An SMT Algorithm for Direct Model Search: Implementation and Experimentation

Scott Cotton and Oded Maler

Verimag, Centre Équation
2, Ave. de Vignate
38610 Gières, France

Abstract. SMT solvers have traditionally been based on the DPLL(T) algorithm, where the driving force behind the procedure is a DPLL search over truth valuations. This traditional framework allows for a degree of modularity in the treatment of theory solvers. Over time, theory solvers have become more and more closely integrated into the DPLL process, and consequently less and less modular. In this paper, we present a DPLL-like algorithm for SMT solving which unifies theory solving and DPLL search into a single process, directly searching for a model over the space of variable valuations. As a case study, we analyze its application to continuous domain linear arithmetic, present implementation techniques and some experimention with difference logic.

1 Introduction

SMT solvers have traditionally been based on the DPLL(T) [GHN+04] algorithm, where the driving force behind the procedure is a DPLL search over truth valuations. This traditional framework allows for a degree of modularity in the treatment of theory solvers. Over time, theory solvers have become more and more closely integrated into the DPLL process with such techniques as theory propagation, theory driven case splitting, and theory learning. These techniques generally correspond to the introduction of new proof rules, and hence non-determinism, in the underlying proof system. Consequently, SMT solvers which incorporate these techniques face the problem of deciding when to apply them. In general, increased non-determinism in the solving process presents a corresponding need for more effective heuristics to guide the solving process. Unfortunately, such heuristics are not well understood, and in fact are often static, *ad hoc*, theory specific, and tuned to artificial benchmarks.

In this paper, we present a DPLL-like algorithm for SMT solving which unifies theory solving and DPLL search into a single process, directly searching for a model over the space of variable valuations, rather than searching piecewise for theory models of (partial) propositional models. The goal of this effort is to find a formalism which supports more dynamic and principled heuristics. In particular, by searching over variable valuations we are able to make use of VSID [MMZ+01] variable ordering heuristics found in modern SAT solvers but applied to arbitrary variables; and by defining a notion of progress over the search space, we are able to determine which learned theory literals are relevant to progress,

thus giving a principled and dynamic heuristic for the creation and maintenance of a set of theory literals.

However, our algorithm also introduces problematics not found in the traditional DPLL(T) based techniques. In particular, theories with unbounded derivations introduce strong restrictions on the algorithm if completeness is desired. Additionally, achieving space efficiency is more involved, and there are various problematics at the implementation level. Thus our algorithm is by no means a panacaea and in fact only outperforms traditional methods on a fairly small portion of the benchmarks we evaluated. Nonetheless, we find the abstract formulation, analysis, and implementation-level solutions interesting and potentially useful to the design of future solvers.

1.1 Organization

The rest of this paper is organized as follows. We present background material in Section 2 and our abstract direct search algorithm in Section 3. Section 4 presents generic implementation problems and solutions. Section 5 presents an application to linear real arithmetic. Experiments on the sub-theory of difference logic problems are presented in 6. Section 7 concludes.

2 Background

2.1 Logical Background

SMT solving addresses the problem of deciding formulas in a fragment of first order logic. Syntactically, a formula consists of a set of symbols, which we characterize as either variables, logical symbols ($\vee, \wedge, \neg, \forall, \exists, =$), or non-logical symbols such as constants (such as $0, 1$), relation symbols (such as $<$), or function symbols (such as $+, \cdot$). Given a mapping of variables to values α, we denote by $\phi[\alpha]$ the result of replacing each variable x in ϕ with $\alpha(x)$.

A *signature* is a set of non-logical symbols. A *structure* may be associated with a signature and consists of set referred to as the *domain* together with an *interpretation* for each non-logical symbol in the signature. The interpretation of a constant is just an element of the domain. The interpretation of an n-ary relation is a subset of D^n where D is the domain. Likewise, the interpretation of an n-ary function is a subset of $D^n \to D$. A *model* of a variable free formula ϕ is a structure for some signature which includes all the non-logical symbols in ϕ and which makes ϕ true[1]. A variable-free formula is *satisfiable* if there exists a model for it; we say that the model *satisfies* the formula. If a formula contains variables, one may simply consider the variables as extra constant symbols, in which case the above definition of satisfiability readily applies. A model for a set of formulas S is a model which satisfies every $s \in S$.

In the following, an *atom* or *atomic predicate* is a formula whose root symbol is either a relation symbol or an equality. In other words, an atom corresponds to constraint which contains no Boolean connectives, such as $2x < y$. A *literal* is

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

either an atom or its negation. As in propositional logic, a *clause* is a disjunction of literals. A *theory* T is a set of formulas. A formula ϕ is satisfiable modulo a theory T if there is a model which satisfies both ϕ and T.

2.2 Related Work

Most SMT solvers are based on DPLL(T), which can be summarized as a state transition system over states which are defined by defined by pairs of sets of clauses and partial truth assignments to atoms. The basic system allows extending a partial truth assignment, learning/forgetting clauses and backtracking, just as in a conflict driven clause recording SAT solver. These rules are extended to allow clause learning based on T-inconsistent sets of literals and to allow extension of a partial truth assignment α to include theory literals which are theory-consequences of the conjunction of literals defined by α. This basic model, summarized in [BSST09] is centered around the notion of a search over truth assignments.

Extensions to the basic DPLL(T) proof system allow for arbitrary generation of theory literals, which in turn allows the proof system to branch on the associated truth values. In its most general form, this extension may be used to simulate our algorithm. Various restricted instantiations [BDdM08,dMB08] of this extension exist, but none views the solving process as a search on variable valuations.

Our work is most closely related to GDPLL and GDPLL-QFLRA [MKS09]. On the abstract level, our algorithm presents a more concrete form than GDPLL by making variable valuations explicit. This in turn allows us to define a semantic notion of progress in terms of the search space and leads to the ability to forget learned clauses. Our work on linear arithmetic uses the same proof rule as GDPLL-QFLRA, but adds the ability to forget learned clauses and incorporates dynamic variable ordering – at the expense of either efficiency or completeness.

3 Unate Consistent Direct Model Search

In this section, we introduce our abstract SMT algorithm, which we call *unate consistent search* (UCS), since it is based on 1-variable local consistency. To describe UCS, consider a quantifier free CNF formula ϕ

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

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

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

That is, $\phi|_x$ is the set of all x-unate clauses. We say ϕ is *unate consistent* if for every variable $x \in vars(\phi)$ it is the case that $\exists x.\phi|_x$. Unate consistency is simply variable-local consistency. Similarly, given an assignment α, we denote by $\phi|_{\alpha,x}$ the set of all clauses $c \in \phi$ such that $vars(c[\alpha]) = \{x\}$.

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

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

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

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

1. $\phi|_{\alpha,x} \models w$
2. $w[\alpha]$ contains no variables and evaluates to false.

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

Fig. 1. Unate Consistent Search Algorithm

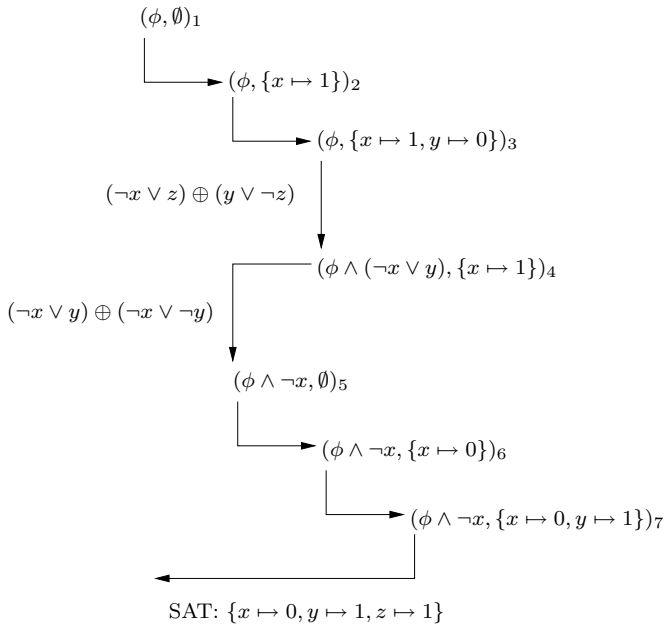
```
   UC-Search(φ, α)
1    if isUC(φ[α]) then
2       let (x, a) = select(φ, α)
3       if vars(φ[α]) = {x} then
4          return (1, α ∪ {x ↦ a})
5       let (r, w) = UC-Search(φ, α ∪ {x ↦ a})
6       if r = 1 or x ∉ vars(w) then
7          return (r, w)
8       else
9          return UC-Search(φ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

much like DPLL. It branches in a depth first fashion over variable valuations and learns clauses when it encounters conflicts. However, `UC-Search` recurses on assignment extensions as well as learned clause extensions. Also, here variables which are constrained are not automatically assigned. These facts highlight that `UC-Search` is in some ways fundamentally different than DPLL.

Example 1 (`UC-Search` *run).* Consider the propositional formula ϕ

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

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



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

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

Theorem 1. `UC-Search` *Soundness*

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

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

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

Theorem 2. `UC-Search` *Progress*
Let $U(\phi, \alpha)$ *denote the set of all unate-feasible* 1*-extensions of* α

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

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

Proof. see Appendix A.1

Progress is an analog of the termination argument for a DPLL solver which learns asserting clauses [ZM03]. Of course, one cannot guarantee termination without specifying something more about the theory or the resolution procedure. However, progress is more subtle in UCS because of the fact that when a variable is unate constrained it may or may not be assigned under α. This situation leads to the possibility of the procedure cycling with respect to a partial assignment. Progress simply says that whenever this happens, the procedure is in a state in which the search space is properly constrained with respect to the variable order in which the variables are assigned. Note that progress takes place even though `UC-Search` forgets clauses from deeper in the call stack. Thus progress allows a solver to safely forget clauses, but still there is no limit on the minimum number of clauses necessary to guarantee progress because many learned unate clauses can be recorded under a given assignment before the procedure backtracks over that assignment. Also note that as stated, progress requires that upon learning a clause w, `UC-Search` backtracks to the *maximal* assignment under which w is unate. A similar notion of progress holds if `UC-Search` backtracks to the minimal assignment, which we omit for simplicity.

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

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

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

To better address this issue, we introduce the notion of the proof graph traced by `UC-Search` in terms of the input-output relation of the procedure `resolve`. Consider a call

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

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

label each edge in this graph with x, referred to as the *pivot variable*. As in propositional resolution, the proof graph is *regular* if for every path in the graph, the corresponding sequence of pivot variables contains each variable at most once. The proof graph is *tree-like*, if each instance of a derived clause[2] can be the antecedent of at most one other derived clause.

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

Definition 1 (Exhaustively Asserting). *We say that* `UC-Search` *is* exhaustively asserting *if any variable chosen after a conflict is the variable constrained by the last learned clause.*

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

Theorem 3 (Restricted Resolution). *If* `UC-Search` *is exhaustively asserting then the proof graph traced by* `UC-Search` *is tree-like and regular.*

Proof. see Appendix A.2.

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

Definition 2 (Finite Width). *Given a finite set of clauses ϕ and a distinguished variable $x \in vars(\phi)$, let*

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

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

Theorem 4. *Termination Sufficiency*
`UC-Search` *terminates for a CNF formula ϕ if there is some set of clauses $P \supset \phi$ such that*

1. *P is closed under* `resolve`*;*

[2] Derived clauses can appear multiple times in the graph. An instance of a derived clause corresponds to a particular call to `resolve`.

2. `resolve` *has finite width for* P*; and*
3. `UC-Search` *is exhaustively asserting.*

Proof. Having established progress (Theorem 2), it will suffice to show that the set of learned clauses is finite. Since `resolve` has finite width it will suffice to show that every learned clause falls in the k-closure of `resolve` for some bound k. By Theorem 3, the proof graph is regular, and so for a formula of n variables, every learned clause falls in the n-closure of `resolve`. □

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

3.1 Comparison to DPLL

Space Requirements Perhaps the most important aspect of DPLL based solvers in comparison to other methods in propositional reasoning is the fact that such solvers can be space efficient. While solvers often do make use of a lot of space, they have the capacity to operate with very limited space. This allows one to control the amount of used space in an effort to find solutions quickly, for example by effective clause garbage collection. The unate consistent search algorithm provides a hint of how one may accomplish space efficiency in a given instantiation but it provides no such guarantee. In particular, Theorem 2 demonstrates that the algorithm makes progress even though it forgets learned clauses. Nonetheless UCS provides no bounds on the number *or size* of clauses.

Learning and Branching Modern DPLL solvers benefit largely not only from learning clauses, but also from the careful construction of learned clauses. In particular, conflict analysis attempts to find a succinct reason behind a dead end in the search, making use of global topologic properties in the implication graph, such as unique implication points and self-subsumption minimal clauses. In the UCS algorithm, clause resolution steps are abstracted away and are determined on the fly during backtracking. Thus it is not possible to generalize these important aspects of DPLL clause learning to UCS.

More importantly, DPLL solvers can make use of unconstrained resolution without sacrificing termination because only a finite number of clauses can be derived. With UCS, if resolution is unbounded, then termination is only guaranteed under severely restricted forms of resolution. In particular, the formal UCS algorithm implements tree-like, regular resolution, and our proof of termination relies on regularity of resolution. However, the only straightforward ways of implementing regular resolution is either tree-like, with the formal UCS algorithm and dynamic variable orderings, or with clause recording and a fixed variable ordering (directed resolution). Both of these forms of resolution are severely restrictive.

3.2 Discussion

Much work has been done generalizing DPLL to richer logics [BSU97,BT03] [BvdPTZ07,MKS09]. Unlike many generalizations of DPLL, UCS does not case split on truth values of atoms, but rather on variable valuations. GDPLL [MKS09] does not specify how an instantiation will case split, but, like UCS, GDPLL-QFLRA [MKS09] searches over variable valuations for the case of linear arithmetic. UCS generalizes the notion of search over variable valuations, providing termination conditions based on the topology of the proof graph. Most importantly, UCS provides a straightforward means to use dynamic variable orderings in the same spirit as SAT solvers and provides a notion of progress which identifies which learned clauses are relevant to progress. Like GDPLL, UCS learns clauses in response to conflicts rather than in response to partial satisfying assignments, as is the case with theory propagation. As a result, relevance to the search process is guaranteed for all learned facts. Unfortunately, these features are costly for theories with unbounded resolution.

4 Implementation

The `UC-Search` algorithm introduces some implementation problems not present in `DPLL` sat solvers. First, we must implement per-variable consistency checks. Second, `UC-Search` uses the call stack to push and pop *clauses* as well as variable assignments. As a consequence, a non-recursive implementation is essential because the stack depth is not bound to the number of variables. Third, learned clauses are not always false under the current assignment, leading to a situation where the procedure must be able to interleave backtracking with consistency checks. In this section, we present our solutions to these problems.

Consistency checks occur incrementally on a per-variable basis. Whenever a clause becomes unate for some variable x, whether as a result of an assignment or as a result of learning, a consistency check occurs for the the variable x. We would like to allow per-variable consistency checks to occur incrementally and in a backtrack-friendly fashion. We accomplish this in a way which also addresses the issue of non-recursive stacking of clauses. Thus, we would like to maintain for each variable the unate constraints placed upon it at any time in the search process, with the ability to add constraints incrementally and remove them upon backtracking, possibly forgetting them if they are learned clauses. To accomplish this in such a way that only unate clauses need to be updated during backtracking, we maintain an index as depicted and described in Figure 2. The constraint index also facilitates backtracking. Backtracking occurs immediately after a call to `resolve`. Backtracking is then a function of the current state of the constraint index and a newly learned clause. Backtracking occurs in per-variable units of work. As each variable x is unassigned, all its outgoing constraints are popped from x's outgoing stack and the constrained variable's incoming stack. This occurs until the newly learned clause w is unate. Once it is unate, a consistency check occurs. If the check succeeds, the procedure stops backtracking and passes control to the `select` function. If the check fails, `resolve` is called

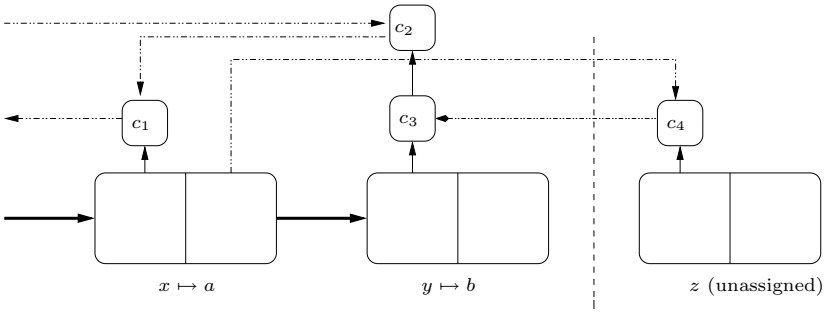


Fig. 2. Constraint Stack Indexing. The figure below presents constraint stack indexing. A variable stack holds the partial assignments in `UC-Search` and is represented on the horizontal axis with bold links. Each variable is placed in an assignment stack and maintains privately two stacks of clauses. In the figure, clauses are labelled c_i with i an index of the time at which the procedure discovered that the clause became unate. Each unate clause belongs to two stacks. First, a stack representing constraints of the variable for which the clause is unate. These stacks are placed vertically associated with each variable in the figure. Second, clauses are placed in a stack associated with the (possibly empty) variable assignment which caused them to become unate. These stacks are linked by dashed lines in the figure.

again, resulting in a new learned clause w' which replaces w and backtracking starts over again with w' and the new state of the constraint index.

Consistency checking also assumes that there is some mechanism in place to evaluate atoms. In our experiments, the solver spent most of its time evaluating atoms. To alleviate this problem, we make use of a database of atoms, so that one atom occuring in multiple clauses will only be evaluated once. We only evaluate atoms occuring in clauses which are unate. This is accomplished with a simple adaptation of two-literal-watching to the case of two-variable watching. For each atom, we cache an evaluation summary, which is simply true or false for a fully evaluated atom and a representation of the atom under constant evaluation for unate atoms. Each evaluation summary for an atom a holds the maximally assigned variable in the variable stack which occurs in a together with the conflict number at the time that variable was assigned. Additionally, every assignment of a variable is associated with the number of conflicts that have occured up to the point of assignment. With this information at hand, a straightforward constant time check allows us to re-evaluate atoms only when they appear in some unate clause and some variables they contain are unassigned and then re-assigned.

5 Application to Linear Real Arithmetic

UCS may be applied to linear real arithmetic. In particular, we can decide a set of clauses of linear constraints over the reals, such as

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

5.1 Resolution

The biggest challenge in instantiating UCS for linear real arithmetic is implementing `resolve()`. Recall that `resolve`(ϕ, α, x) starts with a set of inconsistent clauses $\phi[\alpha]$ about x. A clause $w \in \phi[\alpha]$ represents an interval of forbidden values for x. For example, $(x < 0) \vee (x \leq 1) \vee (x > 4)$ excludes $(1 \ldots 4]$. A set of unate clauses is inconsistent if the union of their respective excluded intervals covers the real line. Following [MKS09], given an assignment α and two x-unate clauses $l[\alpha], u[\alpha]$ under which exclude intervals $I_{\alpha,l}, I_{\alpha,u}$ and such that $I_{\alpha,l} \cup I_{\alpha,u}$ is an interval which is strictly larger, we can derive a new clause $l \oplus_x^\alpha u$ using the shadow rule: assume without loss that the intersection of the weakest lower bound on x in $l[\alpha]$ and the weakest upper bound on x in $u[\alpha]$ is empty. Let $l \equiv \bigvee lb_x(l) \vee \bigvee A$ and $u \equiv \bigvee ub_x(u) \vee \bigvee B$ where $lb_x(l)$ is the set of constraints which lower bound x in l and $up_x(u)$ is the set of constraints which upper bound x in u. If $p \in low_x(l)$ and $q \in up_x(u)$, then denote by $p \oplus_x q$ the result of eliminating[3] x from $\exists x \,.\, p \wedge q$. Then

$$l \oplus_x^\alpha u \doteq \bigvee A \cup B \vee \bigvee \{p \oplus_x q \mid p \in low_x(l), q \in up_x(u)\}$$

To implement `resolve`, we need only repeatedly apply $l \oplus_x^\alpha u$ for an appropriate choice of l and u until x is eliminated.

We state without proof that such an implementation of `resolve` is correct and has finite width for clauses over any set of linearly independent constraints. For details, the reader is referred to [Cot09,MKS09]. From this it follows that `UC-Search` terminates and is a complete algorithm for this class of problems.

5.2 Consistency Checking

Variable consistency checking, *i.e.* an implementation of `isUC()` plays an important role in the `UC-Search` algorithm because it occurs very frequently. In `UC-Search`, the method `isUC()` is called initially and in response to variable

[3] To eliminate x, we can rewrite $p \wedge q$ in the form $f \prec_1 x \prec_2 g$ with each $\prec_i$ either strict or non-strict comparison and conclude $f < g$ if either $\prec_1$ is strict or $\prec_2$ is strict and $f \leq g$ otherwise.

assignments as well as in response to clause learning. Each call to the method checks the unate consistency of all free variables. Of course, one may simply consider the constraints placed on each variable x independently. Over the course of a `UC-Search` run, the constraints over a variable x are asserted incrementally and may be subsequently un-asserted if the procedure backtracks or forgets a clause.

Thus consistency checking begs a per-variable incremental and backtrack friendly implementation. Section 4 describes data structures centered around identifying when clauses become unate, free of true predicates, and non-trivial on the fly. Accordingly, we will assume that consistency checks occur upon the assertion of one non-trivial x-unate clause at a time, for every variable x. More particularly, we consider a sequence of clause assertions $c_{x,1}c_{x,2}\ldots c_{x,k}$ where each clause $c_{x,i}$ is a non-trivial x-unate clause. A convenient means to maintain consistency for x incrementally is to under-approximate the feasible set with lower and upper bounds l_x, u_x such that

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

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

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

If $l'_x \wedge u'_x$ is satisfiable, the under-approximation may defer a real consistency check until a new x-unate constraint is asserted. Otherwise, a real consistency check needs to take place with respect to the current set of clauses to find whether there is a set of inconsistent clauses. Otherwise, a new under-approximation needs to be computed.

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

6 Experimentation with Difference Logic

We implemented a solver for linear real arithmetic based on the ideas presented in Section 5. The solver is restricted to CNF formulas of the form above, and does not use arbitrary precision arithmetic. However, we did implement a complete solver that backtracks farther than the formal UCS algorithm and we did extend the solver to perform unit propagation on propositional literals coded in real linear arithmetic. Nonetheless, As a result of numerical constraints and CNF requirements, we only applied the solver to some difference logic problem sets in SMT-LIB [BRST08]. We found that the formal `UC-Search` algorithm was quite slow and that an incomplete solver which does clause recording was much faster. All the experimental data presented below represents an incomplete solver. All experiments were run on a Sun Java VM version 1.6.0_11 on a Debian Linux machine with dual Xeon 3.20 GHz processors and 4GB RAM.

6.1 Job Shop

Job shop scheduling problems are formulated as a set of jobs, each of which is a sequence of tasks. Each task makes use of a pre-specified resource for some fixed duration. With a fixed set of resources, a query is generated as to whether all tasks can complete within some given time, referred to as the *makespan*. These problems fall within the difference logic fragment of linear arithmetic and traditional SMT solvers usually make use of dedicated algorithms such as that presented in [CM06].

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

The variable selection entries may be one of

1. *fix*. The solver uses a fixed variable ordering based on variable identities[4].
2. *vsid*. All variables are selected according to VSID heuristics, and all variables are incremented on every clause resolution step.
3. *evsid*. The solver does extended assertion level backtracking, in which it backtracks to the minimal assignment which makes a learned clause unate, and selects variables based on VSIDs.

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

The value selection strategies either used the last assigned value, a recent assigned value derived from consistency checking, or the current value. In addition, we instrumented the solver to toggle at various times the direction in which a search for a satisfying assignment would occur on consistency checks. The possible values are to toggle on every assignment, never, or only immediately after asserting a learned clause.

A table of the results is available in Appendix B.1. The most important configuration choice appears to be whether or not a fixed variable order is used. Apart from this, we observe that value selection plays a very important role and that the "last" configuration outperforms the "recent" configuration which in turn generally outperforms the "current" configuration. The bias flipping mechanism also appears to have a significant impact on performance. Generally, bias toggling on assignments seems to work best. But in the best overall configuration, bias toggling on assertion appears to work best. There is a large span of improvement over the space of configurations: the best configuration solves more than 4 times the problems of the worst.

Best Configuration We ran the best configuration with extended asserting backtrack levels, VSIDs, and assertion bias toggling with a timeout of 300 seconds on the same set of problems and with the same machine. A side-by-side of the results against Z3 are summarized in Appendix B.2. Z3 is vastly faster than UCS on scheduling problems.

6.2 Diamonds

In [MKS09], it was observed that the resolution proof rule can produce exponentially shorter proofs for diamond shaped problems. We confirm this observation for the diamond problems in SMT-LIB with a comparison of our best configuration to Z3. The results are available in Appendix B.3

6.3 Parity Games

Another class of difference logic problems from SMT-LIB contains a mix of propositional and integer variables. Our best configuration for the job shop problems was used. Despite an overall bias in favor of Z3, the largely off-diagonol results indicate that the relative strengths of UCS and Z3 are somewhat orthogonal on this set of benchmarks.

7 Conclusion

We have presented an abstract algorithm, `UC-Search`, for deciding a wide range of quantifier-free CNF formulas. The goal of this work is to find a decision procedure formalism which supports more principled and dynamic heuristics for SMT solving. Our procedure supports VSID style heuristics over arbitrary variables, as well as principled and dynamic heuristics for forgetting clauses based on the notion of progress (Theorem 2). While this procedure may be applied to a wide range of theories, we observed that unbounded proof procedures introduce severe

restrictions on the algorithm. Despite this fact, experiments indicate that even an incomplete version of the procedure is much faster than a leading SMT solver based on traditional techniques on a significant portion (albeit a minority) of the benchmarks we performed. While our results do not achieve improvements as a *general-purpose* solver for linear arithmetic, by applying the algorithm in this way we have discovered some fundamental costs associated with introducing dynamic variable orderings in systems with unbounded proof systems. To remedy this situation, a solver based on arbitrary regular resolution, rather than directed or tree-like resolution could be explored. Alternatively, `UC-Search` could be applied to other theories or embedded within a traditional framework, providing a basis for deriving and forgetting theory predicates.

References

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

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

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability Modulo Theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

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

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

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

[CM06] Scott Cotton and Oded Maler. Fast and flexible difference constraint propagation for dpll(t). In *SAT06*, 2006.

[Cot09] Scott Cotton. *On Some Problems in Satisfiability Solving.* PhD thesis, University of Grenoble Joseph Fourier, to be published, 2009.

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

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

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

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

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

A Proofs

A.1 UCS Progress

Theorem 5. `UC-Search` *Progress, Theorem 2*
Let $U(\phi, \alpha)$ *denote the set of all unate-feasible* 1*-extensions of* α

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

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

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

1. Case 1. $\alpha \subseteq \alpha_k$, for all $j \leq k \leq i$. In this case, call i occurs at line 9, and $\phi_i \equiv \phi_j \wedge w$ for some clause w returned by resolve. The procedure `resolve()` returns a clause which excludes an assignment $x \mapsto a$ such that x does not have an assignment in α. Moreover $x \mapsto a$ was a feasible assignment under $\phi_j[\alpha]$. Hence $U(\phi_i, \alpha) \subset U(\phi_j, \alpha)$.
2. Case 2. There is a k with $j \leq k \leq i$ such that $\alpha_k \subset \alpha$. This case is impossible because when the procedure backtracks over α, it either terminates or a clause w' is conjoined which excludes some subassignment α' of α. Moreover, once this clause is conjoined, all subsequent calls either fall in a call scope in which ϕ contains the clause w' or the procedure backtracks out of this scope with a new clause v' which excludes some subassignment of α'.

□

A.2 Restricted Resolution

Theorem 6 (Restricted Resolution). *If* `UC-Search` *is exhaustively asserting then the proof graph traced by* `UC-Search` *is tree-like and regular.*

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

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

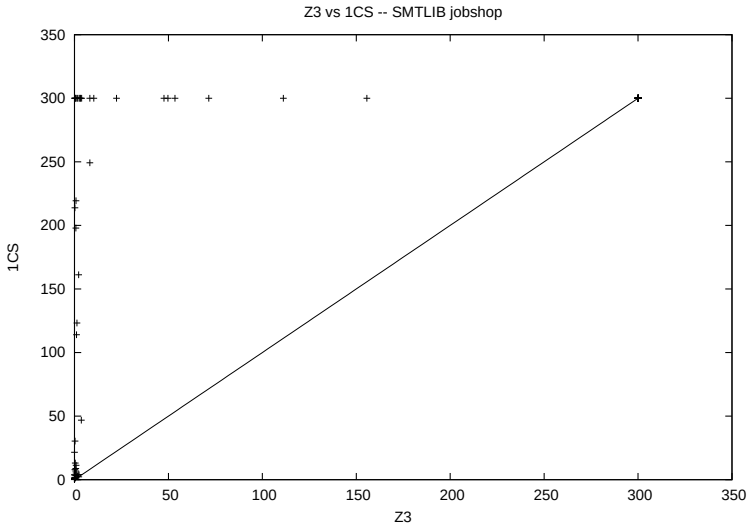
In both cases, the learned clauses are forgotten once they fall out of scope, so they give rise to at most one consequence. Regularity then follows because the pivot variables always follow the reverse order in which they are assigned. □

B Experiments

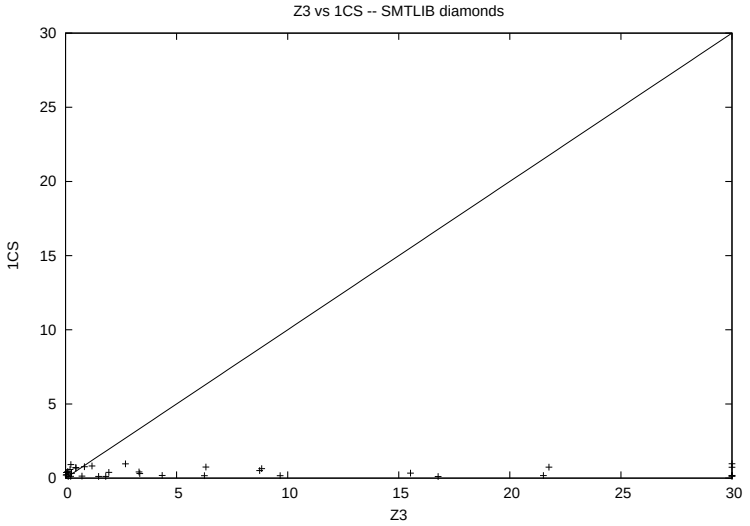
B.1 Job Shop Wide Net

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

B.2 Job Shop Best Configuration compared with Z3



B.3 Diamond Problems



B.4 Parity Games