Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)

The Complexity of Optimal Planning and a More Efficient Method for Finding Solutions

Katrina Ray and **Matthew L. Ginsberg**

University of Oregon CIRL and On Time Systems
1850 Millrace Dr. Ste. 1
Eugene, OR 97402
kray@cs.uoregon.edu ginsberg@otsys.com

Abstract

We present a faster method of solving optimal planning problems and show that our solution performs up to an order of magnitude faster than Satplan on a variety of problems from the IPC-5 benchmarks. Satplan makes several calls to a SAT solver, discarding learned information with each call. Our planner uses a single call to a SAT solver, eliminating this problem.

We explain our technique by describing a new theoretical framework which allows us to prove that a single call to a SAT solver is sufficient to solve every problem in Δ_2. (This does not imply that NP is equal to Δ_2; only that SAT solvers' capabilities are greater than previously realized.) We also prove that optimal planning is $F\Theta_2$ Complete when the plan length is bounded by a polynomial; optimal planning is thus harder than SAT even in the presence of such a bound. Despite the relative complexities, the Δ_2 capability of DPLL and the fact that $\Theta_2 \subseteq \Delta_2$ show that a single satisfiability call can solve optimal planning problems in the presence of a polynomial bound on plan length.

Introduction

Satplan (Kautz *et al*. 2006) is one of the most efficient solvers for finding solutions to planning problems. It is based on the Planning as Satisfiability approach introduced by Kautz and Selman. (Kautz; Selman 1992) Satplan took first place the International Planning Competition in 2004 and tied for first in 2006. The basic algorithm is:

- Generate plan graph up to length *k* (initially 1)
- If goals are unreachable increment *k* and start over
- Convert plan graph into SAT formula
- Call a satisfiability solver
    - If UNSAT increment *k* and try again
    - If SAT return solution

Each time a plan graph is generated and converted to cnf, the satisfiability instance represents the question, "can we find a plan of length *k*?" Satplan automatically finds an optimal solution (a plan of minimum length) because it searches all possible plan lengths in numerical order.

There are drawbacks to using Satplan for optimal planning. Probably the biggest concern is that we discard learned information between successive calls to the SAT solver. Learning techniques have significantly improved the performance of modern SAT solvers. Throwing away learned information between individual calls is particularly detrimental when using satisfiability for planning because of the strong similarities between successive SAT instances. Information learned in searching for a plan of length *k* is valuable in searching for a plan of length *k*+1.

Prior work has been done on retaining learned clauses between successive calls. (Nabeshima *et al*. 2006) They use a sequence of calls to a SAT solver, but keep learned information to avoid redundant work. Their solution outperforms Satplan demonstrating that retaining learned clauses is useful.

As an alternative to their approach, we propose an approach that uses a single call to a SAT solver. We first generate the entire plan graph. We then determine a partial order on branch decisions that will ensure optimality. Finally we call the SAT solver which uses our fixed partial order before relying on its own branching heuristic. Using a single SAT call automatically eliminates the problem of discarding learned information, but may yield additional benefits by relying on the optimization techniques built in to the SAT solver.

Unfortunately, this approach is not feasible in practice because generating the entire plan graph requires too much memory. To handle this issue, we place an upper bound on the plan length. For instance, we could solve blocks world problems by moving all blocks to the table and building up the goal state. The number of steps required to do this is a naïve upper bound on the optimal plan length. We modified Satplan so that it takes an extra argument which represents an upper bound on the plan length and uses only one SAT call to find an optimal solution. Our experiments indicate that these changes result in a dramatic runtime improvement.

For problems such as blocks world it is easy to generate an upper bound on the length of an optimal solution. For many other problems it can be more difficult. Furthermore, it is desirable to have a self contained planner that does not rely on additional input. We modified our first version by automating the process of finding an upper bound. In the next section we will describe how to generate the partial branch order to guarantee optimality and how to automate the process of finding an upper bound.

Plan existence is PSPACE Complete in general and NP Complete when there is a polynomial bound on plan length. We prove that when there is a polynomial bound on plan length that optimal planning is $F\Theta_2$ Complete, where Θ_2 is the set of problems solvable in deterministic polynomial time given a logarithmic number of queries to an NP oracle. It is somewhat surprising that we can use a SAT algorithm to find optimal solutions to planning problems because the complexity of optimal planning is higher than that of SAT, which is well known to be in NP.

When trying to use an algorithm for a problem besides the one that it was designed to solve we must first translate from instances of the problem into inputs for the algorithm. We then run the algorithm and translate from the algorithm output into the correct solution for the problem. In order for this to be useful, the translations must be efficient. There will not always be an efficient translation from a specific problem to a given algorithm, thus there are limitations on what an algorithm is capable of solving.

In an attempt to capture the scope of an algorithm, we define the *capability* of an algorithm in terms of the complexity of problems that it can solve. Informally, we say that an algorithm is X Capable if it can solve all problems in complexity class X using efficient transformations of the input and the output. We define capability in these terms because an algorithm that can solve one problem ought to be able to solve other problems in the same class since problems within a class are efficiently reducible to one another.

Capability is defined in terms of complexity, but there are important distinctions between the two concepts. The most commonly used algorithm for solving SAT is capable of solving problems beyond NP even though SAT is NP Complete. We will elaborate on the differences between capability and complexity later in the paper.

The notion of capability is important for at least three distinct reasons. The first is that knowing the capability of an algorithm helps us determine which problems it will be able to solve without resorting to trial and error. When an algorithm is incapable of solving a problem, knowing the capability provides insight into what subset of the problem the algorithm can solve. The second reason that capability is useful is that it may assist us in determining the complexity of problems. If an algorithm is no more than X capable and it can solve a given problem, then the problem must be in X. Finally, the capability of an algorithm helps us determine if it is more powerful than what is necessary for solving a problem. An algorithm that is more powerful than necessary may have additional machinery that will hinder its performance.

This theoretical framework proved useful in developing a faster solution for optimal planning using SAT. We prove that DPLL is exactly Δ_2 Capable where Δ_2 is the set of problems solvable in polynomial time where we are allowed to make a polynomial number of queries to an NP oracle. This information combined with the facts that optimal planning is Θ_2 Complete[1] and $\Theta_2 \subseteq \Delta_2$ show DPLL is capable of solving optimal planning.

In the next section we describe our modifications to Satplan in more detail. In section three we provide a formal definition of algorithm capability and give some basic examples. Following that we show how these ideas were used in developing a more efficient solver for optimal planning. The last two sections contain experimental results along with summary and conclusion.

A Faster Method of Optimal Planning

As mentioned earlier, one problem with using Satplan for optimal planning that we address is the loss of learned information between calls to the SAT solver. We could instead make a single call by generating the entire plan graph up front and using a predefined branching order to guarantee optimality. When converting to a satisfiability instance, we would introduce new variables G_i which would mean that the goal state has been reached at time i. We add the necessary clauses to the formula to ensure that these variables have meaning.

When we call the SAT solver, we branch on the variables G_1, G_2, … in order so that the first plan that we find is guaranteed to be optimal. We are effectively doing the same as Satplan by first searching for a plan of length 1, then of length 2, and so on until we find one. However, this method would automatically retain learned information because we are pushing the search into one SAT call.

Generating the entire plan graph requires too much memory so this approach is not practical. If we have a reasonable upper bound on the plan length then we could solve the problem in a single call by generating the plan graph up to the upper bound, translating to a SAT problem, and solving. The fixed branch order ensures that we find an optimal solution regardless of how large the upper bound may be. This approach compares favorably to Satplan on a set of problems taken from the IPC-5 benchmarks. Satplan allows us to specify which SAT solver we would like to use. The SAT solvers are independent from Satplan, not built in to it. For our experiments, we used both Tinisat (Huang 2007) and RSAT (Pipatsrisawat; Darwiche 2007).

Now that we have described the idea behind our solver, we will outline the actual modifications that were made to Satplan. The differences are:

- We introduced one new variable for each time step and clauses that ensure that the variable is true iff all of the goals have been satisfied at that time step.
- We modified Satplan to write one extra line to the cnf file which specifies the branch order.

[1] Technically optimal planning is not a decision problem whereas Θ_2 is a class of decision problems. Optimal planning is actually $F\Theta_2$ Complete, but this distinction does not matter for our purposes.

- We modified Tinisat and RSAT to accept a partial order on branching and to use these first before relying on their heuristics for branch decisions.
- Satplan will generate the plan graph up to some fixed upper bound on the plan length.

This is still not ideal because it assumes that we always have an upper bound on the plan length and it requires additional input. Rather than input an upper bound on the plan length, we can automate the process by starting with an initial guess, generating the plan graph up to that guess, and converting it to a SAT instance. If no solution is found then we multiply our previous guess by some constant (larger than one) and repeat. If a solution is found, it will be optimal because of the partial branching order. We use fixed parameters for the initial guess on the plan length and the multiplicative constant. These parameters are independent of the specific domain or problem being considered.

It is not a new idea to try to guess the optimal plan length and adjust accordingly. However, in previous attempts if a solution was found from the initial guess, there was no guarantee that the solution was optimal. The planner still had to try again with a smaller value of k to see if a smaller plan could be found. Ours is guaranteed to be optimal because of the partial order on branch decisions. Also, in previous versions k was incremented or decremented. We are multiplying it by some constant which allows us to converge on a solution more quickly.

Instead of generating the plan graph up to some fixed upper bound, we make the following additional change:

- Satplan will first generate the plan graph up to some guess k. If a solution is found it returns it as optimal. If no solution is found, we multiply k by some constant and try again.

For small values of k, we will not generate the cnf instance because we can determine from the plan graph that the goals are unreachable. There is a range of values where k is large enough that the goals can be reached in the plan graph (and hence we generate a SAT instance), but small enough that there is no plan of length k or less. There are no calls to a SAT solver before we enter this range, and the final call to a satisfiability engine will be when we have increased k beyond this range. Only when we are within this range do we make additional, unnecessary calls to the SAT solver. Depending on the size of the range and the multiplicative constant, we may skip over this range entirely and still end up making only one call. In most other cases, we end up making only a few calls because increasing k by some multiple each time will quickly put us beyond the range of values where we make additional calls. The overall result is that we make very few calls to a satisfiability engine while still maintaining optimality.

Algorithm Capability

Before defining algorithm capability, we must first introduce notation. Let A be an algorithm and let $A(x)$ denote the output of executing algorithm A on input x. Let I be the set of possible inputs to A and O be the set of possible outputs. Problems are typically formulated as languages over a finite alphabet Σ where a language is a subset of the strings that can be formed from the alphabet.

Many algorithms are nondeterministic, potentially having several execution paths running in parallel. Computers are inherently deterministic and must simulate nondeterministic algorithms by running the execution paths in sequence rather than in parallel. This imposes an order on the execution paths. In formalizing the notion of capability we allow for a fixed order on execution paths because computers are deterministic machines.

It may require an exponential amount of information to specify a partial order on execution paths. Alternatively, we could use a partial order on nondeterministic branches. This ordering is polynomial for all problems in NP. Let E be the set of all partial orders on the nondeterministic branches in the algorithm A. For example, in DPLL the set E corresponds to the set of all possible branching orders on the variables. While E might not contain all partial orders on execution paths, it captures a subset of the partial orders on execution paths without using exponential sized expressions of the partial orders.

Now that we have provided notation, we must state what it means for an algorithm to be applicable to a specific problem; after which we can define the capability of an algorithm in terms of the complexity of the problems that it can be applied to. Problems are formally represented as languages where an instance is in the language when the instance represents a yes answer to the problem. For example, the boolean satisfiability problem is the set of strings that represent satisfiable boolean formulae. To apply an algorithm to a specific language we seek polynomial computable transformations t_1 and t_2 that transform the input and output. t_1 maps from strings over the alphabet Σ (instances of the problem) to an input for A. This function may also produce a partial order on the nondeterministic branches. After the algorithm is run, the function t_2 maps from the output to either true or false indicating whether the string is in the language. An algorithm is X Capable if it can be used to solve every language in X. More formally:

Definition 1: Suppose we are given an algorithm A where I is the set of possible inputs, E is the set of partial orders on the nondeterministic branches, and O is the set of possible outputs. A is X Capable if for all languages $L \in X$ where $L \subset \Sigma^*$, there exists a pair of P-Computable functions $t_1:\Sigma^* \rightarrow I \times E$ and $t_2:\Sigma^* \times O \rightarrow \{0,1\}$ such that $w \in L$ whenever $t_2(w, A(t_1(w)))$ is true.

As mentioned earlier, the definition allows us to fix an order on the nondeterministic branches. This accounts for the fact that we are running nondeterministic algorithms on deterministic machines, which naturally imposes an order on the execution paths. Therefore, by demonstrating that an algorithm is X Capable, we have shown that we can apply the algorithm to all problems in X on a normal computer.

By our definition, DPLL is NP Capable. Satisfiability is NP Complete and thus there is a polynomial transformation from every problem in NP to a satisfiability instance. This satisfiability instance can be used as the input to DPLL. We let t_1 be the transformation to the SAT instance where no partial order on execution paths is necessary. We are also guaranteed to be able to answer the original query in polynomial time from the answer to the satisfiability instance, thus we also have our transformation t_2. In general, an algorithm that solves an X Complete problem is at least X Capable.

The NP Capability of DPLL is trivial. What is less intuitive is that NP is not an upper bound on the capability of DPLL. This is important because it highlights the difference between capability and complexity. It may initially appear that the only difference between converting to inputs of an algorithm and converting to instances of the problem is the fixed order on the execution paths. However, there are there are a few important distinctions between the two.

One difference is that an algorithm for a language can always be used to solve the complement of that language. In other words, X Capable is equivalent to co-X Capable. The language of unsatisfiable formulae and the language of satisfiable formulae can be recognized by DPLL even though one is co-NP Complete and the other is NP Complete. This does not mean that NP = co-NP, but a polynomial transformation from unsatisfiable formulae to satisfiable ones would.

Another difference is that a transformation to instances of a problem implies a transformation to inputs of an algorithm solving the problem, but the reverse is not necessarily true. Furthermore, an algorithm may have a wider range of application than the creator realized. There are algorithms that can be used for satisfiability that are also capable of solving QBF.

Note that since DPLL requires exponential time, there is no obvious reason why it cannot be applied to problems of a much higher complexity. However, the algorithm only uses polynomial space so we cannot use it for problems beyond PSPACE. It is not immediately apparent why we cannot use DPLL to solve every problem in PSPACE, but we show in the next section that DPLL can only be used for problems in Δ_2.

Applications of Capability to Planning

It is an unexpected result that we can use a single call to a SAT solver to solve optimal planning because the complexity of optimal planning is higher than the complexity of satisfiability. Yet, by proving that DPLL is Δ_2 Capable and that optimal planning is Θ_2 Complete[2] we can show that DPLL is capable of solving optimal planning. This section is divided into two parts: one devoted to analyzing the capability of DPLL and the other to determining the complexity of optimal planning.

Capability of DPLL

We begin by describing DPLL in more detail, after which we can prove its capability. DPLL is a recursive algorithm for solving satisfiability first introduced by Davis and Putnam (Davis; Putnam 1960) and later modified by Davis, Logemann, and Loveland (Davis *et al.* 1962). Starting with an initially empty assignment of variables, a solution is constructed by first returning the answer if it has already been determined, then adding all immediate consequences of the current partial assignment, and finally selecting an unassigned literal and making one recursive call with it set to true and one with it set to false. The immediate consequences of an assignment are the set of all clauses that contain one unvalued literal and no literals set to true. Clearly the unvalued literal must be set to true in order to satisfy the formula. The algorithm returns the satisfying assignment if one is found and returns UNSAT if a contradiction is reached. Pseudo-code is shown in Figure 1. Some versions of DPLL just return SAT or UNSAT instead of returning the satisfying assignment.

```
1: DPLL (Clauses, Assignment)
2:      Assignment ← UNIT-PROPAGATE (Assignment)
3:      if Clauses is empty
4:          then return Assignment
5:      if $c \in$ Clauses and $c$ is empty
6:          then return UNSATISFIABLE
7:      $l$ ← GET-NEXT-VARIABLE ()
8:      $S$ ← DPLL (Clauses $\{l = 1\}$, Assignment $\cup \{l\}$)
9:      if $S \neq$ UNSATISFIABLE
10:         then return $S$
11:     return DPLL (Clauses $\{l = 0\}$, Assignment $\cup \{\neg l\}$)
```

Figure 1. Pseudo-code for DPLL. Technically DPLL takes one argument which is a formula or set of clauses. From the one argument version, we call the two argument version presented here by giving it the set of clauses and an initially empty partial assignment.

Researchers have applied a variety of optimization techniques in order to make the algorithm more efficient. The optimizations that have been used range from caching and good data structures to branching heuristics, learning methods, and parameter tuning. After proving the

[2] See footnote 1.

capability of DPLL, we will discuss some of the later modifications and how they affect the capability.

Given the popularity and effectiveness of DPLL, it is useful to characterize the set of problems that it can be used to solve. This way when a new problem arises, we know whether or not we can apply DPLL based on the complexity of the problem.

In order to show that DPLL is exactly Δ_2 Capable, we prove that Δ_2 is both a lower and an upper bound on the capability of DPLL. To show it is an upper bound we prove that if a problem can be solved by DPLL then it must be in Δ_2. To show that it is a lower bound, it is sufficient to show that it can be used to solve a Δ_2 Complete problem.

Proposition 1. DPLL is at most Δ_2 Capable unless there is a collapse in the polynomial hierarchy.

Proof: We prove this proposition by showing that if DPLL is capable of solving a language, then the language must be in Δ_2. Let L be a language such that there exists polynomial computable functions t_1 and t_2, where $t_2(x, \text{DPLL}\ (t_1(x)))$ is true whenever $x \in L$. If we have access to a SAT oracle, we could solve L in polynomial time by running a procedure similar to DPLL except that we use the oracle to eliminate those calls that would result in UNSAT. The overall process requires polynomial time with a satisfiability oracle including the transformations t_1 and t_2, making L in Δ_2. ■

This establishes the upper bound on the capability of DPLL. We can prove the lower bound by proving that it can solve a Δ_2 Complete problem. This requires knowing that some problem is Δ_2 Complete. Many variations of satisfiability exist, some of which are not known to be in NP. One such problem is determining whether the lexicographically maximum satisfying assignment is odd. The Odd Maximum Satisfiability (OMS) problem can be more formally expressed as:

$$\text{OMS} = \{\psi(x) \mid x_n = 1 \text{ in max sat assignment of } \psi\}$$

where ψ is a boolean formula over the variables $x_1, \ldots, x_n$ and let x be shorthand for $x_1, \ldots, x_n$. Krentel demonstrated that OMS is complete for Δ_2. (Krentel 1988)

Next, we need to show that we can still use unit propagation which returns all consequences of the current partial assignment. If the current partial assignment is part of the maximum model, then any consequences of that assignment must still be valid. In other words, unit propagation only prunes sections of the search space that do not contain any models. More formally:

Proposition 2. If a particular branching order arranges the leaves of the search tree from maximal to minimal, performing unit propagation will not cause us to incorrectly return a non-maximal model.

Proof: Unit propagation may rearrange the leaves of the search tree and possibly in such a way that the order on interpretations is no longer respected, but we argue that the output is not adversely affected. Suppose that there are two interpretations I_1 and I_2 with I_1 being less than I_2. If neither interpretation is a model of the theory then these two leaves can be swapped because neither will be returned. Similarly, if one is a model and the other is not, then we can interchange them because we will not return the non-model.

The only interesting case occurs when both interpretations are models of the theory. In order for unit propagation to force us to incorrectly return I_1, both interpretations must reside in the subtree rooted at some unit propagation. Furthermore, I_1 must be in the left half of the subtree and I_2 in the right half because unit propagation does not rearrange the leaves within the left and right halves. This leads to a contradiction since we assumed that both interpretations are models of the theory. The leaves in the right half are pruned nodes that cannot be models of the theory. Thus, unit propagation does not cause us to incorrectly return a non-maximal model. ■

Proposition 3. DPLL is at least Δ_2 Capable.

Proof: All problems in Δ_2 can be efficiently reduced to any Δ_2 Complete problem. Demonstrating that DPLL can solve a Δ_2 Complete problem is sufficient to show that DPLL is Δ_2 Capable. We show that DPLL can be used to solve Odd Maximum Satisfiability (OMS) which is known to be Δ_2 Complete.

As mentioned earlier, an order on the execution paths corresponds to ordering the branch variables in the procedure. To determine the lexicographic maximum satisfying assignment we branch on the variables in order and we always attempt to set a variable to true first. t_1 returns the satisfiability formula and the order of the variables $(x_1, \ldots, x_n)$. Proposition 2 tells us that we can still apply unit propagation without any negative consequences even though it rearranges the branching order.

After running the DPLL algorithm, t_2 will take the satisfying assignment produced and check if $x_n = 1$. The order on the variables ensures that the satisfying assignment returned is the lexicographically maximum one. The functions t_1 and t_2 are both polynomially computable, thus DPLL is capable of solving OMS and all of Δ_2. ■

The following is a direct consequence of propositions 1 and 3:

Proposition 4. DPLL is exactly Δ_2 Capable. ■

DPLL can easily be modified to solve Quantified Boolean Formula (QBF) (Cadoli *et al.* 1998) where QBF is known to be PSPACE Complete. It is not difficult to show that this modified version of DPLL is exactly PSPACE Capable, but we will omit the proof here due to space restrictions.

Notice that our proof requires that DPLL return the satisfying assignment if one exists. The version that just returns SAT or UNSAT might not be Δ_2 Capable. We now turn the discussion to variations of the original DPLL algorithm and which versions might not be Δ_2 Capable.

Many modifications have been made to the original DPLL algorithm to make it more efficient. The techniques that have been added can still be used in solving Δ_2 problems if they do not change the order in which the models are discovered. Any modification which reorders the search tree is unusable unless we can devise a method to ensure that the first model found is maximal. We would like to analyze which of the additional techniques can still be used and which ones cannot without adding additional processes to the algorithm. For the ones that cannot be used, we also analyze how this affects the overall performance.

Many optimizations of DPLL involve using intelligent branching heuristics which select the next branching variable in an attempt to minimize run time. One of the more popular branching techniques is VSIDS (Moskewicz *et al.* 2001) which will assign an initial value to each variable based on the number of clauses that it appears in. When new clauses are learned, the score is incremented. Periodically all of the scores are divided by a constant so that the score reflects the number of occurrences with a higher weight on more recently added clauses.

The proof that DPLL is Δ_2 Capable requires a specific branching order to solve OMS. Unless there is another proof that does not require a fixed branching order, we are unable to use the intelligent branching heuristics. However, if only a partial order on branching is required then we can use heuristics afterwards. The addition of a branching heuristic changes the algorithm from a nondeterministic algorithm to a deterministic one, which impacted the capability.

One of the more efficient implementations of unit propagation involves watched literals. (Moskewicz *et al.* 2001) For each clause, we pick two unvalued literals to watch. If either of those literals is set to false while the other is unvalued, we select a new literal to watch if possible. If this is not possible, then either the clause is already true (in which case we do nothing) or the clause is a unit clause (in which case we perform unit propagation).

If at any point, a watched literal is set to false and all other literals are already false, we return 'unsatisfiable'. The watched literal invariant that must be maintained is that for every clause either one of the literals is true or both are unvalued. Watched literals are primarily a book keeping device designed to improve the performance of the unit propagation procedure. This technique does not rearrange the search space and therefore can still be used.

Another technique that is present in some solvers is called pure literal propagation. If a literal occurs in a formula and its negation does not, this variable can be set accordingly. We cannot use this technique to solve the odd max sat problem because the maximum satisfying assignment may set pure literals to false. This is not a significant setback because people tend to agree that pure literal propagation requires more time than it saves. It is costly to perform searches for pure literals and few instances contain enough pure literals to achieve a performance advantage. (Zhang; Malik 2002) Most solvers omit this technique.

One technique that is well used by most modern solvers is some form of learning. The clauses that are learned are a direct result of the original formula. When we add clauses to the formula the result is identical to the original formula and has the same set of models. Whenever we learn a new clause we backtrack to the point at which it would have become unit and begin unit propagation. Since the learned clause is a direct consequence of the original formula, this will only prune sections of the search space that cannot contain any models. While it rearranges the search space, it only groups nonmodels so that they can be pruned together. To look at it another way, the original formula plus all of the learned clauses is identical to the original. The search tree may be a little different because we are inserting unit propagations higher into the tree, but since we still use the same partial branch order, the ordering of the models does not change. Being able to use this technique is critical in gaining large performance advantages over the original Satplan algorithm.

Complexity of Optimal Planning

Planning problems can be arbitrarily complex since some problems require plans of exponential length, thus planning is EXPTime. While it is beyond PSPACE to find a plan for an arbitrary problem, determining plan existence is known to be PSPACE Complete. (Bylander 1991) There are a number of restrictions, such as a polynomial bound on plan length, that reduce the complexity of planning from PSPACE to NP. If we know that the plan length is bounded by a polynomial, then the plan serves as a polynomial sized witness that the planning problem is solvable. Hence polynomial bounded planning is in NP.

In Satplan, we can generate the satisfiability instances in polynomial time if the plan length is polynomially bounded. In general, Satplan may generate exponential sized SAT instances. The satisfiability instances are only guaranteed to be polynomial when the planning problem is in NP.

Whenever it is NP Complete to determine whether or not there is a plan, it is also NP Complete to ask if there is a plan of length *k* for some fixed *k*. However, it is harder than NP to ask for the length of the optimal plan. Similarly the complexity of finding an optimal plan is higher than the complexity of finding any plan. The following proof characterizes the complexity of optimal planning:

Proposition 5. Optimal planning is $F\Theta_2$ Complete when the plan length is polynomially bounded.

Proof: It is in NP to ask whether there is a plan of length *k* for some fixed *k*. Since the plan length is bounded by some polynomial $p(n)$, we can use a divide and conquer technique to recursively divide the search space in half at with each oracle query. We initially ask if there is a plan of length ½ $p(n)$. The answer to this question eliminates half of the search space. Thus, the overall number of queries is $O(log\ p(n)) = O(log\ n)$.

Next we show that optimal planning is $F\Theta_2$ Hard by reducing from a know $F\Theta_2$ Complete problem. The decision version of clique asks for a given graph G and integer *k* whether there is a clique of size *k*. Determining the size of the largest clique is $F\Theta_2$ Complete. (Krentel 1988)

In order to reduce the problem of finding the size of the largest clique to optimal planning we need to construct a set of variables, a set of actions, a start state, and a goal state. Informally, the actions that we will use are removing vertices, the start state is the initial graph, and the goal state is a complete graph on *k* vertices.
More formally, let the variables be:

- $edge(v_i, v_j)$ meaning there is an edge between v_i and v_j.
- $removed(v_i)$ meaning that the vertex v_i has been removed from the graph.

The only action that we use in the construction is $remove(v_i)$ which has no preconditions and the postconditions are $removed(v_i) \wedge \forall j \neg edge(v_i, v_j)$. The start state is that $removed(v_i)$ is false for all *i*, and $edge(v_i, v_j)$ is true if there is an edge between v_i and v_j in the initial graph. The goal state is that we are left with a complete graph: $\forall i,j\ removed(v_i) \vee removed(v_j) \vee edge(v_i, v_j)$. In other words for every pair of vertices, either one of them has been removed or there is an edge between them. The plan with the fewest number of actions removes the fewest number of vertices resulting in the largest clique. To obtain the size of the largest clique we subtract the length of the optimal plan.■

Even though optimal planning is harder than SAT we can still use a single call to DPLL to solve it because DPLL is capable of solving all problems in Δ_2. See Figure 2 for a complexity diagram illustrating the relationship between the various classes and problems.

As mentioned before, the complexity of deterministic planning is beyond NP. We can still use our method to solve optimal planning in the general case, but the transformations are not guaranteed to be polynomial.

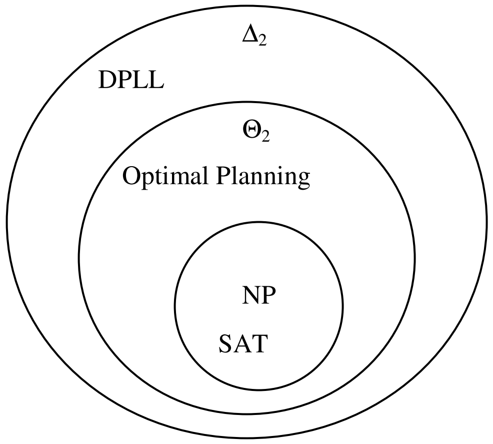

Figure 2. Complexity Diagram

Experimental Results

For our experiments we compared our modified version of Satplan to the original, unmodified version. Satplan uses a command line option to specify which SAT solver is being used. We compared our planner to Satplan using Tinisat for both as well as using RSAT for both. We used the fully automated version that will start from a fixed upper bound value of *k* and multiply *k* by some constant each time that a plan cannot be found.

We named out planner "Cricket" because it takes large jumps through the search space instead of stepping through each possible plan length until finding one. We selected a random set of problems from each of the domains used in the 2006 planning competition. We also used a set of blocksworld problems. The results are shown in Table 1 comparing our planner to the original Satplan. We outperformed Satplan on most of the instances and some by a considerable margin. It is also worth noting that in the one domain where Satplan does better, Cricket solved additional instances where Satplan timed out. These instances are not included in the results shown in Table 1. We also want to note that the results on RSAT are incomparable to the results on Tinisat because we used a different set of randomly selected instances from the domains.

Summary and Conclusions

We made modifications to Satplan that make it better at finding optimal solutions more quickly. Our algorithm first generates the plan graph up to some fixed initial length *k*. If a plan is found in plan graph generation then we return it. If the goals are unreachable we multiply *k* by some fixed constant and start over. Otherwise we convert the plan graph into a satisfiability problem and use the predetermined branching order to guarantee optimality. If the problem is satisfiable, then we convert the satisfying assignment into a plan. Otherwise we multiply *k* by some

Planner / Domain	Satplan - RSAT	Cricket - RSAT	Satplan - Tinisat	Cricket - Tinisat
Blocksworld	3025.31	1417.31	3113.54	1740.06
Pathways	164.38	189.64	354.29	195.36
Pipesworld	845.36	506.91	869.21	637.91
Rovers	458.27	253.08	447.92	256.4
Storage	702.24	473.29	2510.96	282.31
TPP	872.2	718.21	799.41	700.5
Trucks	302.19	140	605.25	279.95

Table 1. Cricket vs. Satplan.

fixed constant and start over. This enables us to skip over large sections of the search space because we do not need to create a separate SAT instance to check every possible plan length. This way we also retain learned information automatically because we search for a plan of length *k* and one of length *k*+1 in the same satisfiability instance.

We ran the modified version of Satplan against the original version of Satplan on a set of problems taken from the IPC-5 benchmarks. Our modifications resulted in dramatic runtime improvements over the original Satplan algorithm. For one of the problem domains we saw an order of magnitude improvement over the time required by Satplan.

Our results are interesting because while we are making fewer calls to the satisfiability solver, we are also generating a larger plan graph than necessary and hence a larger SAT instance. Using a predefined branching order also has the potential to slow down the solver. Our results indicate that these factors are outweighed by the benefits gained from relying on the optimizations built in to the SAT solver. Retaining learned information is probably the biggest factor, but other factors may play a role as well.

We developed a new theoretical framework that led us to our solution for optimal planning. We define the capability of an algorithm in terms of the complexity of problems that it can be used to solve. By proving that DPLL is Δ_2 Capable and that optimal planning is $F\Theta_2$ Complete, we conclude the surprising result that we can solve optimal planning with a SAT algorithm even though optimal planning has a higher complexity.

One interesting direction for this research would be to try to determine if the performance benefits are entirely due to the retention of learned clauses or if some of it is related to other elements of the SAT solvers. It would also be interesting to see how much the performance would improve by tuning parameters.

References

Bylander, Tom. 1991. Complexity Results for Planning. *International Joint Conference on Artificial Intelligence*.

Cadoli, Marco; Giovanardi, Andrea; Schaerf, Marco. 1998. An Algorithm to Evaluate Quantified Boolean Formulae. *American Association for Artificial Intelligence (AAAI).*

Davis, M.; Logemann, G.; Loveland, D. 1962. A Machine Program for Theorem Proving. *Communications of the ACM Volume 5 Issue 7*, pages 394-397.

Davis, M.; Putnam, H. 1960. A Computing Procedure for Quantification Theory. *Journal of the ACM* 7, 1:201-215.

Huang, Jinbo. 2007. A Case for Simple SAT Solvers. *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming.*

Kautz, Henry; McAllester, David; Selman Bart. 1996. Encoding Plans in Propositional Logic. *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning.*

Kautz, Henry; Selman, Bart. 1992. Planning as Satisfiability. *Proceedings of European Conference on Artificial Intelligence.*

Kautz, Henry; Selman, Bart; Hoffman, Joerg. 2006. Satplan: Planning as Satisfiability *Abstracts of the 5th International Planning Competition.*

Krentel, M. The Complexity of Optimization Problems. 1988. *Journal of Computer and System Sciences*, 36:490-509.

Moskewicz, Matthew W.; Madigan, Conor F.; Zhao, Ying; Zhang, Lintao; Malik, Sharad. 2001. Chaff: Engineering an Efficient SAT Solver. *Proceedings of the Design Automation Conference.*

Nabeshima, Hidetomo; Soh, Takehide; Inoue, Katsumi; Iwanuma, Koji. 2006. Lemma Reusing for SAT based Planning and Scheduling. *Proceedings of the 16th International Conference on Automated Planning and Scheduling.*

Pipatsrisawat, K.; Darwiche, A. 2007. RSat 2.0: SAT Solver Description. Technical report D153. Automated Reasoning Group, Computer Science Department, University of California, Los Angeles.

Zhang, L.; Malik, S. 2002. The Quest for Efficient Satisfiability Solvers. *Invited Paper, Proceedings of the 8th International Conference on Computer Aided Deduction.*