1 Introduction

It has often been observed that the classical AI planning problem (that is, planning with complete and certain information) is a form of logical deduction. Because early attempts to use general theorem provers to solve planning problems proved impractical, research became focused on specialized planning algorithms. Sometimes the relationship to inference was explicitly acknowledged: for example, the STRIPS system (Fikes and Nilsson 1971) was originally described as a way to make theorem-proving practical. In other work the relationship to deduction was developed after the fact. For example, Chapman’s (1985) work on TWEAK clarified the logic behind one variety of non-linear planning.

The belief that planning required specialized algorithms was challenged by the work on planning as propositional satisfiability testing of Kautz and Selman (1992, 1996). SATPLAN showed that a general propositional theorem prover could indeed be competitive with some of the best specialized planning systems. The success of SATPLAN can be attributed to two factors:

- The use of a logical representation that has good computational properties. Both the fact that SATPLAN uses propositional logic instead of first-order logic, and the particular conventions suggested for representing time and actions, are significant. Differently declarative representations that are semantically equivalent can still have quite distinct computational profiles.

- The use of powerful new general reasoning algorithms such as Walksat (Selman, Kautz, and Cohen 1994). Many researchers in different areas of computer science are creating faster SAT engines every year. Furthermore, these researchers have settled on common representations that allow algorithms and code to be freely shared and fine-tuned. As a result, at any point in time the best general SAT engines tend to be faster (in terms of raw inferences per second) than the best specialized planning engines. In principle, of course, these same improvements could be applied to the specialized engines; but by the time that is done, there will be a new crop of SAT solvers.

An approach that shares a number of features with with the SATPLAN strategy is the Graphplan system, developed independently by Blum and Furst (1995). Graphplan broke previous records in terms of raw planning speed, and has become a popular planning framework. Comparisons to SATPLAN show that neither approach is strictly superior. For example, SATPLAN is faster on a complex logistics domain, they are comparable on the blocks world, and on several other domains Graphplan is faster. For excellent reviews and discussions of the two systems, see Kambhampati (1997) and Weld (1998).

A practical difference between SATPLAN and Graphplan is that the former takes as input a set of axiom schemas, while the input for the latter is a set of STRIPS-style operators. However, they bear deep similarities. Both systems work in two phases, first creating a propositional structure (in Graphplan, a plan graph, in SATPLAN, a CNF wff) and then performing a search (over assignments to variables) that is constrained by that structure. The propositional structure corresponds to a fixed plan length, and the search reveals whether a plan of that length exists. Kautz and Selman (1996) noted that the plan graph has a direct translation to CNF, and that the form of the resulting formula is very close to the original conventions for SATPLAN. In the unifying framework of Kambhampati (1997), both are examples of disjunctive planners. The initial creation of the propositional structure is a case of plan refinement without splitting, while the search through the structure is a case of plan extraction. We hypothesize that the differences in performance of the two systems can be explained by the fact that Graphplan uses a better algorithm for instantiating (refining) the propositional struc-
ture, while SATPLAN uses more powerful search (extraction) algorithms.

SATPLAN fully instantiates a complete problem instance before passing it to a general logic simplifier (a limited inference algorithm that runs to completion in polynomial time) and a solver (a complete or incomplete model-finding program). By contrast, Graphplan interleaves plan graph instantiation and simplification. The simplification algorithm used in Graphplan is based on mutex computation, an algorithm for determining that pairs of actions or pairs of facts are mutually exclusive. Mutex computation can be viewed as rule of limited inference that is specialized to take particular advantage of the structure of planning problems (Kambhampati et al. 1997). Specifically, mutex computation is a limited application of negative binary propagation:

\[
given: \{ \neg p \lor \neg q \}, \{ p \lor \neg r \}
\]
\[
infer: \{ \neg q \lor \neg r \}
\]

Each application of the rule allows the deduction of a negative binary clause (a mutex). The mutex algorithm used by Graphplan is incomplete (not all mutexes that logically follow can be inferred) and terminates in polynomial time. Note that this algorithm is different from the simplification rule of unit propagation employed by the original SATPLAN: more powerful in propagating negative clauses, but somewhat less powerful in propagating positive information. The set of mutexes is used in two ways by Graphplan, both to prune nodes from the graph during instantiation and to prune branches of the search tree that involve mutually exclusive actions.

These observations have led us to create a new system that combines the best features of Graphplan and SATPLAN. This system, called Blackbox,\(^1\) works in a series of phases:

1. A planning problem (specified in a standard STRIPS notation) is converted to a plan graph of length \(k\), and mutexes are computed as described above;
2. The plan graph is converted to a CNF wff;
3. The wff is simplified by a general CNF simplification algorithm;
4. The wff is solved by any of a variety of fast SAT engines;
5. If a model of the wff is found, then the model is converted to the corresponding plan; otherwise, \(k\) is incremented and the process repeats.

Note that specialized limited inference is used in mutex computation, while general limited inference is used in CNF simplification. We will return to the complementary nature of these two processes in section 3 below. The input to the final general SAT engine can be considered to be the combinatorial core of the problem. The basic translation from a plan graph to SAT is described in Kautz, McAllester, and Selman (1996); in section 2 we will also describe a variation used in some of our experiments. Baioletti et al. (1998) also propose a similar scheme for translating plan graphs augmented with temporally-qualified goals into SAT.

The wff generated from the plan graph can be considerably smaller than one generated by translating STRIPS operators to axioms in the most direct way, as was done by the earlier MEDIC system of Ernst, Millstein, and Weld (1997). Furthermore, the fact that the plan graph’s mutex relationships are directly translated into negative binary clauses makes the formula easier to solve by many kinds of SAT engines.

Blackbox currently includes the local-search SAT solver Walksat, and two systematic SAT solvers, satz (Li and Anbulagan 1997) and relsat (Bayardo and Schrag 1997), in addition to the original Graphplan engine (that searches the plan graph instead of the CNF form). The two systematic solvers are comparable in power although quite different in approach: satz is based on forward-checking, while relsat employs dependency-directed backtracking. In order to have robust coverage over a variety of domains, the system can employ a schedule of different solvers. For example, it can run Graphplan for 30 seconds, then Walksat for 2 minutes, and if still no solution is found, satz for 5 minutes.

The Blackbox system actually introduces new SAT technology as well, namely the use of randomized complete search methods. As shown in Gomes, Selman, and Kautz (1998), systematic solvers in combinatorial domains often exhibit a “heavy tail” behavior, whereby they get “stuck” on particular instances. Adding a small amount of randomization to the search heuristic and rapidly restarting the algorithm after a fixed number of backtracks can dramatically decrease the average solution time. Figure 1 illustrates the effect of adding randomized restarts to a deterministic backtracking search algorithm. We see from the figure that as the cutoff is increased from its lowest setting the mean solution time first rapidly decreases. Then, as the cutoff continues to increase, the mean solution time increases in a near-linear fashion. This increase in expected time is due to the fact that for this problem a non-negligible portion of the runs take arbitrarily long to complete. In this example, at the optimal cutoff value, about 1 out of 100 runs succeeds in finding a solution.

We applied this randomization/restart technique to the version of satz used by Blackbox. The variable-choice heuristic for satz chooses to split on a variable that maximizes a particular function of the number of unit propagations that would

---


---

Figure 1: Relationship of the cutoff value (measured in backtracks until a restart is performed) on expected solution time. Data is for a randomized backtracking algorithm (satz-rand) for the SATPLAN encoding of a logistics planning problem (log.d). The Y-axis specifies the expected number of backtracks performed until a solution is found, counting the previous failed restarts.
be performed if that variable were chosen (see Li and Anbulagan (1997) for details). Our version, satz-rand, randomly selects among the set of variables whose scores are within 40% of the best score. (The value 40% was selected empirically.) The cutoff value is specified in the solver schedule. No one value cutoff value is ideal for all domains. Indeed, predicting good cutoff values is a deep theoretical question (the known asymptotic results (Luby et al. 1993) are often not practical). If you need to solve a number of similar problems, it is feasible to carefully tune the cutoff value on a few of the instances, and then use that value for the rest of the set. However, this cannot be done when a unique problem is encountered, or if you do not specify the parallel plan length in advance and the system must search through a series of different size problems.

A simple and effective practical solution is to provide the solver with a schedule of different cutoff values. The schedule specifies some number of trials with a very low cutoff value, and if those all failed, then so many at a higher value, and so forth until either a solution is found or all trials end in failure. This can be viewed as a version of the well-known search strategy of iterative deepening. Like iterative deepening, the time spent on trials with a low cutoff value is negligible compared to that spent on trials with a higher value, so that even when the early part of the schedule fails to find a solution little overall effort is wasted.

2 Empirical Results

In order to test the effectiveness of the Blackbox approach we selected benchmark problems that have the following characteristics:

- Domains are computationally challenging: there is no simple polynomial time algorithm for optimal planning.
- Solution time is dominated by search: plan graph generation is relatively fast, but solution extraction is hard.
- Problem instances are critically constrained: finding optimal plans is much harder than finding sub-optimal ones.
- Both parallel and sequential planning problems are included.
- Instances push the limits of the approach in order to demonstrate how it scales up.

In this paper we present results on a set of such problems from logistics planning (Velosa 1992), a highly parallel planning domain, and from the classic blocks world, a purely sequential domain. The test machine was an SGI 194 MHz R10000 Challenge server, where each process ran on a dedicated processor. The largest amount of RAM allocated by Blackbox during solution of the largest logistics problem (log.d) was 178 MB, and during the solution of the largest blocks world problem (bw.c) was 660 MB. Most of the memory was allocated during the phase of constructing the initial plan graph.

Table 1 compares the performance of Blackbox (version 3.4), SATPLAN, Graphplan, and IPP (version 3.3) on our logistics benchmark problems, where the optimal plan length is provided as input. Graphplan is the original code developed by Blum and Furst (1995) that also is incorporated in the front-end of Blackbox. IPP (Koehler et al. 1997) is a new implementation of the Graphplan algorithm including many improvements and extensions. We tried the solvers satz, satz-rand, and Walksat for both Blackbox and SATPLAN. Input to Blackbox, Graphplan, and IPP was in PDDL STRIPS format (McDermott 1998): input to SATPLAN was a set of hand-crafted axiom schemas using the “state-based” encodings described in Kautz and Selman (1996). Blackbox simplified wffs before passing them on to a solver using the failed literal rule, while SATPLAN simplified using unit propagation.

Let us first make some general observations. We see that the scaling of Blackbox using any of the SAT solvers is better than Graphplan. It is important to note that up to the point at which the wff is generated from the plan graph, the code running in Blackbox and Graphplan is identical. This indicates that the cost of performing the SAT translation is small compared to the savings gained by using any of the SAT engines instead of the (relatively) simple backward-chaining search performed by Graphplan. Although IPP generally improves upon Graphplan, in this case it only provides faster solution times on the two smallest instances, and is always slower than Blackbox. This is due to the fact that most of the improvements in IPP over Graphplan are not invoked in this test: they only come into play when the plan length is not given, or when the initial state description contains facts that are irrelevant to the solution. This test also does not allow us to take advantage of the “RIFO” heuristic graph pruning option in IPP, because doing so prevents IPP from finding the minimum parallel length solutions at all.

In short: for critically-constrained planning problems where plan extraction is the computational bottleneck for Graphplan-type solvers, translating the plan graph into SAT and applying a general SAT solver can boost performance.

A second general observation is that the scaling of the best solution times for Blackbox (using satz-rand) is close to the scaling of the best solver-only times for SATPLAN (using Walksat). This is quite remarkable, given the fact that the Blackbox encodings were generated automatically, while the SATPLAN axioms were carefully hand-tuned in order to provide the best possible performance for Walksat. The SATPLAN encodings even included explicit state invariants (such as the fact that “a truck is only at one location at a time”) that are known to boost the performance of problem solvers (Kautz and Selman 1998). Even more striking is the fact that when the time to generate the SATPLAN encodings is also taken into account, the overall Blackbox times are consistently better than the SATPLAN times. For example, Blackbox takes 28 seconds to generate and solve log.d, while SATPLAN takes 3.6 minutes (3.5 minutes to generate and 7 seconds to solve).

In short: advances in SAT solvers have made planning using SAT encodings automatically generated from STRIPS operators competitive with planning using hand-crafted SAT encodings.

These results contrast with earlier experiments on solving automatically-generated SAT encodings of planning problems. Kautz and Selman (1996) reported that both Walksat
and ntab (a deterministic backtracking solver, less complex than satz or relsat) had difficulty solving plan graph generated SAT encodings of the larger logistics problems, getting as far as log.b before the running time exploded. The MEDIC system (Ernst, Millstein, and Weld 1997) used the same solvers but generated the SAT encodings directly from the STRIPS axioms without taking advantage of an intermediate plan graph representation, by using the conventions described in Kautz, McAllester, and Selman (1996). They reported a solution time of 1.1 hours using Walksat on log.a. One should note that there is no significant overhead in using a plan graph for generation; in fact, the generation phase in Blackbox takes only a few seconds for each of problems described above, versus several minutes for generation by SATPLAN or MEDIC.

A longer version of this paper will contain a detailed comparison with MEDIC. However, our preliminary experiments indicate that wffs generated from a plan graph (as in Blackbox) have significantly different computational properties from ones generated directly from STRIPS (as in MEDIC), despite the fact that they are logically equivalent (Kautz, McAllester, and Selman 1996). In particular, the plan graph-based wffs contain fewer variables, more clauses, and are easier to solve. For example, the encoding of log.a generated by Blackbox contained 2,709 variables and 27,522 clauses, while the encoding generated by MEDIC (using the regular operator representation with explanatory frame axioms) contained 3,510 variables and 16,168 clauses. As shown above, the Blackbox wff can be solved by satz-rand in 5 seconds, but we have not yet been able to find a setting for the parameters for satz-rand that will let it solve the MEDIC wff in less than 24 hours.

The differences between the two kinds of wffs can be explained by the fact that the plan graph algorithm prunes many unreachable nodes, thus reducing the number of variables in the corresponding encoding, while propagating mutexes between nodes, thus increasing the number of (negative binary) clauses. The added binary clauses increase propagation at each branch of the backtracking search and thus speed the solution time. An interesting open question that we are currently investigating is whether a SAT solver that uses dependency-directed backtracking (e.g., relsat) can actually "learn" the added clauses while running on a MEDIC-type encoding.

### Table 1: Results on critically constrained logistics benchmark planning problems running on a 194 MHz SGI Challenge server

<table>
<thead>
<tr>
<th>problem</th>
<th>parallel time</th>
<th>Blackbox</th>
<th>Graphplan</th>
<th>IPP</th>
<th>SATPLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>walksat</td>
<td>satz</td>
<td>satz-</td>
<td>create</td>
</tr>
<tr>
<td>rocket.a</td>
<td>7</td>
<td>3.2 sec</td>
<td>5 sec</td>
<td>3 sec</td>
<td>3.4 min</td>
</tr>
<tr>
<td>rocket.b</td>
<td>7</td>
<td>2.5 sec</td>
<td>10 sec</td>
<td>5 sec</td>
<td>8.8 min</td>
</tr>
<tr>
<td>log.a</td>
<td>11</td>
<td>7.4 sec</td>
<td>5 sec</td>
<td>5 sec</td>
<td>31.5 min</td>
</tr>
<tr>
<td>log.b</td>
<td>13</td>
<td>1.7 min</td>
<td>7 sec</td>
<td>7 sec</td>
<td>12.7 min</td>
</tr>
<tr>
<td>log.c</td>
<td>13</td>
<td>14.9 min</td>
<td>9 sec</td>
<td>9 sec</td>
<td>—</td>
</tr>
<tr>
<td>log.d</td>
<td>14</td>
<td>—</td>
<td>52 sec</td>
<td>28 sec</td>
<td>3.5 min</td>
</tr>
</tbody>
</table>

Optimal parallel plan length was provided as an input. Blackbox options for column “satz” are “compact -l -then satz”. Blackbox options for column “satz-rand” are: “compact -l -then satz -cutoff 20 -restart 10 -then satz -cutoff 200 -restart 1000000”. SATPLAN solver options for column “satz-rand” are: “satz -cutoff 16 -restart 1000000”. Walksat options for both Blackbox and SATPLAN are: “-best -noise 50 100 -cutoff 100000000”. Timings are real wall-clock times including all input and output; differences less than 1 second between different programs are not significant due to implementation details. Timings for the randomized methods are averages over 1,000 trials. Timings for SATPLAN separate time used to generate wff (create) and time used for each of the solvers. Long dash (—) indicates solution not found after 24 hours.

In short: use of an intermediate plan graph representation appears to improve the quality of automatic SAT encodings of STRIPS problems.

Next, let us consider the differences in performance caused by different SAT solvers for Blackbox and SATPLAN. First, we see that while Walksat performs very well on the smaller Blackbox instances, it does poorly on the two largest, log.c and log.d. By contrast, local search works well for even the largest SATPLAN encodings. (This suggests some specific connection between local search and state-based encodings, a topic that has received relatively little attention since the original SATPLAN work.) The deterministic version of satz shows more consistent behavior across the Blackbox instances, although it stumbles on rocket.b and log.d. Satz stumbles even more dramatically on log.a and log.d for the SATPLAN encodings.

What is happening in each of these “stumbles” is that the satz variable choice heuristic (which is usually very good) has made a wrong choice early on in the search tree, and so the algorithm spends much time exploring a large area of the search space that is devoid of solutions. As discussed in Gomes, Selman, and Kautz (1998), one can observe this occurring for backtrack search for either a deterministic algorithm on a large distribution of problem instances, or for a randomized backtrack algorithm repeatedly solving a single instance. The latter case is the easiest to observe and has formed the basis of most experimental work on the subject, since one can simply do many runs of the algorithm (where the variable choice heuristic randomly breaks “ties”). In the experiments discussed here we have, by contrast, a deterministic algorithm running on small of different instances. In a set of examples this small it is not surprising that the phenomena
Table 2: Results for Blackbox finding optimal solutions to benchmark planning problems where system must search for
the minimum parallel time length. The “Timeout/satz-rand” solver options are “-maxsec 30 graphplan -then satz -cutoff 20
-restart 10 -then satz -cutoff 200 -restart 1”. The “Prove optimal/satz” solver options are “-maxsec 30 graphplan -then satz
-cutoff 20 -restart 10 -then satz”. The “Prove optimal/relsat” solver options are “-maxsec 30 graphplan -then relsat”. Long
dash (—) indicates solution not found after 24 hours. In every
case the same quality solutions were ultimately found.

<table>
<thead>
<tr>
<th>problem</th>
<th>Timeout</th>
<th>Prove optimal</th>
<th>Prove optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/satz-rand</td>
<td>/satz</td>
<td>/relsat</td>
</tr>
<tr>
<td>rocket.a</td>
<td>59 sec</td>
<td>59 sec</td>
<td>57 sec</td>
</tr>
<tr>
<td>rocket.b</td>
<td>1 min</td>
<td>1 min</td>
<td>1 min</td>
</tr>
<tr>
<td>log.a</td>
<td>1.3 min</td>
<td>1.3</td>
<td>1.1</td>
</tr>
<tr>
<td>log.b</td>
<td>2.1 min</td>
<td>45 min</td>
<td>2.1</td>
</tr>
<tr>
<td>log.c</td>
<td>3 min</td>
<td>—</td>
<td>4.9</td>
</tr>
<tr>
<td>log.d</td>
<td>3.7 min</td>
<td>3.7</td>
<td>2.6</td>
</tr>
</tbody>
</table>

In short: Randomized restarts boost the performance of systematic SAT algorithms on encodings of planning
problems.

We next reran the experiments using the randomized/restart version of satz described earlier. The Blackbox sched-
ule for satz-rand used a cutoff of 20 backtracks for up to 10
restarts, followed by a cutoff of 200 backtracks restarting un-
til a solution was found. The SATPLAN schedule for satz-
rand was a cutoff of 16 backtracks restarting until a solution
was found. These schedules were only very roughly tuned by
hand after observing a few trials and are not necessarily op-
timal. However, in each case the observed solution time was
significantly reduced. For Blackbox the times for rocket.b
and log.d were cut in half, while even more significant sav-
ings were seen for SATPLAN, where the solution time for
log.d decreased from 1.8 hours to 1.6 minutes.

In short: Performance of Blackbox for plan length
search tasks can be acceptable, even though information
from failed too-short trials is not reused, as it is in Graph-
plan.

A problem with the SAT approach in some domains is that
the problem encodings become very large. A notable case
is the classic single-armed blocks world. Because no par-
allel actions are permitted, the plan graph must contain as
many layers as there are actions in the solution. If there are
blocks, then there are $O(n^2)$ actions and $O(n^3)$ mutexes
per layer. Thus the translation of a 28-step, 15 block problem
(large.c) contains about 2.5 million clauses, most of which
are negative binary clauses (mutexes).

We therefore developed a modification to the translation
scheme that can reduce the number of clauses in such cases. Note that it is not logically necessary to translate a particular
mutex if that negative binary clause is implied by other parts
of the translation. If particular, if we add clauses that state that
an action implies its effects as well as its preconditions (the
latter are part of the default translation), then mutexes do not
need to be explicitly translated for actions with conflicting ef-
fects or conflicting preconditions: only mutexes where the ef-
eft of an action conflicts with the precondition of another are
needed. Table 3 shows the results of performing this “com-
pressed” translation. The encodings are about 75% smaller
in the number of clauses and considerably easier to solve
by satz-rand (which has no difficulty in chaining through the
Horn clauses that entail the “missing” mutexes). For compar-
ison the final column provides solution times for the Graph-
plan search engine working on the plan graphs that explic-
ity include all the inferred mutex relations. Performance of
Blackbox and Graphplan is comparable, although neither
is currently state of the art. (However, Blackbox’s ability
to find optimal solutions to a 28-step blocks world problems
would have been considered state of the art performance as
little as two years ago.)

In short: SAT encodings become problematically large
in sequential domains with many operators, although refi-
ements to the encoding scheme can delay the onset of the
combinatorial explosion.

In summary, our experiments show that Blackbox pro-
vides an effective approach for “search bound” problem in-
ces in certain highly parallel planning domains such as
logistics planning. The approach runs into difficulties in do-

mains such as the blocks world where both the intermediate
plan graph and the SAT translation become very large, al-
though the technique of compressed encodings provide sig-
nificant relief.

A longer version of this paper (in preparation) will include
results on an expanded set of benchmark problems, including
the instances from the AIPS-98 Planning Competition. Al-
though the performance of Blackbox in the AIPS competi-

only occurred for 4 of the 12 trials.

Table 2 shows the results of running Blackbox and the
same logistics instances where the parallel solution length is
not specified in advance. The times for running Graphplan
or IPP in this mode on these instances are only marginally
higher than when the plan length is given as input: most of the
work the Graphplan-type engines perform occurs when the
plan length reaches the optimal bound. We ran Blackbox
with satz-rand in two modes: in the first “timeout” mode, if a
solution is not found after a few restarts (10 restarts at cutoff
20, 1 restart at cutoff 200), the plan length is incremented.
In the second mode, Blackbox is made complete by mak-
ing the final part of the solver schedule run satz without any
cutoff. Thus, only the second mode actually proves optimality.
By comparing two modes, we see that the first (time-
out) is much faster than the second, even though the same
quality solutions are ultimately found. This is because in the
second mode most of Blackbox’s effort goes into proving
the non-existence of a solution of length one step less than
optimal. Or, in other words, the “co-NP” part of the SAT
translation was empirically harder than the “NP” part for satz-
rand. Finally, the table presents some data from our initial ex-
periments using the dependency-directed backtracking SAT
solver relsat. This is also a complete method that guarantees
optimality, but now we see that it’s timings are comparable
with using satz-rand in its incomplete mode. (When the plan
length is given as input, our preliminary experiments indicate
that satz-rand usually has a edge over relsat.)
Table 3: Comparing the default and “compressed” SAT translations produced by blackbox, for blocks world problems where the optimal plan length is input (no parallel actions possible). Solver used by Blackbox is “-compact -l -then satz -cutoff 40 -restart 20 -then satz -cutoff 400”. Star (*) indicates solver failed due to memory size, and long dash (—) that no solution found after 48 hours.

<table>
<thead>
<tr>
<th>problem</th>
<th>steps</th>
<th>Blackbox</th>
<th>Compress</th>
<th>Graphplan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>default</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>clauses</td>
<td>time</td>
<td>clauses</td>
</tr>
<tr>
<td>reverse</td>
<td>8</td>
<td>1,347</td>
<td>2 sec</td>
<td>917</td>
</tr>
<tr>
<td>12step</td>
<td>12</td>
<td>25,978</td>
<td>5 sec</td>
<td>6,643</td>
</tr>
<tr>
<td>large.a</td>
<td>12</td>
<td>116,353</td>
<td>13 sec</td>
<td>18,061</td>
</tr>
<tr>
<td>large.b</td>
<td>18</td>
<td>469,993</td>
<td>6.5 min</td>
<td>123,653</td>
</tr>
<tr>
<td>large.c</td>
<td>28</td>
<td>2,496,832</td>
<td>*</td>
<td>917,402</td>
</tr>
</tbody>
</table>

Table 4: The number of variables in the encoding of a series of planning problems before simplification, and the percentage determined by simplification by unit propagation (uprop), the failed literal rule (flit), and by the binary failed literal rule (blit). The first set of problems are blocks world and the second set are logistics.

<table>
<thead>
<tr>
<th>problem</th>
<th>vars</th>
<th>uprop</th>
<th>flit</th>
<th>blit</th>
</tr>
</thead>
<tbody>
<tr>
<td>12step</td>
<td>1191</td>
<td>13%</td>
<td>43%</td>
<td>79%</td>
</tr>
<tr>
<td>bw.a</td>
<td>2452</td>
<td>10%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>bw.b</td>
<td>6358</td>
<td>5%</td>
<td>43%</td>
<td>99%</td>
</tr>
<tr>
<td>bw.c</td>
<td>19158</td>
<td>2%</td>
<td>33%</td>
<td>99%</td>
</tr>
<tr>
<td>rocket.a</td>
<td>1337</td>
<td>3%</td>
<td>24%</td>
<td>40%</td>
</tr>
<tr>
<td>rocket.b</td>
<td>1413</td>
<td>3%</td>
<td>21%</td>
<td>49%</td>
</tr>
<tr>
<td>log.a</td>
<td>2709</td>
<td>2%</td>
<td>36%</td>
<td>45%</td>
</tr>
<tr>
<td>log.b</td>
<td>3287</td>
<td>2%</td>
<td>24%</td>
<td>30%</td>
</tr>
<tr>
<td>log.c</td>
<td>4197</td>
<td>2%</td>
<td>23%</td>
<td>27%</td>
</tr>
<tr>
<td>log.d</td>
<td>6151</td>
<td>1%</td>
<td>25%</td>
<td>33%</td>
</tr>
</tbody>
</table>

3 The Role of Limited Inference

The plan graph approach to STRIPS planning gains much of its power through its use of mutex computations, as we briefly described above. During construction of the plan graph, Graphplan marks a pair of instantiated actions as mutually exclusive if one deletes a precondition or add-effect of the other. It further determines that a pair of facts (predicates fully instantiated at a particular time-instant) are mutually exclusive if all the actions that add one are exclusive of all actions that add the other. Additional mutexes are added between actions if a precondition of one is mutex with a precondition of the other. If one takes the number of preconditions or effects of an operator to be constant then mutex computation can be performed in \(O(n^2)\) time, where \(n\) is the number of instantiated actions (where an instantiated action specifies all its parameters as well as a particular time step).

Thus mutex computation is simply a specialized form of constraint propagation, i.e., limited deduction. Some nodes can be determined to be inconsistent during instantiation and immediately eliminated from the plan graph. The remaining mutex relations are used to constrain the search over the either the graph or its SAT translation. It is natural to wonder if other forms of limited inference are useful for planning problems. Blum (personal communication) observes that computing higher-order mutexes (between triples of actions, etc.) is not very useful. Do the binary mutex computations extract all important “local” information from the problem instances?
We decided to test this hypothesis by experimenting with a series of different limited inference algorithms that work on the the SAT encodings of the problems. We used the program “compact” developed by James Crawford, and considered the following options:

**unit propagation** Apply unit resolution. Requires $O(n)$ time.

**failed literal** For each literal, try adding the literal to the formula and applying unit resolution. If inconsistency is determined then then literal can be set to false. Requires $O(n^2)$ time.

**binary failed literal** For each pair of literals, try adding the pair of literals to the formula and applying unit resolution. If inconsistency is determined then the binary clause consisting of the negations of the literals can be added to the formula, and the single failed literal rule applied again. Requires $O(n^3)$ time.

Table 4 shows the result of applying each of these kinds of simplifications to a series of encodings of blocks world and logistics planning problems. For each problem we show the number of variables in the instance and the percentage of those variables whose values are determined by local computation. The results for unit propagation (uprop) seem to confirm the intuition that there is little local information left in these problems. For the blocks world problems only between 2% and 13% of the variables are determined by unit propagation, and for the logistics problems no more than 3% are determined. However, the story changes dramatically for the failed literal rule (flit). In the blocks world from 33% to 100% (i.e., the problem is completely solved!) of the variables are determined. In the logistics domain over 21% of the variables are eliminated. The binary failed literal rule (blit) is even more powerful. All of the blocks world problems were either solved completely or made trivial to solve (less than 131 variables) by this rule. The logistics problems were also further reduced in size, although they remained non-trivial to solve.

These results led us to select the failed literal rule as the default simplification procedure for Blackbox. It runs quickly and greatly decreases the size and hardness of the problem instance. So far the higher overhead ($O(n^3)$ versus $O(n^2)$) for the binary failed literal rule makes it impractical for the domains we have considered: it takes about as long to simplify the problem with the binary rule as to solve it using the unary simplifier and satz-rand. Still, these results suggest that an improved implementation of the binary rule could be of dramatic help in certain domains.

Thus we see that general limited inference computations on the SAT-encoding of planning problems provide a powerful complement to the kind of specialized mutex computations performed by the Graphplan front-end to Blackbox. There is a role both for planning-specific and domain-independent simplification procedures. In future work we plan to see if we can find other polytime simplification algorithms for SAT that take particular advantage of the structure of SAT encodings of planning problems.

### 4 Conclusions

We have provided an overview of the Blackbox planning system, and described how it unifies the plan graph approach to STRIPS planning with the planning as satisfiability framework. It provides a concrete step toward the IJCAI Challenge for unifying planning frameworks (Kambhampati 1997). We discussed empirical results that suggest that new randomized systematic SAT algorithms are particularly suited to solving SAT encodings of planning problems. Finally, we examined the role of limited inference algorithms in the creation and solution of problem encodings.

There is strong evidence that the best current general SAT engines are more powerful than the search (plan extraction) engines used by Graphplan and its descendents. Although it is possible to incorporate the heuristics used by these general solvers back into a specialized planner (see Rintanen (1998) for such an approach), given the rapid development of new SAT engines such a tactic may be premature. As an alternative, Giunchiglia et al. (1998) present evidence that it possible to dramatically boost the performance of a general SAT engine by feeding it a tiny amount of information about the structure of the encoding (in particular, identification of the action variables). There is also much work on improving the plan graph generation phase (e.g., Kambhampati et al. (1997), Nebel et al. (1997)) which could be directly incorporated in Blackbox by replacing its front-end.

Blackbox is an evolving system. Our general goal is to unify many different threads of research in planning and inference by using propositional satisfiability as a common foundation. An important direction that we have not touched on in this paper is the use of domain specific control knowledge in planning (Bacchus and Kabanza 1996; Kautz and Selman 1998; Gerevini and Schubert 1998; Fox and Long 1998); see Cheng, Selman and Kautz (1999) for work on adding control knowledge to Blackbox.

### References


Nagoya, Japan.


