An Evaluation of Blackbox Graph Planning

John Duchi, James Connor, and Bruce Lo
jduchi@stanford.edu, jconnor@stanford.edu, brucelo@cs.stanford.edu,

Stanford University
CS227: Assignment 4
June 4, 2005

I. Introduction

In this paper, we investigate a satisfiability-based approach to solving STRIPS planning problems. Traditionally, popular algorithms for STRIPS planning problems include a stable of total-order planners, such as Prodigy, and partial-order planners, such as UCPOP. Blum and Furst introduced a novel way of constructing solutions for these problems called Graphplan, which significantly outperformed previous approaches [1]. Arguing that SAT solvers are more robust and allow for more expressiveness than specialized planning algorithms, others have attempted to convert STRIPS planning problems into SAT problems, employing a variety of SAT algorithms, and gaining results competitive with those of Graphplan. SATPLAN, one of the early attempts, converted STRIPS planning problems directly into CNF, sometimes even with hand-coded axioms to fine-tune performance.

Combining the strength of Graphplan in defining search space and the strength of modern research into SAT solvers, Kautz and Selman introduced Blackbox. Blackbox first constructs a planning graph with methods similar to Graphplan and then converts the planning graph into a CNF formula to be fed to a SAT solver [3]. By converting from a planning graph, instead of from the STRIPS planning problems directly, they were able to show improvements over both Graphplan and SATPLAN, in addition to the benefits of automatic SAT encoding without the need of hand-coded axioms.

For this project, we implemented the Blackbox planning algorithm, investigating its efficacy on a range of planning problems from the logistics, blocks world, and transport domains. We also investigated the impact of different constraint propagation methods, including mutex propagation in the planning graph and unit propagation and the failed literal rule in the SAT encoding of the problem.

II. STRIPS Planning

For the sake of completeness, we describe briefly STRIPS planning. A STRIPS planning problem consists of the following:

- A set of objects that persist throughout the span of the planning process. Objects can neither be created nor destroyed during planning or the execution of a plan.
- A set of operators that take objects as parameters. Operators have preconditions, add-effects, and delete-effects. The preconditions of an operator are the set of propositions that must be satisfied when the operation is carried out. Add-effects

are the set of propositions made true after an operation occurs, and delete-effects are the set of propositions that an operation makes false.
- A set of initial conditions in the form of propositions true at time step 0.
- A set of goals in the form of propositions that must be true at the end of the plan.

Because STRIPS operates under the closed world assumption, any un-instantiated actions or propositions are asserted to be false, and an action is a fully-instantiated operator with specific objects as parameters. Additionally, STRIPS assumes discrete time steps. A valid plan needs to specify a set of actions and their execution times (levels in the plan) such that the initial conditions hold at time 0, actions are executable, and all goals are met at the last time step of the plan.

III. Data Structures

To represent the planning problems and the resulting planning graph, we used the following data structures:

Class	Members	Description
Object	name	The name of this object
Proposition	name	The name of this proposition
	precs	List of actions of which this proposition is a precondition
	adds	List of actions that have this proposition as an add-effect
Operator	name	The name of this operator
	precs	List of propositions that are preconditions of this action
	adds	List of this action's add-effects
	dels	List of propositions that this action deletes
Instantiated	idx	Index into the array of possible actions or propositions
	args	The objects that instantiate this action or proposition

Figure 1: Data Structures

We used the Instantiated class to represent both an instantiated proposition and an instantiated operator (i.e. action). To facilitate internal bookkeeping, we assign every object an index, every proposition an index, and every operator an index. In addition, because of the relatively small number of propositions, we are able to assign every possible grounded proposition an index (grounded propositions are those with specific lists of objects as arguments—we will use grounded interchangeably with instantiated). This is very useful for our bit array implementation of the planning graph construction (see Section IV).

For every level in our planning graph, we maintain the actions that are instantiated at that specific level, the propositions instantiated there, and the mutex relationships. We are able to maintain both proposition mutexes and action mutexes, supporting constant time queries as to whether propositions or actions are mutex with one another.

IV. Planning Graph Construction

Graphplan uses a polynomial time algorithm to construct a graph representing all possible paths through the planning problem. Below is a short description of planning graph construction.

```
function ConstructPlanningGraph(Ops, Objs, InitState, Goals)

  Initialize G with proposition level 1 using InitState

  for i = 1 incrementing by 1 do
    if (proposition level i contains all Goals
        and no two goals are mutually exclusive at level i) then

      return G

    endif

    Augment G with action level i
    Augment G with proposition level i+1
  endfor

end ConstructPlanningGraph
```

Figure 2: Pseudo-code for planning graph construction

The run time requirement of planning graph construction is polynomial in n, m, p, l, and t, where

n	number of objects
m	number of operators
p	number of propositions
l	length of the longest add-list of any of the operators
t	t-level planning graph
k	largest number of formal parameters in any operator [3].

We will be noting the running time requirement of individual parts of the planning graph construction process below.

<u>Proposition Levels</u>

We used bit arrays to represent propositions that could be instantiated at each proposition level of our graph. Because we index every possible grounded proposition, each proposition level can be a bit array whose indices are 1 if the corresponding proposition is (or might be) instantiated, 0 otherwise. For example, for the proposition ON in the blocks world domain could be grounded with objects A, B, and C: grounded propositions for the ON proposition would include the relations ON(A,B), ON(A,C), ON(B,A), ON(B,C), ON(C,A), and ON(C,B).

Assuming we have *p* propositions and *n* objects in our problem and that the largest number of arguments for any proposition in the problem is *q*, the total number of possible grounded propositions at any level is $g = pn^q$. Utilizing bit arrays and allocate one bit array of size *g* for each proposition level, we can very quickly tell which propositions are instantiated at any level.

Naturally, the first proposition level in the graph has exactly the grounded propositions specified in the initial state of the problem, and the last level has as a subset the goal propositions. Because of the bit array implementation of our proposition levels, we can very quickly detect goal states and more easily instantiate action levels, since calculating intersections and subset relations is fast.

Propositions to Actions

As shown in the ConstructPlanningGraph pseudo-code, the planning graph construction process repeatedly alternates between constructing action level t from proposition level t and constructing proposition level t+1 from action level t. Proposition level t is represented by a set of grounded propositions possible at level t, while action level t contains the set of actions that we can instantiate, based on their preconditions, at level t.

Initially, we have only the initial state propositions at proposition level 0. To go from proposition level t to action level t, we must find all actions whose preconditions appear in proposition level t. This is an easy concept to define but a rather time-consuming process calculation (though it is in the worst case polynomial, which is still far faster than searching for a valid plan). In the most naïve implementation, we consider each operator, instantiating each with all possible permutations of the objects, then checking each operator's preconditions against proposition level t. Assuming that the largest size of precondition list is c, the naïve implementation takes $O(gcmn^k)$, where m is the number of operators, n the number of objects, and k the maximum number of arguments to any operator.

Bit arrays help speed up this check. While we still must loop through all operators and instantiate each operator with possible permutations of the objects, we need not checking each precondition in each of the possible instantiated actions against proposition level t. We maintain a few static bit arrays of size g for quick calculations, and for each action a, we set a's grounded preconditions to be on bits in the array b_{pre}. This allows us to quickly check we can find if the precondition bit array b_{pre} is a subset of the proposition level t bit array b_{prop} by checking if $b_{pre} = b_{prop} \cap b_{pre}$. This operation is a very fast $O(g)$; while we do not lower our big-O running time, the constant factors are easy to deal with. One possible speedup for this procedure is to have propositions keep track of those actions that have them as preconditions; then we could iterate over the propositions that had been instantiated at a level t, constructing as we went a list of those actions whose preconditions had been instantiated. The efficiency gain here, given the speed of the construction regardless and the NP-Completeness of the SAT procedure underlying the planner (to be described later), was thought to be insignificant.

Having calculated all possible instantiations for the actions at the next level, we can populate the action level from our propositions. Unlike our proposition levels, our action levels contain a list only of instantiated actions at each level instead of a bit array with indices corresponding to a list of all possible instantiated actions. The structure of the problems dictated this approach—while propositions had at maximum 2 arguments, actions could have many more, and thus the ordering became undesirable.

Actions to Propositions

The construction of the proposition levels from action levels is a much more straightforward exercise. In order to construct the set of activated propositions at level t+1, we examine the set of instantiated actions at level t and add all of the add-effects of all of the instantiated actions. This process that can be performed in time O($p + mln^k$).

Mutexes

Mutex relationships define pairs of actions or propositions that cannot be true at the same time in any plan. Calculating, and propagating, mutex relations can save much search and significantly speed up planning procedures. There are three different types of mutex relationships, two for actions and one for propositions. Actions have interference, which is when one action a_1 has as an add-effect what another action a_2 has as a delete or when an a_1 has as a precondition what a_2 has as a delete. Actions also have competing needs, which occurs when action a_1 has a precondition that is mutex with the precondition of action a_2. For propositions to be mutex, the criteria is slightly more difficult to meet: a proposition p_1 is mutex with a proposition p_2 if every action (including no-operation actions) that instantiates p_1 is mutex with every action instantiating p_2. Mutexes can be shown to be monotonically decreasing in the sense that once a mutex relation exists, if it ceases to exist at a level t, it can never hold at a level greater than t.

The ordering we imposed on the propositions and our ability to represent a set of propositions as a bit array facilitated significantly the calculation of mutex relationships. To calculate whether an action a_1 was mutex with an action a_2 because of interference, we simply constructed the bit arrays corresponding to their add, delete, and precondition effects, took the appropriate intersections, and if any intersection was non-empty, a_1 and a_2 were determined to be mutex. We then maintain, for each action level, a set of bit arrays, each as long as the number of instantiated actions and corresponding to the mutexes an action has with other instantiated actions. This set allows O(1) mutex checking once it has been created.

Propagating mutex constraints is somewhat more challenging than simple interfering actions. For each proposition p_t at level t, we maintained a vector of the actions at level t-1 that had p_t as an add effect. We then iterate over all pairs of propositions, checking if any of the actions that instantiate them are not mutex. While the complexity of this check is O(g_2A^2), where A is the number of actions that have been instantiated (which can grow as large as mn^k), in practice we quickly find non-mutex actions and note that pairs of propositions are not mutex. As with the actions, we maintain a set of bit arrays indicating mutex propositions.

Checking whether actions are mutex based on competing needs is less tricky. To decide if a_1 and a_2 are mutex because of competing needs, we iterate over all pairs of their preconditions, whose mutex relations we can check in constant time, and if one pair has propositions that are mutex, a_1 and a_2 are mutex.

In spite of all the data and information we keep around, maintaining bit arrays at every level for all mutex relations, all propositions, and a few other indexing pieces of data, the size of our planning graphs are relatively small, growing only to 2 MB or so in the largest cases.

Visualization of Mutexes

We attempted to visualize the action and proposition mutexes. Seeing the mutexes gives us three pieces of information that are difficult to find when Blackbox is running. First, the visualization shows us how dense or sparse the instantiated actions and grounded propositions are in the total space of possible actions and grounded propositions. Second, we see how dense the mutex relations are in the space of instantiated actions and grounded propositions. Third, we see how the number of instantiated actions, grounded propositions, and mutexes evolves at each step of the planning graph construction.

Below, we show the visualization of action mutexes and grounded proposition mutexes for the FERRY3 problem. Each cell in the diagram represents an action or a grounded proposition pair. Gray cells indicate that the proposition or action at that index has not been instantiated, white means there is no conflict between the instantiations to which the cell has indexes, and red shows there are mutex relations. The length of each axis in the action mutex diagram is $|actions| * |objects|^4$. The lengths of the axes in the proposition mutex diagrams are $|propositions| * |objects|^2$.

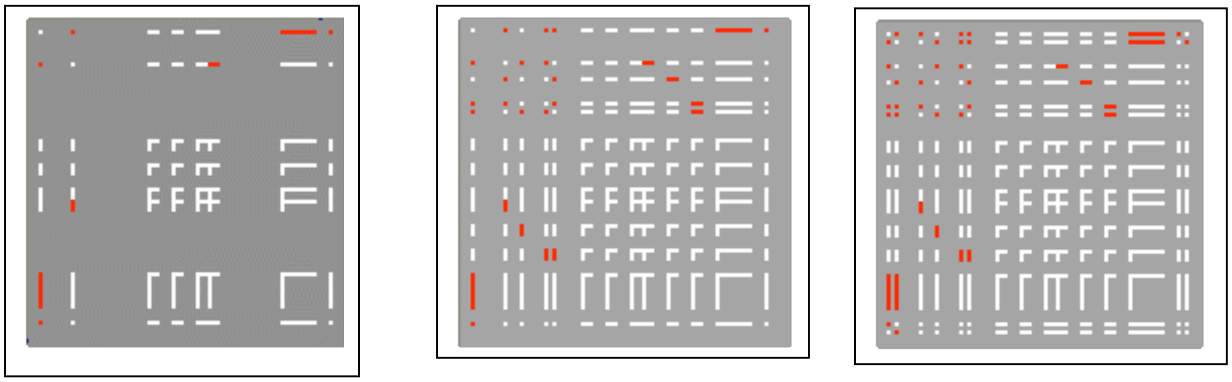
Figure 3: Action Mutexes at levels 0, 1, and 2 for FERRY3

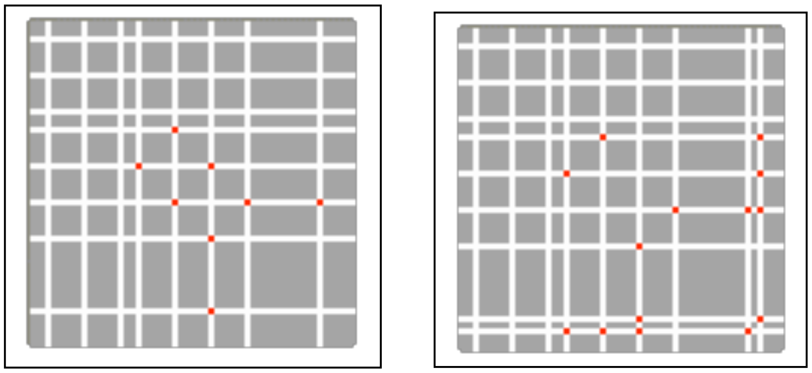
Figure 4: Proposition Mutexes at levels 1 and 2 for FERRY3

The above diagrams show that actions and proposition instantiations are not extremely dense. In addition, some action mutexes persist because of interference, whereas proposition mutexes might persist or go away, but once propositions do not conflict, they never will.

V. Conjunctive Normal Form Conversion

Conjunctive Normal Form (CNF) is an encoding of a wff made of conjunctions of clauses, where each clause is the disjunction of some number of literals. Converting a planning graph to CNF is a key step in the Blackbox procedure; we must add clauses that capture all the information the planning graph encapsulates. This representation is rich enough that we can describe all mutex relationships, even implying some that are not captured in the planning graph. Each level defines a set of variables that are included in the CNF representation, their truth-values to be computed; every true variable implies that its corresponding proposition or action, at a specific level, must have been true.

For each initial state proposition and each goal state proposition, we add a singleton clause to ensure that the propositions are true. For each proposition p, we must add clauses describing the actions that might have added p, which in effect gives us the formula (p -> $a_1 \vee a_2 \vee a_3 \vee \ldots \vee a_n$). To encode this in CNF, we translate to the formula ($\sim p \vee a_1 \vee a_2 \vee a_3 \vee \ldots \vee a_n$). For each action a at each level, too, we must add clauses describing the preconditions of a. Thus, each action implies the conjunction of its preconditions: (a -> p_1 ^ p_2 ^ … ^ p_n). Using distribution of disjunction over conjunction, this can be encoded in clauses of form: ($\sim a \vee p_i$) for each p_i in a's preconditions. A mutex relationship simply means that both its arguments cannot be true at once, so we encode mutex actions a_1 and a_2 in clauses of form ($\sim a_1 \vee \sim a_2$). Mutex propositions are symmetric. Of these two types of mutexes, we need only include action mutexes, since they imply proposition mutexes. Nevertheless, past research has suggested that including proposition mutexes can significantly speed search.

Because of the nature of mutex relations and the possibility that they grow quadratically with the number of actions and propositions instantiated at each level, mutex propagation can make CNF encodings of a planning graph grow very large. For example, in some blocks world problems, when propagating mutexes, we encountered CNF representations with as many as two million clauses. This blow-up is not insurmountable, but it does make problems more difficult.

VI. WalkSAT

The backbone of the planning procedure we will describe sparsely; for a better explanation, see Connor and Duchi's 2005 treatment of satisfiability algorithms [2]. Once given a CNF representation of a problem, WalkSAT will give a random truth assignment to all the variables, then will pick a random unsatisfied clause. With probability p, WalkSAT will pick a random variable from this clause, with probability 1-p, WalkSAT will pick a variable greedily and attempt to minimize the number of unsatisfied clauses. WalkSAT then flips the truth assignment of this variable, checking to see whether the CNF is then satisfied.

WalkSAT requires that a certain number of attempts to solve a CNF be given (usually referred to as the number of tries) and that a certain number of different variable assignments be allowed (the number of flips). For our experiments, as suggested by previous results on WalkSAT, we set the number of flips to be $2num\_var^2$, the number of tries to be twenty. We used the same implementation of WalkSAT, with a few modifications for reading in instances of a graph from the graph planner, as we did in [2].

VII. Augmenting Search with General Limited Inferences

Unit Propagation

Unit propagation finds unsatisfied clauses with one unassigned variable and assigns a truth value to that variable to satisfy the clause. When it assigns the truth value, unit propagation detects whether the assignment made any other unsatisfied clauses have one unassigned variable and adds such clauses to a queue of clauses to be unit-propagated. Given that once a variable has been propagated, its assignment is set; WalkSAT should not change it. We modified WalkSAT so that it would not change unit-propagated literals (or those that were chosen by the failed literal rule). Unit propagation has proved to be very effective for limiting the search in CNF problems generated from planning; given the unit clauses that define the goals and initial state, it seems intuitive that unit propagation would have at least some limited strengths. In fact, unit propagation often allows us to break out of search at a particular level in the planning graph early if there are obviously no solutions because we can propagate in polynomial time to a contradiction.

Failed Literal

In checking for failed literals, we assign each variable to true (and again to false to re-run the test) and then unit propagate this assignment to check for a contradiction. If a contradiction is implied, then we know that the truth assignment we made is impossible, and we force the opposite truth value on the variable in question. If we get contradictions from both propagations, we know the formula cannot be satisfied. In effect, this rule adds some marginal completeness to WalkSAT, at least for obviously unsolvable instances of a SAT problem.

Unit propagation and checking for failed literals speed up search because they find simple implications that would otherwise waste the time of the SAT solver. This is especially significant for a randomized SAT solver like WalkSAT. The structure of CNF formulae produced from a planning graph lends itself to unit propagation and checking for failed literals. There are many unit clauses (e.g. initial and goal states) and binary clauses (e.g. mutexes and actions implying their preconditions) that are sensitive to small truth assignment modifications. If all clauses contained many variables, we would have trouble finding any initial unit clauses to bootstrap unit propagation and clauses' evaluations would be less sensitive to assigning truth assignments when we checked for failed literals. There are, however, small clauses off of which we can bootstrap.

VIII. Results

Combining the SAT solving with Graphplan, we implemented Blackbox[3], which proved to be a relatively effective planning tool. We ran tests on a suite of planning problems from traditional planning domains. These problems ranged from blocks world planning problems and single robot arm domains to logistical planning and toy problems without any objects. Each domain presented some specific challenges, some more difficult than others.

Timing Data

The run times for our different problems varied widely, which is understandable, given the differences between each of the problems. On average, blocks world problems were more difficult than problems of similar length, though all problems whose solutions were longer than twelve steps provided significant difficulty for us.

Problem	**Time**	**Length**
att-bw2	25.15	4
att-log	1.58	10
att-log1a	0.26	6
big-bw1	4.01	7
d1s1-8	0.09	8
d1s1-10	0.21	10
d1s1-4	0	4
ferry3	0.03	3
ferry7	0.25	7
med-bw	31.3	5

Figure 3: Average run times, in seconds, for consistently solved problems

The most efficient solving method, by far, was using a Graphplan that propagated mutex relations and, when translating to CNF, used unit propagation and the failed literal rule (flit) to pre-process and simplify the CNF. Though this method did not solve problems whose solutions were longer than eleven steps, it solved all of the problems it could consistently solve very quickly.

On problem instances that all our variations of Blackbox solved, the mean and median times of Blackbox using mutex propagation, unit propagation, and flit were less than 10% those of a Blackbox planner not employing those strategies (see Figures 3 and 4). Not simplifying our CNF with unit propagation seemed to be the biggest culprit in the blow-up from not using all the available heuristics and simplifications; running with flit and unit propagation but without mutex propagation, the fully optimized Blackbox ran only 20 to 50% faster. When propagating mutexes, Blackbox ran, on average, in 20% of the time of a planner not propagating mutexes, but the median running time was actually 1% of the non-simplifying planner. As such, it seems that flit and unit propagation are very important for the Blackbox planner.

	Nothing	**Only Simplify**	**Only Mutex**
Average	0.094454769	0.556619996	0.213274101
Median	0.015437921	0.807692308	0.010044643

Figure 4: Blackbox's running time as a percent of handicapped planners

The reason for this is probably best explained in view of the structure of the CNF generated by mutex relations. As stated above, every mutex adds a clause of the form ($\sim a \vee \sim b$). These can grow as the square of the number of propositions and actions; this explosion is very difficult for WalkSAT to handle, though the failed literal rule and unit propagation are very well suited for this sort of elimination. This is why combining the two—unit propagation and failed literal rule with mutex calculations—proves effective.

Running times on blocks world problems, as stated above, were much higher than the other problems in the test set, at least when all optimizations for Blackbox were running.

	All Optimize	**Nothing**	**Only Simplify**	**Only Mutex**
Blocks World	25.15	65.52	26.6	29.86
Logistics	1.58	162.71	83.69	280.21

Figure 4: Comparing Blocks World with Logistics

In Figure 4, one can see a strange phenomena: when all optimizations are running, logistics planning (moving a truck full of packages, for example) is significantly faster than blocks world, but without mutex propagation or unit-propagation and flit, blocks world planning is faster. The blocks world solution is four steps long, while the logistics solution is 10 steps, but that a ten step solution is quicker than a four step is interesting in its own right. Our first thought was that when mutexes are propagated through the graph, the logistics problem must have many mutex relations that are difficult to solve. This was not borne out, however, because the blocks world problem in question had 90% of its variables assigned by unit propagation and the failed literal rule, while the logistics problem had only 50%. This is an area for further consideration, but it does bring us to the next topic of this paper: the effects of unit propagation and the failed literal rule.

Impact of simplification

With unit propagation, the failed literal rule, and mutex propagation running, the problems we consistently solved had an average of 77% of their variables set through unit propagation and the failed literal rule. When mutex relations are not propagated through propositions as well as actions, the failed literal rule and unit propagation assign, on average, 64% of the variables. This interaction suggests that propagating mutex relations is a valuable technique in Blackbox planning; while mutex relations do add significantly to the number of clauses (sometimes giving upwards of 2 million clauses), they also improve the efficacy of unit propagation and the failed literal rule.

The interactions between the formula simplifications and the time to solve problems are not completely clear. In Figure 5, we compare the times to solve different problems with and without mutex propagation in relation to the percentage of variables that unit propagation and the failed literal rule assign. There is some correlation—the problem big-bw1 (a blocks world problem) without mutexes was not much affected by unit propagation or the failed literals rule, and it is slow, as is att-log, a logistics problem. But mutex propagation affects other blocks world problems (att-bw2, for example) little, and Blackbox is actually slower to solve the logistic problem att-log1a when it assigns more variables through unit propagation and flit.

	With Mutex Propagation		**No Mutex Propagation**	
	% Assigned	**Time (s)**	**% Assigned**	**Time (s)**
att-bw2	0.917142857	25.15	0.91	26.6
att-log	0.502849003	1.58	0.156695157	83.69
att-log1a	0.601265823	0.26	0.389240506	0.19
big-bw1	0.948051948	4.01	0.073688913	333.15
d1s1-8	0.95	0.09	0.822222222	0.11
d1s1-10	0.96	0.21	0.854545455	0.26
d1s1-4	0.9	0	0.68	0.02
ferry3	0.838235294	0.03	0.838235294	0.03
ferry7	0.665289256	0.25	0.115107914	6.43

Figure 5: Comparing Unit Propagation and Failed Literals Efficacy with Timing.

In the end, though, using the failed literal rule and unit propagation, along with mutex propagation, seems to provide the most effective method for solving planning problems.

IX. Discussion and Future Directions

In this report, we have investigated Blackbox, a planning approach that unifies Graphplan and SATPLAN algorithms to more effectively plan in a wide variety of domains. We have demonstrated that using a combination of mutex propagation, unit propagation in SAT problems, the failed literal rule, and WalkSAT provide a good framework for many planning problems.

The planning graph, when converted into a CNF form, has a structure that other SAT solvers, such as satz [6], a Davis-Putnam based procedure relying on unit-propagation heuristics, might take advantage of. This would provide the advantage of completeness as well as quicker solving. Another possibility is suggested by Kautz and Selman: we could compressing the CNF representation by having actions imply their effects as well as preconditions, rather than instantiating mutex clauses, since these would be entailed by any interfering effects. This would shrink our quadratically growing set of clauses, and Horn-clause theorem proving is a well-established area of computing.

Time management is another area that could be interesting to look at. Given that we work with an inherently incomplete algorithm (WalkSAT), deciding the time at which to break off search and try instantiating a new level is a little nebulous. The approaches we took, to statically allocate time blocks, giving more as we moved down the tree, did not seem very effective. Early in the search, WalkSAT exits before its time is up because it finishes all its flips, but later in the search, WalkSAT may run out of time. Even complete solvers may want time limits, because levels later in the planning graph may have more obvious solutions and be easier to find than optimal ones.

A last area for further exploration is making the planning graph more efficient. Realistically, the only information we need about propositions and actions are the first levels at which they become instantiated and the last levels they are mutex with other actions or propositions. This, however, would entail a significant amount of overhead and complexity to shrink a graph that is polynomial anyway, so we do not see too many payoffs from this approach.

Blackbox provides a good way to merge two evolving algorithm classes—graph based planning and SAT solving—into an algorithm that can achieve impressive results on many planning problems.

X. References

[1] Blum, A. and Furst, M. (1997). Fast Planning Through Planning Graph Analysis. Artificial Intelligence 90, 281-300.

[2] Connor, J. and Duchi, J. (2005). An Experimental Analysis of Satisfiability Algorithms. In Procs. of Eugene's Printer.

[3] Kautz, H. and Selman, B. (1999). Unifying SAT-based and Graph-based Planning. In Procs. of IJCAI-99.

[4] Bonet , B. and Geffner, H. (1999). Planning as Heuristic Search: New Results. Procs. of ECP-99, 360-372. (Springer).

[5] Nguyen, X. and Kambhampati K. (2000). Extracting Effective and Admissible State Space Heuristics from the Planning Graph. In Procs. of AAAI-2000.

[6] Li, C. and Anbulagan. (1997). Heuristics Based on Unit Propagation for Satisfiability Problems. In Procs. of IJCAI-97.