SAT Modulo Monotonic Theories

Sam Bayless
sbayless@cs.ubc.ca
Univ. of British Columbia

Noah Bayless
nbayless@pgmini.org
Point Grey Mini Sec. School

Holger H. Hoos
hoos@cs.ubc.ca
Univ. of British Columbia

Alan J. Hu
ajh@cs.ubc.ca
Univ. of British Columbia

Abstract

Boolean satisfiability (SAT) solvers have been successfully applied to a wide variety of difficult combinatorial problems. Many further problems can be solved by SAT Modulo Theory (SMT) solvers, which extend SAT solvers to handle additional types of constraints. However, building efficient SMT solvers is often very difficult. In this paper, we define the concept of a *Boolean monotonic theory* and show how to easily build efficient SMT solvers, including effective theory propagation and clause learning, for such theories. We present examples showing useful constraints that are monotonic, including many graph properties (*e.g.*, shortest paths), and geometric properties (*e.g.*, convex hulls). These constraints arise in problems that are otherwise difficult for SAT solvers to handle, such as procedural content generation. We have implemented several monotonic theory solvers using the techniques we present in this paper and applied these to content generation problems, demonstrating major speed-ups over SAT, SMT, and Answer Set Programming solvers, easily solving instances that were previously out of reach.

1 Introduction

Boolean satisfiability (SAT) solvers have been demonstrated to be effective in solving NP-hard problems from many applications, including planning, scheduling and circuit design. SAT Modulo Theory (SMT) solvers, which extend SAT with additional types of variables and constraints, can handle an even broader range of problems.

As an example, consider procedural content generation, where one might wish to generate terrain subject to shortest path constraints (*e.g.*, to ensure that the exit from a videogame level is at a certain minimum distance from the entrance). SAT solvers perform poorly on shortest path constraints, motivating the development of an SMT solver for graph properties. However, designing efficient SMT solvers typically requires expertise and deep insight for each theory.

In this work, we identify a class of theories we call *monotonic*, for which we can create efficient SMT solvers either partially or entirely automatically. We show that many common problems can be tackled using these techniques, and build efficient SMT solvers that support efficient reasoning for many important graph constraints, including reachability, shortest paths, minimum spanning trees, maximum $s-t$ flow, and geometric constraints related to convex hulls. These properties are useful for procedural content generation tasks (*e.g.*, terrain and puzzle synthesis for videogames).

We implement an SMT solver for these theories in our tool MONOSAT,[1] showing order-of-magnitude speed-ups and greatly improved scalability over the SAT solver MINISAT 2.2 [Eén and Sörensson, 2004], the Answer Set Programming (ASP) solver CLASP [Gebser et al., 2007], and the SMT solver Z3 [De Moura and Bjorner, 2008].

2 Monotonic Theories

We define a Boolean monotonic predicate as follows:

Definition 1 (Boolean Monotonic Predicate) *A predicate $p : \{0,1\}^n \mapsto \{0,1\}$ is Boolean monotonic if, and only if, for all s_i:*

$$p(\ldots, s_{i-1}, 0, s_{i+1}\ldots) \rightarrow p(\ldots, s_{i-1}, 1, s_{i+1}\ldots)$$

We have given the definition for a positive monotonic predicate; an analogous definition exists for the negative case. Notice that the domain of p is restricted to Boolean values; monotonic functions over the Booleans are also known as *unate* functions. A similar definition for monotonic predicates has been previously explored by [Bradley and Manna, 2008] and by [Marques-Silva, Janota, and Belov, 2013], in the context of finding minimal models and unsatisfiable cores. We define a Boolean monotonic theory as:

Definition 2 (Boolean Monotonic Theory) *A theory T with signature Σ is Boolean monotonic if and only if:*

1. *The only sort in Σ is Boolean;*
2. *all predicates in Σ are monotonic; and*
3. *all functions in Σ are monotonic.*

We allow both positive and negative monotonic functions (and predicates). As is typical for SMT solvers, we consider only decidable, quantifier-free, first-order theories. Atypically for SMT, the only sort in these theories is Boolean.

[1] MONOSAT is open-source and freely available online, at `www.cs.ubc.ca/labs/isd/Projects/monosat`

This restriction allows us to make some simplifying assumptions (described in Section 3) that side-step complications arising from theory combination, compositions of functions, and non-ground variables; these are all issues that we expect could be addressed in future work. However, we will show that even restricted to the Booleans, monotonic theories can express many useful predicates.[2]

As an illustrative example, consider a theory of graph reachability, with predicates of the form $reach_{u,v,G}(edge_1, edge_2, edge_3, \ldots)$, which are true if, and only if, node u reaches node v in graph G, where edge i is included in G if and only if the Boolean $edge_i$ is true. Adding additional edges to G (by assigning edge variables to true) can make a previously unreachable node reachable, but cannot make a reachable node unreachable. Conversely, removing edges from G (by assigning edge variables to false) can make a reachable node unreachable, but cannot make an unreachable node reachable.

3 Theory Propagation and Clause Learning

Many successful SMT solvers follow the *lazy* SMT design [Sebastiani, 2007; De Moura and Bjorner, 2008], in which a SAT solver is combined with a set of theory solvers, and each theory solver supplies (at least) two capabilities: (1) theory propagation (or theory deduction), which takes a partial assignment M to the theory atoms for that theory and checks if any other atoms are implied by that partial assignment (or if M constitutes a conflict in the theory solver), and (2) clause learning (equivalently, deriving conflict or justification sets), where, given a conflict c in M, the theory solver finds a subset of M sufficient to imply c, which the SAT solver can then negate and store as a learned clause. The effectiveness of a lazy SMT theory solver depends on the ability of the theory solver to propagate atoms and detect conflicts early, from small assignments M, and to find small conflict sets in M when a conflict does arise.

Many theories have known, efficient algorithms for deciding the satisfiability of *fully* specified inputs, but not for partially specified or symbolic inputs. Continuing with the reachability example from above, given a concretely specified graph, one can find the set of nodes reachable from u simply using a breadth-first-search. In contrast, determining whether a node is reachable in a graph with symbolically defined edges is less obvious. We will show how to build an efficient theory solver (capable of both theory propagation and clause learning) for any Boolean monotonic theory, given only a procedure for computing the truth-values of the predicates from complete, concrete assignments.

First, we introduce a scheme for applying theory propagation to Boolean monotonic theories. For simplicity, we will assume that the instance has been transformed so as to convert all (Boolean monotonic) functions into Boolean monotonic predicates, and that all Boolean monotonic predicates are *positive* monotonic predicates. A simple, linear-time transformation can ensure this. Having done so, we can split the atoms of the formula into two sets: a set of Boolean monotonic predicates, P, and a set of Boolean variables that are arguments to those predicates (which we will expose to the SAT solver as the set of atoms S).

This simplified formula of the monotonic theory T now has the following useful properties: for any atom $p \in P$, and any partial truth assignment M_S to the atoms of the exposed Boolean variables S *excluding* variable $s \in S$,

$$\text{SAT}_T(M_S \cup \{\neg s, p\}) \Rightarrow \text{SAT}_T(M_S \cup \{s, p\}) \quad (1)$$

$$\text{SAT}_T(M_S \cup \{s, \neg p\}) \Rightarrow \text{SAT}_T(M_S \cup \{\neg s, \neg p\}) \quad (2)$$

Given a (partial) truth assignment M, let M_S be the corresponding (partial) assignment to just the S-atoms. We form two completions of M_S: one in which all the unassigned S-atoms are assigned to false (M_S^-), and one in which they are assigned to true (M_S^+). Since M_S^- contains a complete assignment to the S-atoms (which are the Boolean arguments to the monotonic predicates), we can use it as input to any standard algorithm for computing the truth-value of atom $p \in P$ from concrete inputs, which will determine whether $M_S^- \implies p$. By property (1), if $M_S^- \implies p$, then $M_S \implies p$. Similarly, by property (2), if $M_S^+ \implies \neg p$, then $M_S \implies \neg p$. If either case holds, we can propagate p (or $\neg p$) back to the SAT solver. Thus, M_S^- and M_S^+ allow us to safely under- and over-approximate the truth value of p. We can apply this technique iteratively for each P-atom.

This over/under-approximation scheme gives us theory propagation for P-atoms *only*; however, because there can be no theory-conflicts among the atoms of S by themselves (as each corresponds to an independent Boolean variable), applying propagation to the P-atoms is sufficient to detect any conflicting assignment. Furthermore, because all variables in the formula are included in the SAT solver's search space, this technique provides a complete procedure for deciding the satisfiability of Boolean monotonic theory T, so long as procedures are available for computing each monotonic predicate from complete assignments to their arguments. It allows us to use *standard* algorithms for solving the concrete, fully specified and non-symbolic forms of these theories, to apply theory propagation to the P-atoms.

What about clause learning? In general, given a conflict on a partial assignment M over the theory's atoms, a theory solver can always simply return the clause $\neg M$, which states that at least one of the assigned theory atoms must change in any satisfying assignment. However, for conflicts arising from the over/under-approximation theory-propagation scheme above, we can do better. First, as noted before, all assignments to just the S-atoms are satisfiable, so any conflicting assignment must assign at least one P-atom.

Secondly, observe that the over/under-approximations scheme computes implications from assignments of only the S-atoms M_S to individual P-atoms, and never computes implications from, for example, P-atoms to S-atoms or from P-atoms to other P-atoms. For this reason, in any conflict discovered by this scheme, there must exist a *single* P-atom p, such that the assignment to the S-atoms, along with the assignment to p, are together UNSAT. Note that this may not hold if other techniques are used to apply theory propagation from, for example, the P-atoms to the S-atoms.

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

By property (1), if $M_S^- \implies p$, then the positive theory atoms in M_S are sufficient to imply p by themselves. By property (2), if $M_S^+ \implies \neg p$, then the negative theory atoms in M_S are sufficient to imply $\neg p$ by themselves. Therefore, the positive (resp. negative) assignments to the atoms in M_S alone can safely form justification sets for p (resp. $\neg p$). This is already an improvement over simply learning the clause $\neg M$, but in many cases we can do better.

Many 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. For example, if we used a breadth-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 (all of which are assigned with positive polarity in M) imply that v can be reached from u.

Any algorithm that can produce a witness containing solely positive (resp. negative) S-atoms can be used to produce justification sets for any positive (resp. negative) P-atom assignments propagated by the scheme under-approximation above. We have usually found that standard algorithms produce one, and sometimes both, types of witnesses. For missing witnesses, we can always safely fall back on the strategy of using the positive (resp. negative) theory atoms of M_S as the justification set for p (resp. $\neg p$).

4 Examples of Monotonic Theories

We now introduce several monotonic predicates and show how efficient theory solvers can be built for each of them using the theory propagation and clause learning strategies from Section 3. First, we will consider some common graph properties, and then some geometric properties of pointsets. Section 5 presents applications and results for these theories.

We introduce a set of monotonic graph predicates, collected into a monotonic theory of graphs. Each graph predicate is defined for a graph with a finite set of vertices V and a finite set of *possible* edges $E \subseteq V \times V$. For each predicate, the solver will need to select a subset of the edges E to include in the graph G such that the predicate does or does not hold. Each predicate is defined for a fixed graph $G = (E, V)$ and will accept $|E|$ arguments: one Boolean variable for each edge in E, to select whether that edge is included in the graph or not. Each of those Boolean variables is exposed to the SAT solver by a corresponding theory atom $edge_{u,v,G}$. We will speak of edges being "enabled" or "disabled" as shorthand for the truth values of these $edge$ variables and corresponding atoms; if a formula has multiple predicates for the same graph G, we will assume that those predicates have as arguments the same edge variables in the same order (though this is not strictly necessary). The edge atoms form set S (as defined in Section 3), while the various graph predicates we consider will form set P.

For example, the graph reachability predicate $reach_{u,v,G}(edge_0, edge_1, edge_2, \ldots)$ is true if, and only if, node v is reachable from node u in graph G, under a given assignment to the $edge_i$ variables. As previously observed, given a graph (directed or undirected) and some fixed starting node u, enabling an edge can increase the set of nodes that are reachable from u, but cannot decrease it. Disabling an edge in the graph can decrease the set of nodes reachable from u, but cannot increase it. The other graph predicates we consider are monotonic with respect to the edges in the graph in the same way; for example, enabling an edge can decrease the weight of the minimum spanning tree, but not increase it.

To apply theory propagation under a partial truth assignment M, we construct two auxiliary graphs, G_{under} and G_{over}. The graph G_{under} is formed from the edge assignments in M_S^-: only edges that are assigned true in M are included in G_{under}; edges that are either assigned to false or are unassigned in M are excluded. In the second graph, G_{over}, we include all edges that are either assigned to true *or* unassigned in M, corresponding to the edge assignments in M_S^+. As G_{under} and G_{over} are completely specified, concrete graphs, we can then apply standard graph algorithms to them during theory propagation, using the over/under-approximation scheme described in Section 2.

A summary of the shortest paths and reachability theory solver can be found in the frame below. Two additional graph properties for which we have implemented monotonic theory solvers are maximum s–t flow (enabling edges can increase the maximum flow, but cannot decrease it), and minimum spanning trees (enabling an edge can decrease the minimum weight spanning tree, but cannot increase it).

The over/under-approximation theory propagation scheme makes repeated graph queries in the theory solver — one for each of G_{under} and G_{over}, for each new partial assignment generated by the SAT solver. The performance of this scheme is greatly improved by using dynamic graph algorithms and data structures, which can be cheaply updated as edges are removed or added to the two graphs. For computing shortest paths, we use [Ramalingam and Reps, 1996]; for computing minimum weight spanning trees we use [Spira and Pan, 1975]; while for maximum flow we use a variant of [Edmonds and Karp, 1972], modified to support adding edges incrementally, as described by [Korduban, 2012]. Deriving justification sets for these two latter graph predicates is more involved than for shortest paths; details can be found on arXiv [Bayless et al., 2014].

Three other generic improvements that we implement are worth mentioning. The first is that in cases where several predicates are being checked for the same graph, we can combine the data structures for the graphs, and (possibly) also combine the update checks for the predicates. For example, if many reachability queries were being computed for the same graph from the same starting node, a single breadth-first-search call could compute them all. A second improvement is to check whether, under the current partial assignment, either G_{under} or G_{over} is unchanged from the previous call to theory propagation. If the solver has only enabled edges (resp. only disabled edges) since the last theory propagation call, then the graph G_{over} (resp. G_{under}) will not have changed, and so we do not need to recompute properties for that graph. Finally, if a graph predicate is assigned in M, then we only need to check whether that assignment is violated; in this case, we can skip one of the two computations (for either G_{under} or G_{over}).

Graph Reachability & Shortest Paths

Given a finite, weighted, directed graph, the shortest path between fixed nodes u and v can decrease in length as edges are added, but cannot increase. If there is no $u - v$ path, we set the shortest path length to infinity; then we can compute the graph reachability predicate $reach_{u,v,G}(edges)$ (as described above) by testing whether the shortest $u - v$ path length is at most $\sum_{e \in E} weight_e$.

Monotonic Predicate:
$shortestPath_{u,v,G \leq C}(edges)$, true iff the shortest $u - v$ path $\leq C$ in G, given *edges*.

Algorithm: Ramalingam-Reps [Ramalingam and Reps, 1996], with improvements described by [Buriol, Resende, and Thorup, 2008]. This is a dynamic variant of Dijkstra's Algorithm [Dijkstra, 1959].

Justification for $shortestPath_{u,v,G \leq C}(edges)$:
Let $e_1, e_2, \ldots$ be the shortest $u - v$ path in G_{under}. The length of this path is at most C. The justification set is then $\{\neg e_1, \neg e_2, \ldots, shortestPath_{u,v,G \leq C}(edges)\}$.

Justification for $\neg shortestPath_{u,v,G \leq C}(edges)$:
Traverse the graph backwards from v in G_{over}, following each enabled or unassigned backward edge. Collect all incoming *disabled* edges $e_1, e_2, \ldots$ of the visited nodes. (If node u is visited during this traversal, neither follow nor collect u's incident edges.) The conflict set is $\{e_1, e_2, \ldots, \neg shortestPath_{u,v,G \leq C}(edges)\}$.

To illustrate a monotonic theory with a very different flavour, we now consider geometric theories involving continuous rather than discrete mathematics. Many common geometric properties of point sets monotonically increase (or decrease) as additional points are added. For example, the area of the convex hull of a point set can increase but cannot decrease as points are added to the set. Many other common properties of point sets are also monotonic; examples include the geometric span (*i.e.*, the maximum diameter), and the weight of the minimum Steiner tree of a point set, and the minimum distance between two point sets, but we restrict our attention to convex hulls (in 2 dimensions) here.

The implementation of our geometric solver is very close to the graph solver we described above, computing concrete under- and over-approximations H_{under}, H_{over} of the convex hull of the point set in the same manner as we did for G_{under} and G_{over} above. This solver can also be used to model collisions between the convex hull of a point set and a fixed point, line, or polygon by asserting the points of one of the pointsets. Details of the theory solver for collisions between convex hulls can be found in the frame below.

Collision Detection for Convex Hulls

Given two sets of points, PS_1 and PS_2, with two corresponding convex hulls, adding a point to either set can cause the corresponding hull to grow such that the two hulls overlap; however, if the two hulls already overlap, adding additional points to either set cannot make the hulls disjoint.

Monotonic Predicate:
$overlap_{PS_1,PS_2}(points_1, points_2)$, true iff the convex hull of the points in PS_1 enabled in the array of Booleans $points_1$ overlaps the convex hull of the points of PS_2 enabled in $points_2$.

Algorithm: Andrew's monotone chain algorithm [Andrew, 1979] for convex hulls.

Justification $overlap_{PS_1,PS_2}(points_1, points_2)$:
Convex hulls H_{under1} and H_{under2} overlap. There are two (not mutually exclusive) cases to consider:

1. Vertex p_a of H_{under1} is contained within H_{under2} (or vice versa). Then there must exist three vertices p_{b1}, p_{b2}, p_{b3} of H_{under2} that form a triangle containing p_a. So long as those three points (and p_a) are enabled, the two hulls will overlap. Justification set is $\{\neg p_a, \neg p_{b1}, \neg p_{b2}, \neg p_{b3}, overlap_{PS_1,PS_2}\}$.
2. An edge of H_{under1} intersects an edge of H_{under2}. Let p_{1a}, p_{1b} be points of H_{under1}, and p_{2a}, p_{2b} points of H_{under2}, 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. Justification set is $\{\neg p_{1a}, \neg p_{1b}, \neg p_{2a}, \neg p_{2b}, overlap_{PS_1,PS_2}\}$.

Justification $\neg overlap_{PS_1,PS_2}(points_1, points_2)$:
Convex hulls H_{over1} and H_{over2} do not overlap. There must exist a separating hyperplane Q between H_{over1} and H_{over2}. Let $p_{1a}, p_{1b}, \ldots$ be the disabled points of PS_1 on the far side of Q (or exactly on Q) from H_{over1}; let $p_{2a}, p_{2b}, \ldots$ be the disabled points of PS_2 that are on the far side of Q (or on Q) from H_{over2}. At least one of these points must be enabled, or this hyperplane will continue to separate the hulls. Justification set is $\{p_{1a}, p_{1b}, \ldots, p_{2a}, p_{2b}, \ldots, \neg overlap_{PS_1,PS_2}\}$.

One important consideration of geometric properties is numerical precision. We restrict the coordinates of points to rationals, and use arbitrary precision rational arithmetic throughout our geometric solver. For efficiency, we handle intersection with fixed shapes as special cases in our implementation, but in the interest of space we present only the general case here.

We also include two additional improvements in our geometry solver. First, when detecting whether a point is contained in either the under- or over-approximation convex hull, or whether the hull intersects another polygon, we initially compute axis-aligned bounding boxes, and use those to cheaply eliminate many collision detections. Secondly, when a collision between two hulls is detected, we find a small (not necessarily minimal) set of points which are sufficient to guarantee that overlap. For example, when a point is found to be contained in a convex hull, we find three vertices of that hull that together form a triangle that contains that point (such a containing triangle is guaranteed to exist by Carathéodory's theorem [Eckhoff and others, 1993] for convex hulls). While the points composing that triangle remain in the hull, even if other points are removed from the hull the point will remain contained. This can be checked very cheaply, sometimes allowing us to skip collision checks.

A wide body of techniques exists 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 sophisticated implementation could make use of more efficient data structures (such as hierarchical bounding volumes or trapezoidal maps) to obviate many of the collision checks. As before, because collision detection is performed on concrete under- and over-approximation hulls, standard algorithms and libraries may be used.

5 Applications and Results

Many popular videogames — including the hits *Dwarf Fortress* and *Minecraft* — rely on procedurally generated content. Recently, there has been interest in using logic programming for *declarative* content generation (e.g., [Boenn et al., 2008; Nelson and Smith, 2014]), in which the artifact to be generated is specified as the solution to a formula. These approaches are typically less scalable than traditional procedural content generation, however, they make it convenient to guarantee important properties of the content, such as that all points are reachable in generated terrain.

To demonstrate our graph theory solver, we add shortest path and maximum flow constraints to the open-source, videogame terrain generation tool *Diorama* [Schanda and Brain, 2009]. Diorama considers a set of undirected, planar edges arranged in a grid. Each position on the grid is associated with a height; the solution to the constraints is a heightmap that realises a complex combination of desirable characteristics, such as the positions of mountains, water, cliffs, and players' bases. Edges from this grid are only included in the graph where the heightmap does not have a sharp elevation change or impassable zone (*e.g.*, water).

Diorama expresses its constraints via Answer Set Programming (ASP) [Baral, 2003] — a logic formalism closely related to SAT, and with efficient solvers [Gebser et al., 2007] based on state-of-the-art CDCL SAT solvers. Unlike SAT, ASP can encode reachability constraints in cyclic graphs in linear space and can solve the resulting formulae efficiently. Partly for this reason, ASP solvers have been more widely used than SAT solvers in recent content generation applications; *Refraction* [Smith and Mateas, 2011] and *Variations Forever* [Smith and Mateas, 2010] also use ASP.

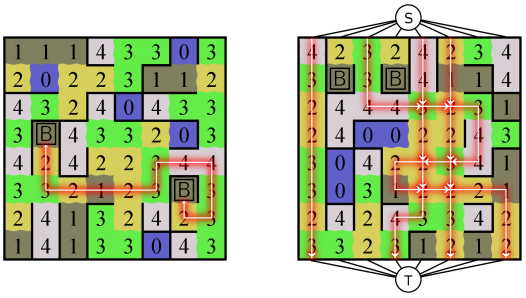

Figure 1: Diorama heightmaps generated by MONOSAT. Numbers correspond to elevations (bases are marked 'B'), with impassable cliffs marked in black. On the left, a distance constraint between bases; on the right, a maximum s–t flow constraint between the top and bottom of the map.

Shortest Paths	MONOSAT	CLASP	MINISAT
8x8,Distance 8-16	<1s	<1s	9s
16x16,Dst.16-32	4s	7s	>3600s
16x16,Dst.32-48	4s	23s	2096s
16x16,Dst.32-64	4s	65s	>3600s
16x16,Dst.32-96	4s	>3600s	>3600s
16x16,Dst.32-128	4s	>3600s	>3600s
24x24,Dst.48-64	46s	30s	>3600s
24x24,Dst.48-96	61s	1125s	>3600s
32x32,Dst.64-128	196s	>3600s	>3600s

Maximum Flow	MONOSAT	CLASP	MINISAT
8x8, Flow 16	2s	2s	1s
16x16, Flow 8	9s	483s	>3600s
16x16, Flow 16	8s	27s	>3600s
16x16, Flow 24	14s	26s	>3600s
24x24, Flow 16	81s	>3600s	>3600s
32x32, Flow 16	450s	>3600s	>3600s

Tables 1 & 2: Diorama runtimes, extended with varying shortest-path and maximum flow constraints. We can see that for small shortest path and maximum flow constraints, CLASP and MONOSAT both perform well, while for large constraints, MONOSAT greatly outperforms CLASP.

We provide comparisons of our solver MONOSAT (based on the SAT solver MINISAT 2.2) against MINISAT and the ASP solver CLASP 3.10 (for the graph predicates) and against the SMT solver Z3 4.3.1 [De Moura and Bjorner, 2008] (for the geometric predicates, which rely on arbitrary precision arithmetic and cannot be concisely encoded in SAT or ASP). Experiments were conducted on an Intel X5650 CPU, 2.67GHz (12MB L3), in Ubuntu 12.04, 64-bit, with 16 GB RAM, limited to 3600 seconds (walltime).[4]

One important factor in this comparison is how the various graph and geometry constraints are encoded into SAT, ASP, and (for Z3), the theory of linear rational arithmetic

[4]CLASP comes with a set of pre-built configurations to choose from; we ran all of them and in each case report the best results.

(LRA). The obvious encodings for shortest path constraints and maximum flow, into both SAT and ASP, encode distances (and flows) in unary, and are $\mathcal{O}(|V| \cdot |E|)$; however, by encoding distances in binary, an $\mathcal{O}(\log(|V|) \cdot |E|)$ encoding is possible. The unary encodings effectively unroll the Bellman-Ford algorithm [Bellman, 1956] and allow the solver to directly compute the shortest paths in a fixed graph using unit-propagation. In contrast, the binary encodings for shortest paths require the solvers to guess the distances to each node non-deterministically, introducing a search problem. For shortest paths, we show only results for the unary encodings, as we found that the more concise binary encoding always performed very poorly. For maximum flows, both encodings require the solver to guess flows non-deterministically, and here the binary encodings perform better (and we present those results). For the SMT convex-hull encoding, we used an encoding that non-deterministically guesses separating axes between hulls; it compares all pairs of points and requires quadratic space.[5]

Shortest Paths: We considered a modified version of the Diorama terrain generator, replacing Diorama's existing constraint limiting the number of cliffs with a new constraint that the distance (as the cat runs, not as the crow flies) between bases fall within certain ranges. Table 1 shows that while CLASP performs as well or better on small constraints, MONOSAT performs much better on larger constraints.

Maximum-Flow: The theory of maximum s–t flows/minimum s–t cuts can be used to create, or to prevent, chokepoints in a map, and be used, for example, to control traffic flow or create defensible locations. We assign each edge a fixed edge capacity (in this case, 4), and modify the Diorama constraints to enforce that the minimum cut between the top and bottom of the map must be exactly equal to a fixed multiple of that edge capacity (*e.g.*, 8,16, or 24). This requires chokepoints of a certain size between the top and bottom of the map. Table 2 shows that, as with shortest paths, MONOSAT outperforms CLASP and MINISAT on moderate-to-large instances.

Convex Hulls: To demonstrate our theory of convex hulls of pointsets, we consider the problem of synthesizing "art galleries", subject to constraints. The Art Gallery problem [Chvatal, 1975] is a classic NP-hard problem [Lee and Lin, 1986], 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; we consider a common (still NP-hard) variant in which cameras are restricted to being placed on vertices, and only the vertices of the room must be watched (but not the walls between the vertices). We define the Art Gallery Synthesis problem to be to *design* a floor plan that can be completely surveilled by at most a given number of cameras, subject to aesthetic constraints on the floor plan. One example application for this problem would be to construct mazes or levels that can be guarded by a specified number of computer-controlled characters (while also respecting other design constraints).

[5]We are deeply indebted to Adam M. Smith for suggesting improved encodings for shortest paths, and providing the maximum-flow encoding for ASP. While we have made a significant effort to find good encodings, we do not claim that the ASP, CNF, or SMT encodings we compare against are optimal – in general, there are no known proofs of optimal encodings for these constraints.

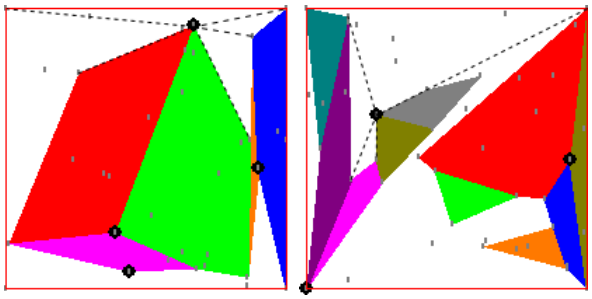

Figure 2: Artificial 'art galleries' synthesized by MONOSAT. White-space is open floor, while colored regions are walls. Cameras are large black circles; potential vertices are gray dots. Each polygon is the convex hull of a subset of the gray dots, selected by the solver. The cameras have been placed such that the vertices of all polygons can be seen, some along very tight angles or only from one side (sight lines have been drawn in dashed lines for two cameras). Sometimes cameras are placed to cover a single vertex that is completely embedded in adjacent polygons.

Art Gallery Synthesis	MONOSAT	Z3
10 points, 3 hulls, $\leq$3 cameras	2s	7s
20 points, 4 hulls, $\leq$4 cameras	36s	433s
30 points, 5 hulls, $\leq$5 cameras	187s	$>$ 3600s
40 points, 6 hulls, $\leq$6 cameras	645s	$>$ 3600s
50 points, 7 hulls, $\leq$7 cameras	3531s	$>$ 3600s

Table 3: Art gallery synthesis results.

Formally, given a box containing a fixed set of (2 dimensional) points, we must find N non-overlapping convex polygons with endpoints selected from those points, such that a) the area of each polygon is greater than some fixed constant (to prevent unrealistically slim walls 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 N cameras (placed at those vertices).

There are many ways that one could constrain the art gallery synthesis problem; we chose these constraints mostly to exhibit all of the features of our convex hull theories, but also to produce visually interesting and sufficiently complex galleries; however, these are not intended to be realistic instances. Real-world applications would likely combine these with more complex constraints (interesting possibilities would be to incorporate safety regulations or crowd-flow optimizations into the floor plan constraints). For the constraints we considered here, Table 3 shows that MONOSAT is able to solve much larger instances than Z3.

6 Conclusion

We have introduced the concept of a Boolean monotonic theory and provided a systematic technique to build efficient SAT Modulo Monotonic Theory (SMMT) solvers incorporating such theories. Our technique leverages commonplace, highly efficient algorithms for fully specified problem instances, in order to achieve efficient theory propagation and clause learning. We demonstrated the generality of the monotonic theory concept by providing several predicates drawn from graph theory and geometry. These example theories are expressive — permitting compact encodings for problems arising from procedural content generation and geometry — and the SMT solvers we produce using the techniques described in this paper perform well in practice on a range of instances, scaling much better to large instances than other solvers.

Moreover, the techniques we have introduced should generalize well beyond the small number of graph and geometric properties discussed in this paper. In particular, many additional graph predicates of interest are Boolean monotonic, including constraints on graph planarity, graph connectivity, graph diameter, global minimum-cut size, and many variants of network flow constraints. We expect each of these to yield efficient solvers following the techniques in this paper.

Recent work introduced SAT and ASP solvers with support for detecting acyclicity in graphs [Gebser, Janhunen, and Rintanen, 2014a; 2014b]. To the best of our knowledge, this is the only other major work on dedicated graph SMT solvers, outside of the general purpose graph solving capabilities of ASP solvers discussed above. Acyclicity is Boolean monotonic with respect to the edges of a graph, and so could be supported by our monotonic theory solver techniques (although we have not yet implemented such a theory solver). Although the techniques introduced by Gebser, Janhunen, and Rintanen are substantially different from ours, we believe they are compatible with ours, and we look forward to incorporating their techniques in the future. Conversely, SMMT is a general framework that could provide an avenue to extend their work beyond just acyclicity predicates to the large set of graph theories we have introduced.

There are several ways we can envision relaxing the restriction to Boolean sorts, for example to allow monotonic predicates over the integers or reals; doing so would allow a much wider set of useful theories to be handled (and allow a more natural presentation of the theories in this paper), but comes with a number of challenges (in particular, for supporting theory combination).

The most immediate direction for future work, however, is to discover additional Boolean monotonic theories and new application domains that can benefit from them.

7 Acknowledgments

As mentioned above, we gratefully acknowledge Adam M. Smith's assistance and insight on the subject of Answer Set Programming. Our research has been supported by the use of computing resources provided by WestGrid and Compute/Calcul Canada, and by funding provided by the Natural Sciences and Engineering Research Council of Canada.

References

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

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

Bayless, S.; Bayless, N.; Hoos, H. H.; and Hu, A. J. 2014. SAT modulo monotonic theories. *arXiv preprint arXiv:1406.0043*.

Bellman, R. 1956. On a routing problem. Technical report, DTIC Document.

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

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

Buriol, L. S.; Resende, M. G.; and Thorup, M. 2008. Speeding up dynamic shortest-path algorithms. *INFORMS Journal on Computing* 20(2):191–204.

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

De Moura, L., and Bjorner, N. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 337–340.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1(1):269–271.

Eckhoff, J., et al. 1993. Helly, Radon, and Carathéodory type theorems. *Handbook of convex geometry* 389–448.

Edmonds, J., and Karp, R. M. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)* 19(2):248–264.

Eén, N., and Sörensson, N. 2004. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, 333–336. Springer.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. clasp: A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning*. Springer. 260–265.

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

Gebser, M.; Janhunen, T.; and Rintanen, J. 2014b. SAT modulo graphs: Acyclicity. In *Logics in Artificial Intelligence*. Springer. 137–151.

Korduban, D. 2012. Incremental maximum flow in dynamic graphs. Theoretical Computer Science Stack Exchange. http://cstheory.stackexchange.com/q/10186 (version: 2012-02-18).

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

Marques-Silva, J.; Janota, M.; and Belov, A. 2013. Minimal sets over monotone predicates in Boolean formulae. In *Computer Aided Verification*, 592–607. Springer.

Nelson, M. J., and Smith, A. M. 2014. ASP with applications to mazes and levels. In Shaker, N.; Togelius, J.; and Nelson, M. J., eds., *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

Ramalingam, G., and Reps, T. 1996. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms* 21(2):267–305.

Schanda, F., and Brain, M. 2009. The Warzone map tools: Diorama.

Sebastiani, R. 2007. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)* 3:141–224.

Smith, A. M., and Mateas, M. 2010. Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, 273–280. IEEE.

Smith, A. M., and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *Computational Intelligence and AI in Games, IEEE Transactions on* 3(3):187–200.

Spira, P. M., and Pan, A. 1975. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing* 4(3):375–380.