escape routing. In *Proceedings of the Design Automation Conference*, 2009. → pages 86, 87, 91

[208] T. Yan and M. D. Wong. Recent research development in PCB layout. In *Proceedings of the International Conference on Computer-Aided Design*, 2010. → pages 86

[209] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review*, 38(2):17–29, 2008. → pages 95, 96

[210] M.-F. Yu and W. W.-M. Dai. Pin assignment and routing on a single-layer pin grid array. In *Proceedings of the Asia and South Pacific Design Automation Conference*, 1995. → pages 86

[211] M.-f. Yu, J. Darnauer, and W. W.-M. Dai. *Planar interchangeable 2-terminal routing*. Technical Report, UCSC, 1995. → pages 85

[212] Y. Yuan, A. Wang, R. Alur, and B. T. Loo. On the feasibility of automation for bandwidth allocation problems in data centers. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, 2013. → pages 69, 94, 99, 100, 101, 102, 103

[213] Z. Yuanlin and R. H. Yap. Arc consistency on n-ary monotonic and linear constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 470–483. Springer, 2000. → pages 23

[214] H. Zang, C. Ou, and B. Mukherjee. Path-protection routing and wavelength assignment (RWA) in WDM mesh networks under duct-layer constraints. *IEEE/ACM Transactions on Networking (TON)*, 11(2):248–258, 2003. → pages 69

[215] ZeroStack. Private cloud provider. https://www.zerostack.com/, (accessed 26-01-2016). → pages 104

[216] H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *In Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH96), Fort Lauderdale (Florida USA*. Citeseer, 1996. → pages 8

[217] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Proceedings of the International Conference on Computer-Aided Verification*, pages 17–36. Springer, 2002. → pages 12

[218] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001. → pages 11

[219] A. Zook and M. O. Riedl. Automatic game design via mechanic generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 530–537, 2014. → pages 63

[220] E. Zulkoski, V. Ganesh, and K. Czarnecki. Mathcheck: A math assistant via a combination of computer algebra systems and SAT solvers. In *Proceedings of the International Conference on Automated Deduction*, pages 607–622. Springer, 2015. → pages 22, 23, 56, 62

Appendix A

The MonoSAT Solver

Here we provide a brief, technical overview of our implementation of the SMT solver MonoSAT, used throughout this thesis for our experiments. While each of the theory solvers is described in detail in this thesis (first as an abstract framework in Chapter 4, and then in specifics in Chapters 5, 7, and 8), here we discuss the implementation of the SMT solver itself, and its integration with the theory solvers.

Except for the unusual design of the theory solvers, MonoSAT is a typical example of a lazy SMT solver, as described, for example, in [179]. The SAT solver is based on MiniSat 2.2 [84], including the integrated SatELite [83] preprocessor. Unlike in MiniSat, some variables are marked as being theory atoms, and are associated with theory solvers. Theory atoms are prevented from being eliminated during preprocessing.

As described in Chapter 4, each theory solver implements both a *T-Propagate* and a *T-Analyze* method. However, we also implement a few additional methods in each theory solver:

1. *T-Enqueue*(l), called each time a theory literal l is assigned in the SAT solver.
2. *T-Backtrack*(l), called each time a theory literal l is unassigned in the SAT solver.
3. *T-Decide*$(\mathcal{M})$, called each time the SAT solver makes a decision.

Throughout Chapter 4, each algorithm (e.g., Algorithm 3) is described in a stateless manner. For example, each time *T-Propagate* is called, the **Update-Bounds** section of Algorithm 3 recomputes the over- and under-approximate assignments from scratch. In practice, such an implementation would be very inefficient. In MONOSAT, we make a significant effort to avoid redundant work, in several ways. First, as described throughout the thesis, where available, we use dynamic algorithms to actually evaluate predicates during theory propagation and analysis.

Secondly, differing from the description in Chapter 4, our actual implementation of the **UpdateBounds** method described in Chapter 4 is stateful, rather than stateless, and is implemented in the *T-Enqueue* and *T-Backtrack* methods, rather than in *T-Propagate*. In fact, when *T-Propagate* is called, the under- and over-approximate assignments have already been computed, and *T-Propagate* only needs to execute the **PropagateBounds** section. For example, Algorithm 18 describes *T-Enqueue* and *T-Backtrack*, as implemented in our graph theory solver. Each time the SAT solver assigns a theory literal (to either polarity), it calls *T-Enqueue* on the corresponding theory solver; each time it unassigned a theory literal (when backtracking), it calls *T-Backtrack*. Each of our theory solvers implements the **UpdateBounds** functionality in this manner.

A second important implementation detail is the way that justification clauses are produced. In some SMT solvers (such as Z3 [70]), each time a theory solver implies a theory literal l during theory propagation, the theory solver immediately generates a 'justification set' for l. (Recall from Section 2.4 that a justification set is a collection of mutually unsatisfiable atoms.) The justification set is then negated and turned into a learned clause which would have been sufficient to imply l by unit propagation, under the current assignment. Adding this learned clause to the solver, even though l has already been implied by the theory solver, allows the SAT solver to derive learned clauses correctly if the assignment of l leads to a conflict.

In our implementation, we delay creating justification sets for theory implied literals, creating them only lazily when, during conflict analysis, the SAT solver needs to access the 'reason' cause for l. At that time, we create the justification clause in the theory solver by backtracking the theory solver to just after l would be assigned, and executing *T-Analyze* in the theory solver to create a justification set for l. As the vast majority of implied theory literals are never actually directly

Algorithm 18 Implementations of *T-Enqueue* and *T-Backtrack* in MONOSAT for the theory of graphs, which replace the **UpdateBounds** section described in Chapter 4, with a stateful implementation, in which E^- and E^+ are maintained between calls to the theory solver (E^- is initialized to the empty set, while E^+ is initialized to E).

```
function T-Enqueue($l$)
    $l$ is a theory literal
    if $l$ is an edge literal $(e \in E)$ then
        $E^- \leftarrow E^- \cup \{e_i\}$
    else if $l$ is a negated edge literal $(e \notin E)$ then
        $E^+ \leftarrow E^+ \setminus \{e_i\}$
function T-Backtrack($l$)
    $l$ is a theory literal
    if $l$ is an edge literal $(e \in E)$ then
        $E^- \leftarrow E^- \setminus \{e_i\}$
    else if $l$ is a negated edge literal $(e \notin E)$ then
        $E^+ \leftarrow E^+ \cup \{e_i\}$
```

involved in a conflict analysis, justification sets are only rarely actually constructed by MONOSAT. This is important, as in the theories we consider, *T-Analyze* is often very expensive (for example, for the maximum flow and reachability predicates, *T-Analyze* performs an expensive minimum cut analysis).

Finally, each theory solver is given an opportunity to make decisions in the SAT solver (with a call to the function *T-Decide*), superseding the SAT solver's default VSIDS heuristic. Currently, only the reach and maximum flow predicates actually implement *T-Decide*. Those implementations are described Chapter 5. While in many cases these theory decisions lead to major improvements in performance, we have also found that in some common cases these theory decisions can cause pathologically bad behaviour in the solver, and in particular are poorly suited for maze generation tasks (such as the ones described in Section 6.1). In this thesis, theory solver decision heuristics are enabled only in the experiments of Sections 6.2, 6.3, for the maximum flow predicate.

MONOSAT is written in C++ (with a user-facing API provided in Python 2/3). The various theory solvers include code from several open source libraries, which are documented in the source code. In the course of our experiments in this thesis, various, improving versions of MONOSAT were tested. In Section 6.1, MONOSAT was compiled with G++ 4.8.1; in the remaining experiments, it was compiled with g++ 6.0. Experiments were conducted using the Python API of MONOSAT, with Python 3.3 (Section 6.1) and Python 3.4 (all other experiments).

MONOSAT is freely available under an open-source license, and can be found at www.cs.ubc.ca/labs/isd/projects/monosat.

Appendix B

Monotonic Theory of Bitvectors

We describe a simple monotonic theory of bitvectors. This theory solver is designed to be efficiently combinable with the theory of graphs (Chapter 5), by deriving tight bounds on each bitvector variable as assignments are made in the solver, and passing those bounds to other theory solvers in the form of comparison atoms on shared bitvector variables (as described in Section 4.5).

The theory of bitvectors we introduce here is a theory of fixed-width, non-wrapping bitvectors, supporting only comparison predicates and addition. In contrast, the standard theory of bitvectors widely considered in the literature (e.g., [30, 42, 70, 100]), which is essentially a theory of finite modular arithmetic, supports many inherently non-monotonic operators (such as bitwise xor).

In the standard theory of bitvectors, the formula

$$x + 2 = y$$

with x, y both 3-bit unsigned bitvectors, both $\{x = 5, y = 7\}$, and $\{x = 7, y = 1\}$ are satisfying assignments, as $7 + 2 \equiv 1 \pmod 8$. In contrast, in our non-wrapping theory of bitvectors, $\{x = 5, y = 7\}$ is satisfying, while $\{x = 7, y = 1\}$ is not.

Formally, the theory of non-wrapping bitvectors for bit-width n is a subset of the theory of integer arithmetic, in which every variable and constant x is constrained to the range $0 \leq x < 2^n$, for fixed n. As all variables are restricted to a finite range of values, this fragment could be easily bit-blasted into propositional logic, however we instead will handle it as a lazy SMT theory, using the SMMT framework. While the performance of this theory solver on its own is in fact not competitive with bit-blasting bitvector theory solvers, handling this theory lazily has the advantage that it allows for comparison atoms to be created on the fly without needing to bit-blast comparator circuits for each one. This is important, as this bitvector theory will be combined with other monotonic theory solvers by deriving and passing large numbers of comparison atoms for each bitvector. In practice, a very large number of such comparison atoms will be created by the solver, and we found it to be prohibitively expensive to construct bit-blasted comparator circuits for each of these comparison atoms, thus motivating the creation of a new theory solver.

In addition to comparison predicates, this theory of bitvectors supports one function: addition. Formally, the function $add(a,b) \mapsto c$ takes two bitvector arguments (a,b) of the same width and outputs a third bitvector (c) of the same bit-width. In our non-wrapping bitvector theory, if a and b are the n-width bitvector arguments of an addition function, it is enforced that $a+b < 2^n$ (by treating $a+b \geq 2^n$ as a conflict in the theory).

In Section 4.3, we described two approaches to supporting function composition in monotonic theory solvers:

1. Flatten any function composition by introducing fresh variables, treating the functions as predicates relating those variables, and then apply Algorithm 3.

2. Directly support propagation with function composition, using Algorithm 8.

While we take the latter approach in Chapter 8, we take the former approach here (for historical reasons, as the bitvector theory solver was developed before we began work on the CTL theory solver).

Our implementation of theory propagation for the (non-wrapping) theory of bitvectors is described in below. For each n-bit bitvector b, we introduce atoms $(b_0 \in b),(b_1 \in b)\ldots(b_{n-1} \in b)$ to expose the assignment of each bit to the SAT solver. For notational convenience, we represent the negation of $(b_i \in b)$ as $(b_i \notin b)$.

Algorithm 19 Theory propagation for the theory of non-wrapping bitvectors.

```
function T-Propagate($\mathcal{M}$)
    $\mathcal{M}$ is a partial assignment; theory propagate returns a tuple (FALSE, conflictSet) if $\mathcal{M}$ is found to be UNSAT, and otherwise returns (TRUE, $\mathcal{M}$), possibly deriving additional atoms to add to $\mathcal{M}$. In practice, we maintain bounds $b^-, b^+$ from previous calls, only updating them if an atom involving b has been added to or removed from $\mathcal{M}$.
    assignmentChanged $\leftarrow$ FALSE
    for each $n$-bit bitvector variable $b$ do
        $b^- \leftarrow 0, b^+ \leftarrow 0$
        for $i$ in $0..n-1$ do
            if $(b_i \notin b) \notin \mathcal{M}$ then
                $b^+ \leftarrow b^+ + 2^i$
            if $(b_i \in b) \in \mathcal{M}$ then
                $b^- \leftarrow b^- + 2^i$
    for each comparison to constant $(b \leq \sigma_i)$ do
        if $(b \leq \sigma_i) \in \mathcal{M}$ then
            $b^+ \leftarrow min(b^+, \sigma_i)$
        else if $(b > \sigma_i) \in \mathcal{M}$ then
            $b^- \leftarrow max(b^-, \sigma_i + 1)$
        else
            if $b^+ \leq \sigma_i$ then
                assignmentChanged $\leftarrow$ TRUE
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{(b \leq \sigma_i)\}$
            else if $b^- > \sigma_i$ then
                assignmentChanged $\leftarrow$ TRUE
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{(b > \sigma_i)\}$
    for each comparison atom between bitvectors $(a \leq b)$ do
        if $(a \leq b) \in \mathcal{M}$ then
            $a^+ \leftarrow min(a^+, b^+)$
            $b^- \leftarrow max(b^-, a^-)$
        else if $(a > b) \in \mathcal{M}$ then
```

```
            $a^- \leftarrow max(a^-, b^- + 1)$
            $b^+ \leftarrow min(b^+, a^+ + 1)$
        **else**
            **if** $a^+ \leq b^-$ **then**
                $assignmentChanged \leftarrow$ TRUE
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{(a \leq b)\}$
            **else if** $a^- > b^+$ **then**
                $assignmentChanged \leftarrow$ TRUE
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{(b > a)\}$
    **for each** addition function $add(a + b) \mapsto c$ **do**
        $c^+ \leftarrow min(c^+, a^+ + b^+)$
        $c^- \leftarrow max(c^-, a^+ + b^+)$
        $a^+ \leftarrow min(a^+, c^+ - b^-)$
        $a^- \leftarrow max(a^-, c^- - b^+)$
        $b^+ \leftarrow min(b^+, c^+ - a^-)$
        $b^- \leftarrow max(b^-, c^- - a^+)$
    **for each** bitvector variable $b$ of width $n$ **do**
        $b^- \leftarrow$ REFINE_LBOUND$(b^-, n, \mathcal{M})$
        $b^+ \leftarrow$ REFINE_UBOUND$(b^+, n, \mathcal{M})$
        **if** $b^- > b^+$ **then**
            **return** FALSE, *T-Analyze*$(\mathcal{M})$
        **else**
            **if** $(b \leq b^+) \notin \mathcal{M}$ **then**
                $assignmentChanged \leftarrow$ TRUE
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{(b \leq b^+)\}$
            **if** $(b \geq b^-) \notin \mathcal{M}$ **then**
                $assignmentChanged \leftarrow$ TRUE
                $\mathcal{M} \leftarrow \mathcal{M} \cup \{(b \geq b^-)\}$
    **if** $assignmentChanged$ **then**
        **return** *T-Propagate*$(\mathcal{M})$
    **else**
        **return** TRUE, $\mathcal{M}$
```

We describe our theory propagation implementation in Algorithm 19. There are two improvements that we make in this implementation, as compared to Algorithm 3. The first is to add calls to REFINE_LBOUND and REFINE_UBOUND; these functions attempt to derive tighter bounds on a bitvector by excluding assignments that are incompatible with the bit-level assignment of the variable. For example, if we have a bound $b \geq 2$, but we also have $(b_1 \notin b) \in \mathcal{M}$, then we know that b cannot equal exactly 2, and so we can refine the bound on b to $b \geq 3$. These functions perform a search attempting to tighten the lower bound using reasoning as above, requiring linear time in the bit-width n. We omit the pseudo-code from this thesis, as the implementation is a bit involved; it can be found in "BvTheorySolver.h" in the (open-source) implementation of MONOSAT. We also omit the description of the implementation of *T-Analyze*, which is a straightforward adaptation of Algorithm 6.

The second change we make relative to Algorithm 3 is, in the case where any literals have been added to $\mathcal{M}$, to call THEORYPROPAGATE again, with the newly refined $\mathcal{M}$ as input. This causes THEORYPROPAGATE to repeat until the $\mathcal{M}$ reaches a fixed point (termination is guaranteed, as literals are never removed from $\mathcal{M}$ during THEORYPROPAGATE, and there is a finite set of literals that can be added to $\mathcal{M}$). Calling THEORYPROPAGATE when the $\mathcal{M}$ changes allows refinements to the bounds on one bitvector to propagate through addition functions or comparison predicates to refine the bounds on other bitvectors, regardless of the order in which we examine the bitvectors and predicates in a given call to THEORYPROPAGATE.

The non-wrapping bitvector theory solver we describe here is sufficient for our purposes in this thesis, supporting the weighted graph applications described in Section 6.3, which makes fairly limited use of the bitvector theory solver. However, the performance of this solver is not competitive with existing bitvector or arithmetic theory solvers (in particular, the repeated calls to establish a fixed point can be quite expensive).

An alternative approach, which we have not yet explored, would be to implement an efficient difference logic theory solver. Difference logic is sufficiently expressive for some of our applications, and should be combinable with our monotonic theories in the same manner as the theory of bitvectors, as difference logic supports comparisons to constants. A fast difference logic solver would almost certainly outperform the above bitvector theory (considered in isolation from other monotonic theory solvers). However, while existing difference logic theory solvers (e.g., as described in [10, 144, 190, 200]) are designed to quickly determine the satisfiability of a set of difference logic constraints; they are not (to the best of our knowledge) designed to efficiently derive not just satisfying, but tight, upper and lower bounds on each variable, from partial assignments during theory propagation. In contrast, the bitvector theory solver we described in this section is designed to find and maintain tight upper and lower bounds on each variable, from partial assignments. This is critically important, as passing tight bounds to other monotonic theory solvers (such as the weighted graph theory of Chapter 5) can allow those other theory solvers to prune unsatisfying assignments early.

Appendix C

Supporting Materials

C.1 Proof of Lemma 4.1.1

This lemma can be proven for the case of totally ordered σ, by considering the following exhaustive, mutually exclusive cases:

Proof.

1. $T \models \neg\mathcal{M}_A$. In this case, $\mathcal{M}_A$ is by itself already an inconsistent assignment to the comparison atoms. This can happen if some variable x is assigned to be both larger and smaller than some constant, or is assigned to be larger than $\sigma_\top$ or smaller than $\sigma_\perp$. In this case, as $\mathcal{M}_A$ (and hence any superset of $\mathcal{M}_A$) is theory unsatisfiable, $\mathcal{M}_A \underset{T}{\Rightarrow} p$ holds trivially for any p, and hence both the left and right hand sides of 4.1 and 4.2 hold.

2. $\mathcal{M}_A \vDash T$, $T \cup \mathcal{M}_A \models \neg(x \geq \sigma_j)$. In this case, $\mathcal{M}_A \cup \{x \geq \sigma_i\} \underset{T}{\Rightarrow} p$ holds trivially for any p, and so 4.1 holds.

3. $\mathcal{M}_A \vDash T$, $T \cup \mathcal{M}_A \models \neg(x \geq \sigma_i)$. In this case, $\mathcal{M}_A \cup \{x \geq \sigma_i\} \underset{T}{\Rightarrow} p$ holds trivially for any p. As $\mathcal{M}_A$ is by itself theory satisfiable, and $\mathcal{M}_A$ only contains assignments of variables to constants, there must exist some $\sigma_k < \sigma_i$, such that atom $(x \leq \sigma_k) \in \mathcal{M}_A$. As $\sigma_j \geq \sigma_i$, we also have $\sigma_j > \sigma_k$, and $T \cup \mathcal{M}_A \models \neg(x \geq \sigma_j)$. Therefore, $\mathcal{M}_A \cup \{x \geq \sigma_i\} \underset{T}{\Rightarrow} p$ holds trivially for any p, and so 4.1 holds.

4. $\mathcal{M}_A \cup \{x \geq \sigma_i, x \geq \sigma_j\} \vDash T$. In this case, if $\mathcal{M}_A \cup \{x \geq \sigma_i\} \underset{T}{\Rightarrow} p$, then by Definition 1, $\mathcal{M}_A \cup \{x \geq \sigma_j\} \underset{T}{\Rightarrow} p$, as p is positive monotonic. Otherwise, if $\mathcal{M}_A \cup \{x \geq \sigma_i\} \underset{T}{\Rightarrow} \neg p$, then the antecedent of 4.1 is false, and so 4.1 holds.

   The cases for $T \cup \mathcal{M}_A \models \neg(x \leq \sigma_j)$,$T \cup \mathcal{M}_A \models \neg(x \leq \sigma_i)$,$\mathcal{M}_A \cup \{x \leq \sigma_i, x \leq \sigma_j\} \vDash T$ are symmetric, but for 4.2.

□

C.2 Proof of Correctness for Algorithm 7

In Section 4.3, we introduced algorithm APPROX, for finding safe upper and lower approximations to compositions of positive and negative monotonic functions. We repeat it here for reference:

Algorithm 7 Approximate evaluation of composed monotonic functions

```
function APPROX($\phi$,$\mathcal{M}_A^+, \mathcal{M}_A^-$)
    $\phi$ is a formula, $\mathcal{M}_A^+, \mathcal{M}_A^+$ are assignments.
    if $\phi$ is a variable or constant term then
        return $\mathcal{M}_A^+[\phi]$
    else $\phi$ is a function term or predicate atom $f(t_0, t_1, \ldots, t_n)$
        for $0 \leq i \leq n$ do
            if $f$ is positive monotonic in $t_i$ then
                $x_i =$ APPROX($t_i, \mathcal{M}_A^+, \mathcal{M}_A^-$)
            else arguments $\mathcal{M}_A^+, \mathcal{M}_A^-$ are swapped
                $x_i =$ APPROX($t_i, \mathcal{M}_A^-, \mathcal{M}_A^+$)
        return $evaluate(f(x_0, x_1, x_2, \ldots, x_n))$
```

In the same section, we claimed that the following lemma holds:

Lemma C.2.1. *Given any term ϕ composed of mixed positive and negative monotonic functions (or predicates) over variables $vars(\phi)$, and given any three complete, T-satisfying models for ϕ, $\mathcal{M}_A^+$, $\mathcal{M}_A^*$, and $\mathcal{M}_A^-$:*

$$\forall x \in vars(\phi), \mathcal{M}_A^+[x] \geq \mathcal{M}_A^*[x] \geq \mathcal{M}_A^-[x] \implies \text{APPROX}(\phi, \mathcal{M}_A^+, \mathcal{M}_A^-) \geq \mathcal{M}_A^*[\phi]$$

—and—

$$\forall x \in vars(\phi), \mathcal{M}_A^+[x] \leq \mathcal{M}_A^*[x] \leq \mathcal{M}_A^-[x] \implies \text{APPROX}(\phi, \mathcal{M}_A^+, \mathcal{M}_A^-) \leq \mathcal{M}_A^*[\phi]$$

The proof is given by structural induction over APPROX:

Proof.

Inductive hypothesis:

$$\forall x \in vars(\phi), \mathcal{M}_A^+[x] \geq \mathcal{M}_A^*[x] \geq \mathcal{M}_A^-[x] \implies \text{APPROX}(\phi, \mathcal{M}_A^+, \mathcal{M}_A^-) \geq \mathcal{M}_A^*[\phi]$$

—and—

$$\forall x \in vars(\phi), \mathcal{M}_A^+[x] \leq \mathcal{M}_A^*[x] \leq \mathcal{M}_A^-[x] \implies \text{APPROX}(\phi, \mathcal{M}_A^+, \mathcal{M}_A^-) \leq \mathcal{M}_A^*[\phi]$$

Base case (ϕ is x, a variable or constant term):

APPROX returns $\mathcal{M}_A^+[x]$; therefore,
$\mathcal{M}_A^+[x] \geq \mathcal{M}_A^*[x]$ implies $\text{APPROX}(x, \mathcal{M}_A^+, \mathcal{M}_A^-) \geq \mathcal{M}_A^*[x]$.
$\mathcal{M}_A^+[x] \leq \mathcal{M}_A^*[x]$ implies $\text{APPROX}(x, \mathcal{M}_A^+, \mathcal{M}_A^-) \leq \mathcal{M}_A^*[x]$.

Inductive step (ϕ is a function or predicate term $f(t_0, t_1, \ldots t_n)$):

APPROX returns $evaluate(f(x_0, x_1, \ldots, x_n))$. For each x_i, there are two cases:

1. f is positive monotonic in argument i, $x_i \leftarrow \text{APPROX}(t_i, \mathcal{M}_A^+, \mathcal{M}_A^-)$

   In this case we have (by the inductive hypothesis):

   $\forall x \in vars(\phi), \mathcal{M}_A^+[x] \geq \mathcal{M}_A^*[x] \geq \mathcal{M}_A^-[x]$ implies $x_i = \text{APPROX}(t_i, \mathcal{M}_A^+, \mathcal{M}_A^-) \geq \mathcal{M}_A^*[t_i]$, and we also have

   $\forall x \in vars(\phi), \mathcal{M}_A^+[x] \leq \mathcal{M}_A^*[x] \leq \mathcal{M}_A^-[x]$ implies $x_i = \text{APPROX}(t_i, \mathcal{M}_A^+, \mathcal{M}_A^-) \leq \mathcal{M}_A^*[t_i]$.

2. f is negative monotonic in argument i, $x_i \leftarrow \text{APPROX}(t_i, \mathcal{M}_A^-, \mathcal{M}_A^+)$

   In this case we have (by the inductive hypothesis):

   $\forall x \in vars(\phi), \mathcal{M}_A^+[x] \geq \mathcal{M}_A^*[x] \geq \mathcal{M}_A^-[x]$ implies $x_i = \text{APPROX}(t_i, \mathcal{M}_A^-, \mathcal{M}_A^+) \leq \mathcal{M}_A^*[t_i]$, and we also have

   $\forall x \in vars(\phi), \mathcal{M}_A^+[x] \leq \mathcal{M}_A^*[x] \leq \mathcal{M}_A^-[x]$ implies $x_i = \text{APPROX}(t_i, \mathcal{M}_A^-, \mathcal{M}_A^+) \geq \mathcal{M}_A^*[t_i]$.

In other words, in the case that $\mathcal{M}_A^+[x] \geq \mathcal{M}_A^-[x]$, then for each positive monotonic argument i, we have $x_i \geq \mathcal{M}_A^*[t_i]$, and for each negative monotonic argument i, we have $x_i \leq \mathcal{M}_A^*[t_i]$. Additionally, in the case that $\mathcal{M}_A^+[x] \leq \mathcal{M}_A^-[x]$, we have the opposite: then for each positive monotonic argument i, we have $x_i \leq \mathcal{M}_A^*[t_i]$, and for each negative monotonic argument i, we have $x_i \geq \mathcal{M}_A^*[t_i]$.

Therefore, we also have:

$\forall x \in vars(\phi), \mathcal{M}_A^+[x] \geq \mathcal{M}_A^*[x] \geq \mathcal{M}_A^-[x]$ implies $f(x_0, x_1, \ldots, x_n) \geq f(\mathcal{M}_A^*[t_0], \mathcal{M}_A^*[t_1], \ldots, \mathcal{M}_A^*[t_n])$ implies $\textsc{Approx}(f(t_0, t_1, \ldots, t_n), \mathcal{M}_A^+, \mathcal{M}_A^-) \geq \mathcal{M}_A^*[f(t_0, t_1, \ldots, t_n)]$,

and we also have

$\forall x \in vars(\phi), \mathcal{M}_A^+[x] \leq \mathcal{M}_A^*[x] \leq \mathcal{M}_A^-[x]$ implies $f(x_0, x_1, \ldots, x_n) \leq f(\mathcal{M}_A^*[t_0], \mathcal{M}_A^*[t_1], \ldots, \mathcal{M}_A^*[t_n])$ implies $\textsc{Approx}(f(t_0, t_1, \ldots, t_n), \mathcal{M}_A^+, \mathcal{M}_A^-) \leq \mathcal{M}_A^*[f(t_0, t_1, \ldots, t_n)]$.

This completes the proof.

□

C.3 Encoding of Art Gallery Synthesis

In Table 7.1, we described a set of comparisons between MONOSAT and the SMT solver Z3 on *Art Gallery Synthesis* instances. While the translation of this problem into the geometry theory supported by MONOSAT is straightforward, the encoding into Z3's theory of linear real arithmetic is more involved, and we describe it here.

Each synthesis instance consists of a set of points P, at pre-determined 2D coordinates, within a bounded rectangle. From these points, we must find N non-overlapping convex polygons, with vertices selected from those points, such that the area of each polygon is greater than some specified constant, and all vertices of all the polygons (as well as the 4 corners of the bounding rectangle) can be 'seen' by a fixed set of at most M cameras. A camera is modeled as a point, and is defined as being able to see a vertex if a line can reach from that camera to the vertex without intersecting any of the convex polygons. Additionally, to avoid degenerate solutions, we enforce that the convex polygons cannot meet at only a single point: they must either share entire edges, or must not meet at all.

Encoding this synthesis task into linear real arithmetic (with propositional constraints) requires modeling notions of point containment, line intersection, area, and convex hulls, all in 2 dimensions. The primary intuition behind the encoding we chose is to ensure that at each step, all operations can be applied in arbitrary precision rational arithmetic. In particular, we will rely on computing cross products to test line intersection, using the following function:

$$crossDif(O,A,B) = (A[0]-O[0]) * (B[1]-O[1]) - (A[1]-O[1]) * (B[0]-O[0])$$

This function takes non-symbolic, arbitrary precision arithmetic 2D points as arguments, and return non-symbolic arbitrary precision results.

Our encoding can be broken down into two parts. First, we create a set of *N symbolic polygons*. Each symbolic polygon consists of a set of Booleans $enabled(p)$, controlling for each point $p \in P$ whether p is enabled in that symbolic polygon. Additionally, for each pair of points (p_1, p_2), the symbolic polygon defines a formula $onHull(p_1, p_2)$ which evaluates to TRUE iff (p_1, p_2) is one of the edges that make up the convex hull of the enabled points. Given points p_1, p_2, $onHull(p_1, p_2)$ evaluates to true iff:

$$enabled(p_1) \wedge enabled(p_2) \bigwedge_{\forall p_3 \in P \setminus \{p_1, p_2\}} (crossDif(p2, p1, p3) \geq 0 \implies enabled(p_3))$$

As there are a bounded number of points, the universal quantifier is unrolled into a conjunction of $|P| - 2$ individual conditionals; as *onHull* is computed for each ordered pair p_1, p_2, there are a cubic number of total constraints required to compute *onHull* for each polygon.

Given two symbolic polygons S_1, S_2 with $onHull_{S1}$, $onHull_{S2}$ computed as above, our approach to enforcing that they do not overlap is to assert that there exists an edge normal to one of the edges on the hull of one of the two polygons, such that each enabled point of S_1 and S_2 do not overlap when projected onto that normal. The normal edge is selected non-deterministically by the solver (using $|P|^2$ constraints), and the projections are computed and compared using dot products, using $|P|^3$ constraints.

The separation constraint described above requires $|P|^3$ constraints for a given pair of symbolic polygons S_1, S_2, and is applied pairwise between each unique pair of symbolic polygons.

The next step of the encoding is to ensure that it is possible to place M cameras in the room while covering all of the vertices. Before describing how this is computed, we first introduce three formulas for testing point and line intersection. $containsPoint(S, p)$ takes a symbolic polygon S and a point p, and returns true if p is contained within the convex hull of S.

We define $containsPoint(S, p)$ as follows:

$$containsPoint(S, p) = \bigwedge_{\forall (e_1, e_2) \in edges(S)} (\neg onHull_S(e_1, e_2) \vee crossDif(e_2, e_1, p) < 0)$$

$lineIntersectsPolygon(S, (p_1, p_2))$ takes a symbolic polygon S, and a line defined as a tuple of points (p_1, p_2), and evaluates TRUE if the line (p_1, p_2) intersects one of the edges of the hull of S (not including vertices). If the line does not intersect S, or only intersects a vertex of S, then *lineIntersectsPolygon* evaluates to FALSE.

We define $lineIntersectsPolygon(S,(p_1,p_2))$ as follows:

$$lineIntersectsPolygon(S,(p_1,p_2)) = containsPoint(S,p_1) \vee containsPoint(S,p_2) \bigvee_{\forall (e_1,e_2) \in edges(S)} (onHull_S(e_1,e_2) \wedge lineIntersectsLine((p_1,p_2),(e_1,e_2)))$$

Finally, $lineIntersectsLine((p_1,p_2),(e_1,e_2))$ returns true if the two (non symbolic) lines defined by endpoints p_1, p_2 and e_1, e_2 intersect each other. This can be tested for non-symbolic rational constants using cross products.

Given a fixed camera position p_c, and a fixed vertex to observe p_v, $isVisible(p_c, p_v)$ is evaluated as follows:

$$isVisible(p_c,p_v) = \bigwedge_{\forall S \in Polygons} \neg lineIntersectsPolygon(S,(p_c,p_v))$$

We then use a cardinality constraint to force the solver to pick C positions from the point set P to be cameras, and use the line intersection formula described above to force that every vertex that is on hull of a symbolic polygon must be visible from at least one vertex that is a camera (meaning that no symbolic polygon intersects the line between the camera and that vertex).

Each instance also specifies a minimum area A for each symbolic polygon (to ensure we do not get degenerate solutions consisting of very thin polygon segments). We compute the area of the convex hull of each symbolic polygon S as:

$$area(S) = \frac{1}{2} \sum_{\forall (e_1,e_2) \in edges(S)} If(onHull_S(e1,e2), e1[0]*e2[1] - e1[1]*e2[0])$$

To complete the encoding, we assert:

$$\forall S, area(S) \geq A$$

Appendix D

Monotonic Predicates

This is a listing of the monotonic predicates considered in this thesis, arranged by topic.

D.1 Graph Predicates

D.1.1 Reachability

Monotonic Predicate: $reach_{s,t}(E)$, true iff t can be reached from u in the graph formed of the edges enabled in E.

Implementation of $evaluate(reach_{s,t}(E), G)$**:** We use the dynamic graph reachability/shortest paths algorithm of Ramalingam and Reps [171] to test whether t can be reached from s in G. Ramalingam-Reps is a dynamic variant of Dijkstra's Algorithm [79]); our implementation follows the one described in [48]. If there are multiple predicate atoms $reach_{s,t}(E)$ sharing the same source s, then Ramalingam-Reps only needs to be updated once for the whole set of atoms.

Implementation of $analyze(reach_{s,t}E, G^-)$ Node s reaches t in G^-, but $reach_{s,t}(E)$ is assigned FALSE in $\mathcal{M}$. Let $e_0, e_1, \ldots$ be a $u - v$ path in G^-; return the conflict set $\{(e_0 \in E), (e_1 \in E), \ldots, \neg reach_{s,t}(E)\}$.

Implementation of $analyze(\neg reach_{s,t}(E), G^+)$ Node t cannot be reached from s

in G^+, but $reach_{s,t}(E)$ is assigned TRUE. Let $e_0, e_1, \ldots$ be a cut of disabled edges ($e_i \notin E$) separating u from v in G^+. Return the conflict set $\{(e_0 \notin E), (e_1 \notin E), \ldots, reach_{s,t}(E)\}$.

We find a minimum separating cut by creating a graph containing all edges of E (including both the edges of G^+ and the edges that are disabled in G^+ in the current assignment), in which the capacity of each disabled edge of E is 1, and the capacity of all other edges is infinity (forcing the minimum cut to include only edges that correspond to disabled edge atoms). Any standard maximum *s-t* flow algorithm can then be used to find a minimum cut separating u from v. In our implementation, we use the dynamic Kohli-Torr [140] minimum cut algorithm for this purpose.

Decision Heuristic: (Optional) If $reach_{s,t}(E)$ is assigned TRUE in $\mathcal{M}$, but there does not yet exist a $u - v$ path in G^-, then find a $u - v$ path in G^+ and pick the first unassigned edge in that path to be assigned true as the next decision. In practice, such a path has typically already been discovered, during the evaluation of $reach_{s,t}$ on G^+ during theory propagation.

D.1.2 Acyclicity

Monotonic Predicate: $acyclic(E)$, true iff there are no (directed) cycles in the graph formed by the enabled edges in E.

Implementation of $evaluate(acyclic(E), G)$**:** Apply the PK dynamic topological sort algorithm (as described in [164]). The PK algorithm is a fully dynamic graph algorithm that maintains a topological sort of a directed graph as edges are added to or removed from the graph; as a side effect, it also detects directed cycles (in which case no topological sort exists). Return FALSE if the PK algorithm successfully produces a topological sort, and return TRUE if it fails (indicating the presence of a directed cycle).

Implementation of $analyze(acyclic(E), G^-)$ There is a cycle in E, but $acyclic(E)$ is assigned TRUE. Let $e_0, e_1, \ldots$ be the edges that make up a directed cycle in E; return the conflict set $\{(e_0 \in E), (e_1 \in E), \ldots, acyclic(E)\}$.

Implementation of $analyze(\neg acyclic(E), G^{+})$ There is no cycle in E, but $acyclic(E)$ is assigned FALSE. Let $e_0, e_1, \ldots$ be the set of all edges not in E^{+}; return the conflict set $\{(e_0 \notin E), (e_1 \notin E), \ldots, \neg acyclic(E)\}$. (Note that this is the default monotonic conflict set.)

D.1.3 Connected Components

Here we consider a predicate which constraints the number of (simple) connected components in the graph to be less than or greater than some constant. There are both directed and undirected variations of this predicate; we describe the directed version, which counts the number of strongly connected components.

Monotonic Predicate: $connectedComponentsE, m$, true iff the number of connected components in the graph formed of edges E is $\leq m$, with m a bitvector.

Implementation of $evaluate(connectedComponents(E,m), G)$**:** Use Tarjan's SCC algorithm [195] to count the number of distint strongly connected components in G; return TRUE iff the count is $\leq m$.

Implementation of $analyze(connectedComponents(E,m), G^{-})$**:** The connected component count is less or equal to m, but $connectedComponents(E,m)$ is FALSE in $\mathcal{M}$. Construct a spanning tree for each component of G^{-} (using depth-first search). Let edges $e_1, e_2, \ldots$ be the edges in these spanning trees; return the conflict set $\{(e_1 \in E), (e_2 \in E), \ldots, (m > m^{-}), \neg connectedComponents(E,m)\}$.

Implementation of $analyze(\neg connectedComponents(E,m), G^{+})$**:** The connected component count is greater than m, but $connectedComponents(E,m)$ is TRUE in $\mathcal{M}$ Collect all disabled edges $e_i = (u,v)$ where u and v belong to different components in G^{+}; the conflict set is $\{(e_1 \notin E), (e_2 \notin E), \ldots, (m < m^{+}), connectedComponents(E,m)\}$.

Above we describe the implementation for directed graphs; for undirected graphs, Tarjan's SCC algorithm does not apply. For undirected graphs, we count the number of connected components using disjoint-sets/union-find.

D.1.4 Shortest Path

Monotonic Predicate: $shortestPath_{s,t}(E,l,c_0,c_1,\ldots)$, true iff the length of the shortest $s-t$ path in the graph formed of the edges enabled in E with edge weights $c_0,c_1,\ldots$ is less than or equal to bit-vector l.

Implementation of $evaluate(shortestPath_{s,t}(E,l,c_0,c_1,\ldots),G)$**:** We use the dynamic graph reachability/shortest paths algorithm of Ramalingam and Reps [171] to test whether the shortest path to t from s is less than l. If there are multiple predicate atoms $shortestPath_{s,t}(E)$ sharing the same source s, then Ramalingam-Reps only needs to be updated once for the whole set of atoms.

Implementation of $analyze(shortestPath_{s,t},E,l,c_0,c_1,\ldots)$**:** There exists an s-t path in G^- of length $\leq l$, but $shortestPath_{s,t}$ is assigned FALSE in $\mathcal{M}$. Let $e_0,e_1,\ldots$ be a shortest $s-t$ path in G^- with weights $w_0,w_1,\ldots$; return the conflict set $\{(e_0 \in E),(w_0 \geq w_0^-),(e_1 \in E),(w_1 \geq w_1^-),\ldots,\neg shortestPath_{s,t}\}$.

Implementation of $analyze(\neg shortestPath_{s,t},E,l,c_0,c_1,\ldots)$**:** Walk back from u in G^+ along enabled and unassigned edges with edge weights $w_0,w_1,\ldots$; collect all incident disabled edges $e_0,e_1,\ldots$; return the conflict set $\{(e_0 \notin E),(e_1 \notin E),\ldots,(w_0 \leq w_0^+),(w_1 \leq w_1^+)\ldots,shortestPath_{u,v,G\leq C}(edges)\}$.

D.1.5 Maximum Flow

Monotonic Predicate: $maxFlow_{s,t}(E,m,c_0,c_1,\ldots)$, true iff the maximum s-t flow in G with edge capacities $c_0,c_1,\ldots$ is $\geq m$.

Implementation of $evaluate(maxFlow_{s,t},G,m,c_0,c_1,\ldots)$**:** We apply the dynamic minimum-cut/maximum s-t algorithm by Kohli and Torr [140] to compute the maximum flow of G, with edge capacities set by c_i. Return TRUE iff that flow is greater or equal to m.

Implementation of $analyze(maxFlow_{s,t},G^-,m^+,c_0^-,c_1^-,\ldots)$**:** The maximum s-t flow in G^- is f, with $f \geq m^+$). In the computed flow, each edge e_i is either disabled in G^-, or it has been allocated a (possibly zero-valued) flow f_i, with $f_i \leq c_i^-$. Let $e_a, e_b, \ldots$ be the edges enabled in G^- with non-zero allocated flows $f_a, f_b, \ldots$. Either one of those edges must be disabled in the graph, or one of the capacities of those edges must be decreased, or the flow will be at least f. Return the conflict set $\{(e_a \in E), (e_b \in E), \ldots, (c_a \geq f_a), (c_b \geq f_b) \ldots, (m \leq f), maxFlow_{s,t}(E, m, c_0, c_1, \ldots)\}$.

Implementation of $analyze(\neg maxFlow_{s,t}, G^+, m^-, c_0^+, c_1^+, \ldots)$**:** The maximum *s*-*t* flow in G^+ is f, with $f < m^-$. In the flow, each edge that is enabled in G^+ has been allocated a (possibly zero-valued) flow f_i, with $f_i \leq c_i^+$.

Create a graph G^{cut}. For each edge $e_i = (u, v)$ in G^+, with $f_i < c_i^+$, add a forward edge (u, v) to G^{cut} with infinite capacity, and also a backward edge (v, u) with capacity f_i. For each edge $e_i = (u, v)$ in G^+ with $f_i = c_i^+$, add a forward edge (u, v) to G^{cut} with capacity 1, and also a backward edge (v, u) with capacity f_i. For each edge $e_i = (u, v)$ that is disabled in G^+, add only the forward edge (u, v) to G^{cut}, with capacity 1.

Compute the minimum *s-t* cut of G^{cut}. Some of the edges along this cut may have been edges disabled in G^+, while some may have been edges enabled in G^+ with fully utilized edge capacity. Let $e_a, e_b, \ldots$ be the edges of the minimum cut of G^{cut} that were disabled in G^+. Let $c_c, c_d, \ldots$ be the capacities of edges in the minimum cut for which the edge was included in G^+, with fully utilized capacity. Return the conflict set $\{(e_a \notin E), (e_b \notin E), \ldots, (c_c \leq f_c), (c_d \leq f_d), \ldots, (m > f), \neg maxFlow_{s,t}(E, m, c_0, c_1, \ldots)\}$.

In practice, we maintain a graph G^{cut} for each maximum flow predicate atom, updating its edges only lazily when needed for conflict analysis.

Decision Heuristic: (Optional) If $maxFlow_{s,t}$ is assigned TRUE in $\mathcal{M}$, but there does not yet exist a sufficient flow in G^-, then find a maximum flow in G^+, and pick the first unassigned edge with non-zero flow to be assigned TRUE as the next decision. If no such edge exists, then pick the first unassigned edge capacity and assign its capacity to its flow G^+, as the next decision. In practice, such a flow is typically already discovered, during the evaluation of $maxFlow_{s,t}$ on G^+ during theory propagation.

D.1.6 Minimum Spanning Tree Weight

Here we consider constraints on the weight of the minimum spanning tree in a weighted (non-negative) undirected graph. For the purposes of this solver, we will define unconnected graphs to have infinite weight.

Monotonic Predicate: $\mathit{minSpanningTree}(E, m, c_0, c_1, \ldots)$, evaluates to TRUE iff the minimum weight spanning tree in the graph formed of edges E with edge weights given by $c_0, c_1, \ldots$, has weight $\leq m$.

Implementation of $\mathit{evaluate}(\mathit{minSpanningTree}(E, m, c_0, c_1, \ldots), G$**:** We use Spira and Pan's [187] fully dynamic algorithm for minimum spanning trees to detemrine the minimum weight spanning tree of G; return TRUE iff that weight is $\leq m$.

Implementation of $\mathit{analyze}(\mathit{minSpanningTree}, G^-, m^-, c_0^+, c_1^+, \ldots)$**:** The minimum weight spanning tree of G^- is less or equal to m^-, but $\mathit{minSpanningTree}$ is FALSE in $\mathcal{M}$. Let $e_0, e_1, \ldots$, with edge weights $w_0, w_1, \ldots$, be the edges of that spanning tree. Return the conflict set $\{(e_0 \in E), (w_0 \leq w_0^+), (e_1 \in E), (w_1 \leq w_1^+), (m \geq m^-), \neg \mathit{minSpanningTree}\}$.

Implementation of $\mathit{analyze}(\neg \mathit{minSpanningTree}, G^+, m^-, c_0^-, c_1^-, \ldots)$**:** The minimum weight spanning tree of G^+ is $> m^+$, but $\mathit{minSpanningTree}$ is TRUE in $\mathcal{M}$. There are two cases to consider:

1. If G^+ is disconnected, then we consider its weight to be infinite. In this case, we find a cut $\{e_1, e_2, \ldots\}$ of disabled edges separating any one component from the remaining components. Given a component in G^+, a valid separating cut consists of all disabled edges (u, v) such that u is in the component and v is not. We can either return the first such cut we find, or the smallest one from among all the components. For disconnected G^+, return the conflict set $\{(e_1 \notin E), (e_2 \notin E), \ldots, \mathit{minSpanningTree}(E, m)\}$.
2. If G^+ is *not* disconnected, then we search for a minimal set of edges required to decrease the weight of the minimum spanning tree. To do so, we visit each disabled edge (u, v), and then examine the cycle

that would be created if we were to add (u,v) into the minimum spanning tree. (Because the minimum spanning tree reaches all nodes, and reaches them exactly once, adding a new edge between any two nodes will create a unique cycle in the tree.) If any edge in that cycle has higher weight than the disabled edge, then if that disabled edge were to be enabled in the graph, we would be able to create a smaller spanning tree by replacing that larger edge in the cycle with the disabled edge. Let $e_0, e_1, \ldots$ be the set of such disabled edges that can be used to create lower weight spanning trees. Additionally, let $w_0, w_1, \ldots$ be the weights of the edges in the minimum spanning tree of G^+. Return the conflict set is $\{(e_0 \notin E), (e_1 \notin E), \ldots, (w_0 \geq w_0^-), (w_1 \geq w_1^-), (m \leq m^+), minSpanningTree(E,m)\}$.

In practice, we can visit each such cycle in linear time by using Tarjan's off-line lowest common ancestors algorithm [99] in the minimum spanning tree found in G^+. Visiting each edge in the cycle to check if it is larger than each disabled edge takes linear time in the number of edges in the tree for each cycle. Since the number of edges in the tree is 1 less than the number of nodes, the total runtime is then $O(|V|^2 \cdot |D|)$, where D is the set of disabled edges in M.

D.2 Geometric Predicates

D.2.1 Area of Convex Hulls

Monotonic Predicate: $hullArea_{\geq x}(S)$, $hullArea_{>x}(S)$, with S a finite set $S \subseteq S'$, true iff the convex hull of the points in S has an area $\geq$ (resp. $>$) than x (where x is a constant). This predicate is positive monotonic with respect to the set S.

Algorithm: Initially, compute $area(bound^-), area(bound^+)$. If $area(bound^-) > x$, compute $area(H^-)$; if $area(bound^+) < x$, compute $area(H^+)$. The areas can be computed explicitly, using arbitrary precision rational arithmetic.

Conflict set for $hullArea_{\geq x}(S)$: The area of H^- is greater or equal to x. Let

$p_0, p_1, \ldots$ be the points of S^- that form the vertices of the under-approximate hull H^-, with $area(H^-) \geq x$. Then at least one of the points must be disabled for the area of the hull to decrease below x. The conflict set is $\{(p_0 \in S), (p_1 \in S), \ldots, \neg hullArea_{\geq x}(S)\}$.

Conflict set for $\neg hullArea_{\geq x}(S)$: The area of H^+ is less than x; then at least one point $p_i \notin S^+$ that is not contained in H^+ must be added to the point set to increase the area of H^+. Let points $p_0, p_1, \ldots$ be the points of S' not contained in H^+. The conflict set is $\{(p_0 \notin S^+), (p_1 \notin S^+), \ldots, hullArea_{\geq x}(S)\}$, where $p_0, p_1, \ldots$ are the (possibly empty) set of points $p_i \notin S^+$.

D.2.2 Point Containment for Convex Hulls

Monotonic Predicate: $hullContains_q(S)$, true iff the convex hull of the 2D points in S contains the (fixed point) q.

Algorithm: First, check whether q is contained in the under-approximative bounding box $bound^-$. If it is, then check if q is contained in H^-. We use the PNPOLY [95] point inclusion test to perform this check, using arbitrary precision rational arithmetic, which takes time linear in the number of vertices of H^-. In the same way, only if q is contained in $bound^+$, check if q is contained in H^+ using PNPOLY.

Conflict set for $hullContains_q(S)$: Convex hull H^- contains p. Let p_0, p_1, p_2 be three points from H^- that form a triangle containing p (such a triangle must exist, as H^- contains p, and we can triangulate H^-, so one of those triangles must contain p; this follows from Carathodory's theorem for convex hulls [51]). So long as those three points are enabled in S, H^- must contain them, and as they contain p, p must be contained in *hunder*. The conflict set is $\{(p_0 \in S), (p_1 \in S), (p_2 \in S), \neg hullContains_q(S)\}$.

Conflict set for $\neg hullContains_q(S)$: p is outside of H^+. If H^+ is empty, then the conflict set is the default monotonic conflict set. Otherwise, by the separating axis theorem, there exists a separating axis between H^+ and p. Let $p_0, p_1, \ldots$ be the (disabled) points of S whose projection onto that axis is

$\geq$ the projection of p onto that axis[1]. At least one of those points must be enabled in S in order for H^+ to grow to contain p. The conflict set is $\{(p_0 \notin S), (p_1 \notin S), \ldots, hullContains_q(S)\}$.

D.2.3 Line-Segment Intersection for Convex Hulls

Monotonic Predicate: $hullIntersects_r(S)$, true iff the convex hull of the 2D points in S intersects the (fixed) line-segment r.

Algorithm: First, check whether line-segment r intersects $bound^-$. If it does, check if r intersects H^-. If H^- is empty, is a point, or is itself a line-segment, this is trivial (and can be checked precisely in arbitrary precision arithmetic by computing cross products following [110]). Otherwise, we check if either end-point of r is contained in H^-, using PNPOLY as above for point containment. If neither end-point is contained, we check whether the line-segment intersects H^-, by testing each edge of H^- for intersection with r (as before, by computing cross-products). If r does not intersect the under-approximation, repeat the above on $bound^+$ and H^+.

Conflict set for $hullIntersects_r(S)$: Convex hull H^- intersects line-segment r. If either end-point or r was contained in H^-, then proceed as for the point containment predicate. Otherwise, the line-segment r intersects with at least one edge (p_i, p_j) of H^-. Conflict set is $\{(p_i \in S), (p_j \in S), \neg hullIntersects_r(S)\}$.

Conflict set for $\neg hullIntersects_r(S)$: r is outside of H^+. If H^+ is empty, then the conflict set is the naïve monotonic conflict set. Otherwise, by the separating axis theorem, there exists a separating axis between H^+ and line-segment r. Let $p_0, p_1, \ldots$ be the (disabled) points of S whose projection onto that axis is $\geq$ the projection of the nearest endpoint of r onto that axis. At least one of those points must be enabled in S in order for H^+ to grow to contain p. The conflict set is $\{(p_0 \notin S), (p_1 \notin S), \ldots, hullIntersects_r(S)\}$.

[1] We will make use of the separating axis theorem several times in this section. The standard presentation of the separating axis theorem involves normalizing the separating axis (which we would not be able to do using rational arithmetic). This normalization is required if one wishes to compute the minimum distance between the projected point sets; however, if we are only interested in *comparing* distances (and testing collisions), we can skip the normalization step, allowing the separating axis to be found and applied using only precise rational arithmetic.

D.2.4 Intersection of Convex Hulls

Monotonic Predicate: $\mathit{hullsIntersect}(S_0, S_1)$, true iff the convex hull of the points in S_0 intersects the convex hull of the points of S_1.

Algorithm: If the bounding box for convex hull H_0 intersects the bounding box for H_1, then there are two possible cases to check for:

1. 1: A vertex of one hull is contained in the other, or
2. 2: an edge of H_0 intersects an edge of H_1.

If neither of the above cases holds, then H_0 and H_1 do not intersect.

Each of the above cases can be tested in quadratic time (in the number of vertices of H_0 and H_1), using arbitrary precision arithmetic. In our implementation, we use PNPOLY, as described above, to test vertex containment, and test for pair-wise edge intersection using cross products.

Conflict set for $\mathit{hullsIntersect}(S_0, S_1)$: H_0^- intersects H_1^-. There are two (not mutually exclusive) cases to consider:

1. A vertex of one hull is contained within the other hull. Let the point be p_a; let the hull containing it be H_b^-. Then (as argued above) there must exist three vertices p_{b1}, p_{b2}, p_{b3} of H_b^- that form a triangle containing p_a. So long as those three points and p_a are enabled, the two hulls will overlap. Conflict set is $\{(p_a \in S_a), (p_{b1} \in S_b), (p_{b2} \in S_b), (p_{b3} \in S_b), \neg \mathit{hullsIntersect}(S_0, S_1)\}$.
2. An edge of H_0^- intersects an edge of H_1^-. Let p_{1a}, p_{1b} be points of H_0^-, and p_{2a}, p_{2b} points of H_1^-, such that line segments $\overline{p_{1a}, p_{1b}}$ and $\overline{p_{2a}, p_{2b}}$ intersect. So long as these points are enabled, the hulls of the two point sets must overlap. Conflict set is $\{(p_{0a} \in S_0), (p_{0b} \in S_0), (p_{1a} \in S_1), (p_{1b} \in S_1), \neg \mathit{hullsIntersect}(S_0, S_1)\}$.

Conflict set for $\neg \mathit{hullsIntersect}(S_0, S_1)$: H_0^+ do not intersect H_1^+. In this case, there must exist a separating axis between H_0^+ and H_1^+. (Such an axis can be discovered as a side effect of computing the cross products of each edge in step 2 above.) Project all disabled points of S_0 and S_1 onto that axis. Assume (without loss of generality) that the maximum projected point of H_1^+ is less than the minimum projected point of H_1^+. Let $p_{0a}, p_{0b}, \ldots$ be the disabled points of S_0 whose projections are on the far side of the maximum projected point of H_1^+. Let $p_{1a}, p_{1b}, \ldots$ be the disabled points of S_1 whose projections are near side of the minimum projected point of H_1^+. At least one of these disabled points must be enabled, or this axis will continue to separate the two hulls. The conflict set is $\{(p_{0a} \notin S_0), (p_{0b} \notin S_0), \ldots, (p_{1a} \notin S_1), (p_{1b} \notin S_1), \ldots, hullsIntersect(S_0, S_1)\}$.

D.3 Pseudo-Boolean Constraints

Monotonic Predicate: $p = \sum_{i=0}^{n-1} c_i b_i \geq c_n$, with each c_i a constant, and b_i a Boolean argument.

Implementation of $evaluate(p = \sum_{i=0}^{n-1} c_i b_i \geq c_n, b_0, b_1, \ldots)$**:** Compute the sum of $\sum_{i=0}^{n-1} c_i b_i$ over the supplied arguments b_i, and return TRUE iff the sum $\geq c_n$. In practice, we maintain a running sum between calls to evaluate, and only update it as the theory literals b_i are assigned or unassigned.

Implementation of $analyze(p = \sum_{i=0}^{n-1} c_i b_i \geq c_n)$**:** Return the default monotonic conflict set (as described in Section 4.2).

Implementation of $analyze(\neg maxFlow_{s,t}, G^+, m^-, c_0^+, c_1^+, \ldots)$**:** Return the default monotonic conflict set (as described in Section 4.2).

D.4 CTL Model Checking

In Chapter 8, we describe a predciate for CTL model checking, along with associated functions. We describe theory propagation in that chapter, however the conflict analysis procedure for the theory of CTL model checking handles each CTL operator separately, and as a result was too long to include in the main body of Chapter 8. The complete procedure follows:

```
function ANALYZECTL($\phi, s, K^+, K^-, \mathcal{M}$)
    Let ($T^+$,$P^+$) = $K^+$
    Let ($T^-$,$P^-$) = $K^-$
    $c \leftarrow \{\}$
    if $\phi$ is $EX(\psi)$ then
        for each transition $t$ outgoing from $s$ do
            if $(t \notin T) \in \mathcal{M}$ then
                $c \leftarrow c \cup \{(t \in T)\}$.
        for each transition $t = (s,u)$ in $T^+$ do
            if $evaluate(\psi, u, K^+) \mapsto$ FALSE then
                $c \leftarrow c \cup$ ANALYZECTL$(\psi, n, K^+, K^-, \mathcal{M})$.
    else if $\phi$ is $AX(\psi)$ then
        Let $t = (s,u)$ be a transition in $T^-$, with $evaluate(\psi, u, K^+) \mapsto$ FALSE.
        (At least one such state must exist)
        $c \leftarrow c \cup \{(t \notin T)\}$
        $c \leftarrow c \cup$ ANALYZECTL$(\psi, u, K^+, K^-, \mathcal{M})$.
    else if $\phi$ is $EF(\psi)$ then
        Let $R$ be the set of all states reachable from $s$ in $T^+$.
        for each state $r \in R$ do
            for each transition $t$ outgoing from $r$ do
                if $(t \notin T) \in \mathcal{M}$ then
                    $c \leftarrow c \cup \{(t \in T)\}$
            $c \leftarrow c \cup$ ANALYZECTL$(\psi, r, K^+, K^-, \mathcal{M})$.
```

```
else if $\phi$ is $AF(\psi)$ then
    Let $L$ be a set of states reachable from $s$ in $T^-$, such that $L$ forms a lasso
from $s$, and such that for $\forall u \in L, evaluate(\psi, r, K^+) \mapsto$ FALSE.
    for each transition $t \in lasso$ do
        $c \leftarrow c \cup \{(t \notin T)\}$
    for each state $u \in L$ do
        $c \leftarrow c \cup$ ANALYZECTL$(\psi, u, K^+, K^-, \mathcal{M})$
else if $\phi$ is $EG(\psi)$ then
    Let $R$ be the set of all states reachable from $s$ in $T^+$, while only taking
transitions to states $v$ for which $evaluate(\psi, v, K^+) \mapsto$ TRUE.
    for each state $r \in R$ do
        for each transition $t = (r, v)$ outgoing from $r$ do
            if $(t \notin T) \in \mathcal{M}$ then
                $c \leftarrow c \cup \{(t \in T)\}$
            if $evaluate(\psi, v, K^+) \mapsto$ FALSE then
                $c \leftarrow c \cup$ ANALYZECTL$(\psi, v, K^+, K^-, \mathcal{M})\}$
else if $\phi$ is $AG(\psi)$ then
    Let $P$ be a path of transitions in $T^-$ from $s$ to some state $r$ for which
$evaluate(\psi, r, K^+) \mapsto$ FALSE.
    for each transition $t \in P$ do
        $c \leftarrow c \cup \{(t \notin T)\}$
    $c \leftarrow c \cup$ ANALYZECTL$(\psi, r, K^+, K^-, \mathcal{M})\}$
else if $\phi$ is $EW(\psi_1, \psi_2)$ then
    Let $R$ be the set of all states reachable from $s$ in $T^+$, while only taking
transitions to states $v$ for which $evaluate(\psi_1, v, K^+) \mapsto$ TRUE.
    for each state $r \in R$ do
        $c \leftarrow c \cup$ ANALYZECTL$(\psi_2, r, K^+, K^-, \mathcal{M})\}$
        for each transition $t = (r, v)$ outgoing from $r$ do
            if $(t \notin T) \in \mathcal{M}$ then
                $c \leftarrow c \cup \{(t \in T)\}$
                if $v \notin R$ and $evaluate(\psi_1, v, K^+) \mapsto$ FALSE then
                    $c \leftarrow c \cup$ ANALYZECTL$(\vee(\psi_1, \psi_2), v, K^+, K^-, \mathcal{M})\}$
```

```
else if $\phi$ is $AW(\psi_1, \psi_2)$ then
    Let $R$ be a set of states, for which there is path $P$ of transitions in $T^-$ from $s$ to some state $v$, such that each of the following conditions hold:
    1) $\forall r \in R, evaluate(\psi_2, r, K^+) \mapsto$ FALSE
    2) $\forall r \in R/\{v\}, evaluate(\psi_1, r, K^+) \mapsto$ TRUE
    3) $evaluate(\psi_1, v, K^+) \mapsto$ FALSE.
    for each transition $t \in P$ do
        $c \leftarrow c \cup \{(t \notin T)\}$
    $c \leftarrow c \cup$ ANALYZECTL$(\psi_1, v, K^+, K^-, \mathcal{M})\}$
    for each state $r \in R/\{v\}$ do
        $c \leftarrow c \cup$ ANALYZECTL$(\psi_2, r, K^+, K^-, \mathcal{M})\}$
else if $\phi$ is $EU(\psi_1, \psi_2)$ then
    Let $R$ be the set of all states reachable from $s$ in $T^+$, while only taking transitions to states $v$ for which $evaluate(\psi_1, v, K^+) \mapsto$ TRUE.
    for each state $r \in R$ do
        $c \leftarrow c \cup$ ANALYZECTL$(\psi_2, r, K^+, K^-, \mathcal{M})\}$
        for each transition $t = (r, v)$ outgoing from $r$ do
            if $v \notin R$ and $(t \notin T) \in \mathcal{M}$ then
                $c \leftarrow c \cup \{(t \in T)\}$
                if $evaluate(\psi_1, v, K^+) \mapsto$ FALSE then
                    $c \leftarrow c \cup$ ANALYZECTL$(\vee(\psi_1, \psi_2), v, K^+, K^-, \mathcal{M})\}$
else if $\phi$ is $AU(\psi_1, \psi_2)$ then
    if there exists a set of states $R$, for which there is path $P$ of transitions in $T^-$ from $s$ to some state $v$, such that each of the following conditions hold:
    1) $\forall r \in R, evaluate(\psi_2, r, K^+) \mapsto$ FALSE
    2) $\forall r \in R/\{v\}, evaluate(\psi_1, r, K^+) \mapsto$ TRUE
    3) $evaluate(\psi_1, v, K^+) \mapsto$ FALSE.
    then return ANALYZECTL$(AW(\psi_1, \psi_2), s, K^+, K^-, \mathcal{M})\}$
    else
    return ANALYZECTL$(AF(\psi_2), s, K^+, K^-, \mathcal{M})\}$
```

```
else if $\phi$ is $\vee(\psi_1, \psi_2)$ then
    $c \leftarrow c \cup \textsc{analyze}\mathrm{CTL}(\psi_1, s, K^+, K^-, \mathcal{M})\}$
    $c \leftarrow c \cup \textsc{analyze}\mathrm{CTL}(\psi_2, s, K^+, K^-, \mathcal{M})\}$
else if $\phi$ is $\wedge(\psi_1, \psi_2)$ then
    if $evaluate(\psi_1, s, K^+) \mapsto \textsc{False}$ and $evaluate(\psi_2, s, K^+) \mapsto \textsc{False}$ then
        Learn the smaller of the two conflict sets for $\psi_1, \psi_2$
        $c_1 \leftarrow \textsc{analyze}\mathrm{CTL}(\psi_1, s, K^+, K^-, \mathcal{M})\}$
        $c_2 \leftarrow \textsc{analyze}\mathrm{CTL}(\psi_2, s, K^+, K^-, \mathcal{M})\}$
        if $|c_1| \leq |c_2$ then
            $c \leftarrow c \cup c_1$
        else
            $c \leftarrow c \cup c_2$
    else if $evaluate(\psi_1, s, K^+) \mapsto \textsc{False}$ then
        $c \leftarrow c \cup \textsc{analyze}\mathrm{CTL}(\psi_1, s, K^+, K^-, \mathcal{M})\}$
    else
        $c \leftarrow c \cup \textsc{analyze}\mathrm{CTL}(\psi_2, s, K^+, K^-, \mathcal{M})\}$
else if $\phi$ is the negation of a property state $\neg p$ then
    $c \leftarrow c \cup \{\neg(p(s))\}$
else if $\phi$ is property state $p$ then
    $c \leftarrow c \cup \{(p(s))\}$
return $c$
```