Efficient Constraint-Satisfaction in Domains with Time

Perrin G. Bignoli, Nicholas L. Cassimatis, Arthi Murugesan

Department of Cognitive Science
Rensselaer Polytechnic Institute
Troy, NY 12810

{bignop, cassin, muruga}@rpi.edu

Abstract

Satisfiability (SAT) testing methods have been used effectively in many inference, planning and constraint satisfaction tasks and thus have been considered a contribution towards artificial general intelligence. However, since SAT constraints are defined over atomic propositions, domains with state variables that change over time can lead to extremely large search spaces. This poses both memory- and time-efficiency problems for existing SAT algorithms. In this paper, we propose to address these problems by introducing a language that encodes the temporal intervals over which relations occur and an integrated system that satisfies constraints formulated in this language. Temporal intervals are presented as a compressed method of encoding time that results in significantly smaller search spaces. However, intervals cannot be used efficiently without significant modifications to traditional SAT algorithms. Using the Polyscheme cognitive architecture, we created a system that integrates a DPLL-like SAT-solving algorithm with a rule matcher in order to support intervals by allowing new constraints and objects to be lazily instantiated throughout inference. Our system also includes constraint graphs to compactly store information about temporal and identity relationships between objects. In addition, a memory retrieval subsystem was utilized to guide inference towards minimal models in common sense reasoning problems involving time and change. We performed two sets of evaluations to isolate the contributions of the system's individual components. These tests demonstrate that both the ability to add new objects during inference and the use of smart memory retrieval result in a significant increase in performance over pure satisfiability algorithms alone and offer solutions to some problems on a larger scale than what was possible before.

Introduction

Many AI applications have been successfully framed as SAT problems: planning (Kautz and Selman 1999), computer-aided design (Marques-Silva and Sakallah 2000), diagnosis (Smith and Veneris 2005), and scheduling (Feldman and Golumbic 1990). Although SAT-solvers have successfully handled problems with millions of clauses, tasks that require an explicit representation of time can exceed their capacities.

Adding a temporal dimension to a problem space has the potential to greatly expand search space sizes because SAT algorithms propositionalize relational constraints. The most direct way to incorporate time is to have a copy of each state variable for every time point over which the system reasons. This increases the number of propositions by a factor equal to the number of time points involved. Since SAT algorithms generally become slower as the size of a problem increases, adding a temporal dimension to even relatively simple problems can make them intractable.

Although problems with time require more space to encode, the true expense of introducing time stems from the additional cost required to find a SAT solution. Consider a task that requires the comparison of all the possible ways that a car can visit three locations in order. If the problem has no reference to time points, there is only one solution: the car just moves from *a* to *b* to *c*. On the other hand, there are clearly more possibilities, since the car could potentially move or not move at every time. Compared to other SAT-solvers, LazySAT (Singla and Domingos 2006), which lazily instantiates constraints, is less affected by the increased memory demands of larger search spaces. Unfortunately, lazy instantiation will not increase the tractability of larger problems with respect to runtime.

There is, however, a more efficient way of representing time. Since it is unlikely that the truth value of a proposition will change at every time point, temporal intervals can be used to denote segments of contiguous time points over which its value is constant. This practice alleviates the need to duplicate all propositions at every time point for most problem instances, thus significantly reducing the search space size. Intervals also mitigate the arbitrary granularity of time because they are continuous and scale independent.

However, existing SAT solvers cannot process intervals efficiently because they do not allow new objects to be introduced during the course of inference. It is clearly impossible to know exactly which temporal intervals will be required. Therefore, every possible interval must be defined in advance. Since $^{(n-1)(n-2)}\!/_2$ unique intervals can be defined over n times, there would be little advantage to use them with current searching methods.

To capture the benefits of SAT while supporting the use of intervals, we created an integrated system that combines a DPLL-like search with several specialized forms of inference: a rule matcher, constraint graphs, and memory retrieval. Rule matching allows our system to both lazily instantiate constraints and introduce new objects during inference. Constraint graphs compactly store temporal and identity relationships between objects. Memory retrieval supports common sense reasoning about time. Although SAT has previously been applied to planning (Shanahan and Witkowski 2004) and reasoning (Mueller 2004) with the event calculus, we present a novel approach.

Language

We have specified a language that can express relationships between objects at different points in time. This language incorporates temporal intervals and a mechanism for introducing new objects during inference. For example, in path planning tasks, it is often useful to have a condition such as: If at location *p1* and must be at location *p2*, which is not adjacent to *p1*, then move to some location *px* that is adjacent to *p1* at the next possible time. We write this constraint as:

$$\begin{aligned}&Location(?x,?p1,?t1) \wedge Location(?x,?p2,?t2) \wedge \neg Same(?p1,?p2,E) \Rightarrow (\infty)\\&Adjacent(?p1,?px,E) \wedge Meets(?t1,?tx,E)\end{aligned}$$

Note that we prefix arguments with a '?' to denote variables. Variables permit this one constraint to apply to all locations in the system. Additionally, we can assign weights to constraints as a measure of their importance. Because this constraint holds for any path, it is given an infinite weight to indicate that it must be satisfied.

Formally, constraints have the form $A_1 \wedge \ldots \wedge A_m \Rightarrow (w)\, C_1 \wedge \ldots \wedge C_n$, where w is a positive rational number, m ≥ 0 and n ≥ 1. A_i and C_j are first order literals. Literals have the form $P(arg_1, \ldots, arg_n)$, where *P* is a predicate, arg_i is a term, and arg_n must be a time point or interval. Terms that are prefixed with a '?' are variables; others are constant "objects." A grounded predicate is one in which no term is a variable. Predicates specify relations over the first n-1 terms, which hold at time arg_n. There is a special type of relation called an attribute. Attributes are predicates, *P*, with three terms, *o*, *v*, and *t*, such that only one relation of the form *P(o, v, t)* holds for each *o* at every *t*. The negation of a literal is expressed with the ¬ operator. Every literal is mapped to a truth value.

If all of the literals in a constraint are grounded, then the constraint itself is grounded. Only grounded constraints can be satisfied or broken, according to the truth value of its component literals. A constraint is broken iff every antecedent literal is assigned to true and every consequent literal is assigned to false. The cost of breaking a constraint is given by w, which is infinite if the constraint is hard.

Some predicates and objects are included in the language. For instance, Meets, Before, and Includes are temporal relations that are similar to the predicates in Allen's interval calculus (Allen 1981). We reserve a set of objects of the form *{t1, …, tn}*, where n is the number of times in the system. This set is known as the native times. Another time, E, denotes eternity and is used in literals whose assignments do not change over time.

A model of a theory in this language consists of a complete assignment, which is a mapping of every literal to a truth value. Valid models are those such that their assignment permits all hard constraints to be satisfied. All models have an associated cost equal to the cost of its broken constraints. Each theory has many valid models, but it is often useful to find one of the models with the minimum cost. For instance, this process can perform change minimization, a form of commonsense reasoning motivated by the frame problem (Shanahan 1997).

System Architecture

We created an integrated system using the Polyscheme cognitive architecture (Cassimatis 2002) in order to efficiently solve problems with time. This approach allowed us to glean the benefits of SAT while capitalizing on the properties of specialized forms of inference. It is easiest to describe how our system works by framing it as a DPLL-like search. DPLL (Davis, Logemann et al. 1962) performs a branch-and-bound depth first search that is guaranteed to be complete for finite search spaces. The algorithm searches for the best assignment by making assumptions about the literals that appear in its constraint set. An assumption consists of selecting an unassigned literal, setting it to true or false, and then performing inference based on this assignment. When necessary, the search backtracks to previous decision points to explore the ramifications of making the opposite assumption.

The DPLL-FIRST procedure in Algorithm 1 takes a set of constraints, *c*, as input and outputs an assignment of literals to truth values that minimizes the cost of broken constraints. In the input constraint set, there must be at least one fully grounded constraint with a single literal. Such constraints are called facts. Within the DPLL-FIRST procedure, several data structures are declared and passed to DPLL-RECUR, which is illustrated in Algorithm 2. First, there is a structure, *assign*, which stores the current assignment of literals to truth values. The facts specified in the input are stored in *assign* with an assignment that corresponds to the valence of the fact's literal. Second, there is a queue, *q*, which stores the literals that have been deemed relevant by the system in order to perform inference after the previous assumption. At the beginning, *q* contains the facts in the input. Third, *b* stores the best total assignment that has been found so far. Fourth, the cost of *b* is stored in *o*. Initially, *b* is empty and *o* is infinite.

```
Algorithm 1. DPLL-FIRST(c):
    return DPLL-RECUR(c, assign, q, o, b)
```

Each time DPLL_RECUR is called, it performs an elaboration step that infers new assignments based on the current assumption. Initially, when there is no assumption, the elaboration step attempts to infer new information from the constraints specified in the input. After elaboration, the current assignment is examined to determine if one of the three termination conditions is met. The first condition is if the assignment is contradictory because the same literal has been assigned to both true and false. The second condition is if the cost of the current assignment exceeds the lowest cost of a complete assignment that has been discovered in previous iterations. Since new assignments can never reduce the total cost, it is unnecessary to continue searching. The third condition is if the current assignment is complete. In all of these cases, the search backtracks to a previous assumption and investigates any remaining unexplored possibilities. Afterwards, the search selects an unassigned literal and creates two new branches in the search tree: one where the literal is assumed to be true and one where it is assumed to be false. DPLL-RECUR is then invoked on those subtrees and the assignment from the branch with the lower cost is returned.

Algorithm 2. DPLL-RECUR(*c*, *assign*, *q*, *o*, *b*)

```
call ELABORATION($c$, $assign$, $q$)
if Contradictory($assign$) then
   return Fail
else if Cost($assign$) > $o$
   return Fail
else if Complete($assign$)
   return $assign$
end if
$u \leftarrow$ next element of $q$
$newassign \leftarrow assign$ with $u$ assigned to true
$b1 \leftarrow$ call DPLL-RECUR($c$, $newassign$, $o$, $b$)
$newassign \leftarrow assign$ with $u$ assigned to false
$b2 \leftarrow$ call DPLL-RECUR($c$, $newassign$, $o$, $b$)
if Cost($b1$) < Cost($b2$) then
   return b1
else
   return b2
end if
```

The elaboration step in basic DPLL is called unit-propagation. Unit-propagation examines the current assignment to determine if there are any constraints that have exactly one literal unassigned. If such constraints exist and exactly one assignment (i.e., true or false) for that literal satisfies the constraint, then DPLL makes that assignment immediately instead of through a later assumption. Our system augments this basic technique by introducing several more specialized forms of inference.

To understand the importance of elaboration, consider that all of the best available complete SAT-solvers are based on some version of DPLL (Moskewicz, Madigan et al. 2001; Een and Sorensson 2005). DPLL is so effective because its elaboration step eliminates the need to explore large numbers of unnecessary assumptions. It is more efficient to infer assignments directly rather than to make assumptions, because each assumption is equivalent to creating a new branch in DPLL's abstract search tree. Elaboration also allows early detection of contradictions in the current assignment.

Despite its elaboration step, DPLL is unable to handle the large search spaces that occur when time is explicitly represented. The goal of our approach is to improve elaboration by using a combination of specialized inference routines. Previous work (Cassimatis, Bugjaska et al. 2007) has outlined the implementation of SAT solvers in Polyscheme. Following that approach, we implemented DPLL using Polyscheme's focus of attention. One call to DPLL-RECUR is implemented by one focus of attention in Polyscheme. Logical worlds are used to manage DPLL assumptions. For each assumption, an alternative world is created in which the literal in question is either true or false. Once Polyscheme focuses on an assumption literal, it is elaborated by polling the opinions of several specialists. These specialists implement the specialized inference routines upon which our system relies. One of these specialists, the rule matcher, lazily instantiates grounded constraints that involve the current assumption. The assignment of a literal is given by Polyscheme's final consensus on the corresponding proposition. This elaboration constitutes the main difference between our system and standard DPLL.

Our elaboration step, which is illustrated in Algorithm 3, loops over the literals that have been added to the queue because their assignments were modified by previous inference. Two procedures are performed on each of these literals. First, a rule matcher is used to lazily instantiate grounded constraints from relevant variable constraints provided in the input. Relevant constraints are those that contain a term that corresponds to the current literal in focus. These constraints are "fired" to propagate truth value from the antecedent terms to the consequent terms. Newly grounded literals, which may contain new objects, are introduced during this process. Any such literals are added to the assignment store and the queue.

The second procedure involves formulating an assignment for the current proposition based on suggestions from the various components of the system. For instance, the temporal constraint graph is queried here in the case that the proposition being examined describes a temporal relationship. Likewise, the identity constraint graph would be queried if the examined proposition was an identity relationship. In the extended system, this is the step at which the memory retrieval mechanism would be utilized. These opinions are combined with the old assignment of the proposition to produce a new assignment. If the new assignment differs from the old one, the literal is placed back on the queue.

Algorithm 3. ELABORATION(*c*, *assign*, *q*):

```
while $q$ is not empty do
   $l \leftarrow$ the next element in $q$
   $ris \leftarrow$ call Match($l$, $c$, $q$)
   $delta \leftarrow \emptyset$
   for each $ri$ in $ris$ do
      $delta \leftarrow$ delta $\bigcup$ call Propagate($ri$, $assign$)
   end for
```

```
    rs ← the rule system's opinion on l
    tc ← the temporal constraint graph's opinion on l
    ic ← the identity constraint graph's opinion on l
    mr ← the memory retrieval system's opinion on l
    c ← call Combine(rs, tc, ic, mr)
    if c ≠ l's assignment in assign then
      delta ← delta ⋃ l
    end if
    q ← q ⋃ delta
  end while
  return
```

Rule Matching Component

Lazy instantiation is efficient because it avoids the creation of constraints that do not need to be considered to produce an acceptable assignment. We accomplish lazy instantiation by treating constraints with open variables as templates that can be passed to a rule matcher. The rule matching component (RMC) attempts to bind the arguments of the last dequeued literal to variables in the constraint rules. A binding is valid if it allows all variables in the antecedent terms of a constraint to be bound to objects in the arguments of literals that are stored in the system's memory. All valid bindings are evaluated as constraints by propagating truth value from the antecedents to the consequents. A constraint is considered broken only if the grounded literals in its antecedent terms have been assigned to true and the propositions in its consequent terms have been assigned to false.

A simple example will illustrate the binding process. Let the literal currently being assigned be $Location(car1, road, t1)$. If $Location(?x, ?y, ?t1) \Rightarrow (10)\ Location(?x, ?y, ?t2) \wedge Meets(?t1, ?t2, E)$ appears as a constraint, then the following fully grounded instance is created:

$$Location(car1, road, t1) \Rightarrow (10)\ Location(car1, road, tnew) \wedge Meets(t1, tnew, E)$$

In this case, *?x* binds to *car1*, *?y* binds to *road*, *?t1* binds to *t1*, and *?t2* binds to a new object, *tnew*. If *Location(car1, road, t1)* is assigned to true and *Location(car1, road, t_new)* is later assigned to false, then any models that contain that assignment will accrue a cost of 10.

Temporal Constraint Graph Component

As the number of objects in a problem instance increase, so do the number of literals and constraints. For instance, when time objects are introduced, it is often necessary to know how those times are ordered. If there are n times in the system, approximately $n^2/2$ literals are required to represent all of the values of a binary relation over those times. Usually, only a small portion of these literals provide useful information for solving the problem. Instead of eagerly encoding all temporal relations, we created a component that could be queried on demand to determine if a given relation holds according to the current knowledge of the system. This component represents relations in graphical form.

Using a graph enables the system to derive new entailed relations without storing them explicitly as propositions. All of the fundamental temporal relationships described by Allen can be represented in the following way. Whenever a literal that involves such a relationship is encountered, the temporal constraint graph (TCGC) decomposes the time object arguments into three parts: the start point, the midpoints, and the end point. These parts form the nodes of the graph. Edges are created between two nodes in the graph if their temporal relationship is known.

Every interval relationship can be derived from only two types of relationships on the parts of times: Before and Equals. A time object's start point is defined to be before its midpoints and its midpoints are before its end point. By creating edges between the parts of different time objects, it is possible to record relationships between the objects themselves. For instance, to encode *Meets(t1, t2, E)* in the graph, one would use the relationship: *Equals(end-t1, start-t2, E)*. To illustrate why the graph is an efficient way to store this information, consider the following example. If it is known that *Meets(t1, t2, E)* and *Meets(t2, t3, E)*, then the graph can be traversed to find *Before(t1, t3, E)*, among other relationships. Thus, these propositions are stored implicitly and do not need to be assigned unless they are present in a grounded constraint.

Identity Constraint Graph Component

The identity constraint graph component (ICGC) is similar to the TCGC, but it handles propositions about the identity of objects. This graph consists of nodes that represent objects and edges that represent either equality or inequality. By traversing the graph, it is possible to capture the transitivity of the identity relation. Although a rule could be used to generate the transitivity property, doing so has the potential to drastically increase the number of propositions over which DPLL must search.

Another important use for the ICGC is that it can detect inconsistencies in truth assignments to identity propositions. Consider that the following set of identity propositions is known: *Same(a, b)*, *Same(b, c)*. Then, the proposition, *¬Same(a, c)*, is examined and assumed to be true. Clearly this is inconsistent, but without including a rule that defines the properties of identity, the system will continue to perform inference based on these facts until an explicit contradiction is encountered. Traversing the edges connecting *a*, *b*, and *c* in the graph will indicate that this scenario is contradictory. This information can be reported to the system as a whole during elaboration.

Memory Retrieval Component

Systems that explicitly represent time often also have to reason about change. However, in situations with imperfect knowledge, reasoning about change can be difficult. Consider the following information about the color of a car, assuming that Location is an attribute: *Color(car, red, t1)*, *Color(car, blue, t6)*, *Color(car, green, t11)*. In between *t1* and *t6* and *t6* and *t11*, it is consistent for the car to be any color. However, in many scenarios, the car would remain *red* until it was painted *blue* and then *blue* until it was painted *green*. Although the world is dynamic, it also exhibits inertia. The principle of change minimization states that changes to particular objects are relatively infrequent and when changes do occur, they will be salient. It is worthwhile for a reasoning system to bet on such recurring patterns, because doing so significantly reduces the complexity of many problems.

We can frame the change minimization problem as a weighted SAT problem as follows. Given a set of attribute literals that involve the same first term, but whose second term changes over time, determine the minimum number of changes required to explain the data. To this end, constraints can be defined so that the least costly models will be those that exhibit the smallest possible number of attribute value changes. The memory retrieval component (MRC) is designed to accommodate this procedure by regulating which attribute values appear in grounded constraints. Only attribute values with some prior evidence in memory are considered. As an example, the car could have been *purple* at *t2* and *brown* at *t3*, but models that contain such literals are automatically excluded.

To control which values are considered, the MRC makes a copy of each literal that encodes an attribute with a native time index. This copy is modified to contain a time index that corresponds to an open-ended interval. Only when new attribute values are observed will that interval's endpoints be constrained. For instance, when the system elaborates *Color(car, red, t1)*, a new literal *Color(car, red, t-car-red)* will be introduced. Because it uses a content-addressable retrieval system, the MRC will henceforth report that the color of the car is likely to be red at every time point until new information is discovered. If the literal *Color(car, blue, t6)* is observed, then the literal *Color(car, blue, t-car-blue)* will be introduced. Also, constraints will be added to limit the right side of the *t-car-red* interval at *t6* and the left side of the *t-car-blue* interval at *t1*.

Even with this optimization, change minimization is expensive if performed naively, since there are $^{(n-1)(n)}/_{2}$ possible transitions between n attribute values if nothing is known about when the values hold relative to each other. For instance, if a car is seen to be *red*, then *blue*, then *green*, the naïve formulation will consider such possibilities as a change from *green* to *red* even though this is clearly impossible. The change detection mechanism in the MRC utilizes the TCGC in conjunction with content-accessible memory to significantly reduce the number of impossible changes that are considered by the system. When an attribute literal containing an interval is elaborated, the only values that are adjacent in time to the current interval will be considered. Since the intervals do not have completely fixed endpoints, these neighboring times can be detected by looking at what times the current interval could possibly include. For instance, once it has been established that there is a change between *t1* and *t6*, the interval created around *t1* can no longer include *t11*, so the location *t1* will not appear as a possible previous state of the location at *t11*.

Results

Our system was designed to improve the efficiency of applying SAT-solving to problems that involve an explicit representation of time. In order to accomplish this goal, we used Polyscheme to implement a DPLL-like algorithm with specialized forms of inference that permitted the creation of new objects. The ability to introduce new objects allowed us to use intervals to reduce the search space of these problems.

In order to test the level of improvement gained by the ability to add new objects independently of other techniques, we ran an evaluation with the memory retrieval component deactivated. The task we selected was optimal path planning through space. We represented this space as a graph. Although the particular problem we used did not model a changing environment, an explicit represent of time would be required if actions had side effects or if objects in the environment had changing states. For instance, a particular environment might contain walls that crumble over time. We asked the system to find the shortest valid path between two particular locations on the graph. Valid paths consisted of movements between adjacent locations that did not contain active obstacles. These types of problems are important to the object tracking and motion planning domains.

We initially compared our system against LazySAT because it is similar to our approach in that it also supports the lazy instantiation of constraints. Through experimentation, we determined what values of LazySAT's parameters enabled high performance on this task. LazySAT, however, is based on a randomized local search, which is inefficient for many structured domains. Therefore, we ran the same set of tests on MiniMaxSAT (Heras, Larrrosa et al. 2007), which is one of the best DPLL-based systematic SAT-solvers. Because Markov logic is not complete and cannot report unsatisfiability, we were forced to select a configuration of space in which a valid path existed.

The evaluation problem involved a 9-location graph with one obstacle, which had to be circumvented. We were limited to 9 locations because the performance of the systems degraded on larger problems. To determine how well each system handled time, we created ten versions of the problem from 5 to 50 time points in increments of 5. For each condition and each system, we ran 10 trials and recorded the average runtime. These results are displayed in Figure 1. While our system was consistently better than LazySAT, MiniMaxSAT outperformed both until it had to contend with more than 30 time points. Our system required approximately constant time to solve the problem due to the fact that intervals make this task equivalent to the case of planning without explicit time points.

A second evaluation was conducted to show that the memory retrieval subsystem allowed change minimization problems to be solved efficiently. We ran these tests on the system with and without the MRC activated. These problems consisted of a number of attribute value observations that involved the color of an object. For instance, suppose the following facts were given as input: *Color(car, red, t1)*, *Color(car, blue, t6)*, and *Color(car, green, t11)*. The system would then be tasked with finding the least costly model that explained this data, namely that the color changed from *red* to *blue* and then from *blue* to *green*.

The results from this evaluation are depicted in Figure 2. Not only does enabling the MRC permit the change minimization problems to be solved in less time than with the basic system, but it also increases the upper limit on problem size. Without the MRC, our system ran for over 50,000 seconds attempting to solve the 5 attribute value problem. These results demonstrate that the elimination of irrelevant constraints is a powerful technique for improving the performance of SAT-solvers.

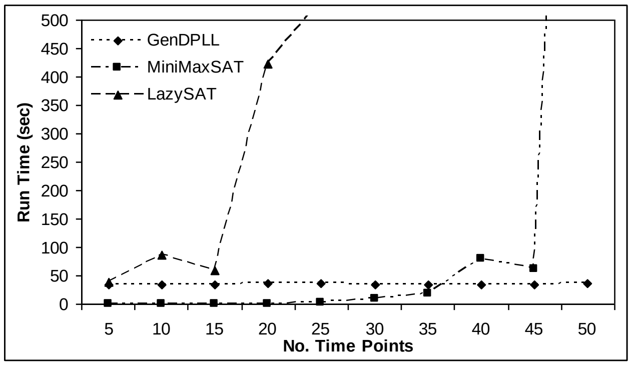

Figure 1: Path planning problem results

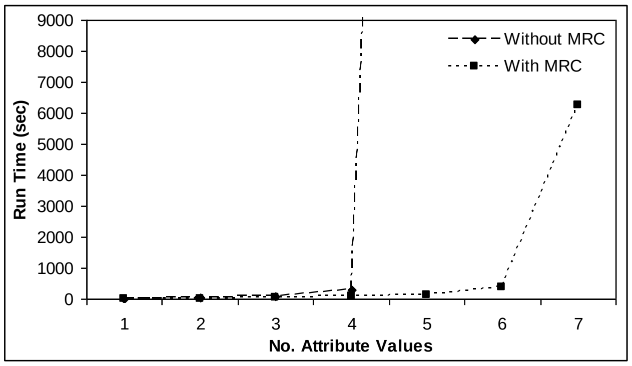

Figure 2: Change minimization results

Conclusion

Although SAT is in some cases an efficient approach to domain-general problem solving, it does not scale well to the large search spaces that result from tasks that require an explicit representation of time. Temporal intervals help to reduce the size of such problems, but can be used effectively only with SAT-solvers that permit the introduction of novel objects throughout inference. Hence, we created a system that combines specialized inference techniques with a DPLL-like algorithm. This system was shown to outperform MiniMaxSAT and LazySAT in a series of evaluations involving a simple path planning domain.

When time is represented explicitly, it is also beneficial to incorporate common sense reasoning that exploits common patterns in real-world problem instances. Towards this end, a memory retrieval subsystem was developed that prevented the exploration of fruitless paths in the DPLL search tree under certain conditions. This technique was demonstrated to increase the efficiency of how our system solves the change minimization problem.

References

Allen, J. (1981). Maintaining knowledge about temporal intervals. TR-86, Computer Science Department, University of Rochester, Rochester, NY.

Cassimatis, N. L. (2002). Polyscheme: A Cognitive Architecture for Integrating Multiple Representation and Inference Schemes. Media Laboratory. Cambridge, MA, Massachusetts Institute of Technology.

Cassimatis, N. L., M. Bugjaska, et al. (2007). An Architecture for Adaptive Algorithmic Hybrids. AAAI-07, Vancouver, BC.

Davis, M., G. Logemann, et al. (1962). "A Machine Program for Theorem Proving." Communications of the ACM **5**(7): 394–397.

Een, N. and N. Sorensson (2005). MiniSat-A SAT solver with conflict-clause minimization. SAT 2005 Competition.

Feldman, R. and M. C. Golumbic (1990). "Optimization algorithms for student scheduling via constraint satisfiability." Computer Journal **33**: 356-364.

Heras, F., J. Larrrosa, et al. (2007). "MiniMaxSAT: a new weight Max-SAT solver." International Conference on Theory and Application of Satisfiability Testing: 41-55.

Kautz, H. and B. Selman (1999). Unifying SAT-based and Graph-based Planning. IJCAI-99.

Marques-Silva, J. P. and K. A. Sakallah (2000). "Boolean satisfiability in electronic design automation." Proc., IEEE/ACM Design Automation Conference (DAC '00).

Moskewicz, M., C. Madigan, et al. (2001). Chaff: Engineering an Efficient SAT Solver 39th Design Automation Conference, Las Vegas.

Mueller, E. T. (2004). "Event calculus reasoning through satisfiability." Journal of Logic and Computation **14**(5): 703-730.

Russell, S. and P. Norvig (1995). Artificial Intelligence: A Modern Approach, Prentice Hall.

Shanahan, M. (1997). Solving the Frame Problem, a mathematical investigation of the common sense law of inertia, M.I.T. Press.

Shanahan, M. and M. Witkowski (2004). "Event calculus planning through satisfiability." Journal of Logic and Computation **14**(5): 731-745.

Singla, P. and P. Domingos (2006). Memory-efficient inference in relational domains. Proceedings of the Twenty-First National Conference on Artificial Intelligence, Boston, MA.

Smith, A. and A. Veneris (2005). "Fault diagnosis and logic debugging using boolean satsifiability." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **24**.