Recent Advances in AI Planning

Daniel S. Weld
Department of Computer Science & Engineering
University of Washington, Box 352350
Seattle, WA 98195–2350 USA

Technical Report `UW-CSE-98-10-01`; to appear in *AI Magazine*, 1999

October 8, 1998

Abstract

The past five years have seen dramatic advances in planning algorithms, with an emphasis on propositional methods such as Graphplan and compilers that convert planning problems into propositional CNF formulae for solution via systematic or stochastic SAT methods. Related work on the *Deep Space One* spacecraft control algorithms advances our understanding of interleaved planning and execution. In this survey, we explain the latest techniques and suggest areas for future research.

Contents

1 Introduction

The field of AI planning seeks to build control algorithms that enable an agent to synthesize a course of action that will achieve its goals. Although researchers have studied planning since the early days of AI, recent developments have revolutionized the field. Two approaches, in particular have attracted much attention:

- The two phase Graphplan [7] planning algorithm, and
- Methods for compiling planning problems into propositional formulae for solution using the latest, speedy systematic and stochastic SAT algorithms.

These approaches have much in common and both are impacted by recent progress in constraint satisfaction and search technology. The current level of performance is quite impressive with several planners quickly solving problems which are orders of magnitude harder than the testpieces of only two years ago. As a single, representative example, the BLACKBOX planner [55] requires only six minutes to find a 105-action logistics plan in a world with 10^{16} possible states.

Furthermore, work on propositional planning is closely related to the algorithms used in the autonomous controller for NASA's *Deep Space One* spacecraft, scheduled to be launched in late 1998. As a result, our understanding of interleaved planning and execution has advanced as well as the speed with which we can solve classical planning problems.

The goal of this survey is to explain these recent advances and suggest new directions for research. Since this paper requires minimal AI background (*e.g.*, simple logic and basic search algorithms), it's suitable for a wide audience. We progress as follows. The remainder of the introduction defines the planning problem and surveys freely-downloadable planner implementations. The next sections discuss graphplan, SAT compilation, and interleaved planning and execution. We conclude by quickly mentioning other recent advances and suggestion topics for future research.

1.1 Preliminaries

A simple formulation of the planning problem defines three inputs:

1. a description of *initial state* of the world in some formal language,
2. a description of the agent's *goal* (*i.e.*, what behavior is desired) in some formal language, and
3. a description of the possible actions that can be performed (again, in some formal language). This last description is often called a *domain theory*.

The planner's output is a sequence of actions which, when executed in any world satisfying the initial state description, will achieve the goal. Note that this formulation of the planning problem is quite abstract — in fact, it really specifies a *class* of planning problems parameterized by the languages used to represent the world, goals, and actions. For example, one might use propositional logic to describe the effects of actions, but this would make it quite awkward to describe actions with universally quantified effects, such as a machine shop spray paint action which coats *all* objects in the hopper. Thus one might describe the effects of actions with first order predicate calculus, but this still assumes that all effects are deterministic. In general, there is a spectrum of more and more expressive languages for representing the world, an agent's goals, and possible actions. In this paper we start by explaining algorithms for planning with the "STRIPS representation."[1] The STRIPS representation describes the initial state of world with a complete set of ground literals. The STRIPS representation is restricted to *goals of attainment*, and these goals are defined as a propositional conjunction; all world states satisfying the goal formula are considered equally good. A domain theory (*i.e.* a formal description of the actions that are available to the agent) completes a planning problem. In the STRIPS representation, each action is described with a conjunctive *precondition* and conjunctive *effect* that define a transition function from worlds to worlds. The action can be executed in any world w satisfying the precondition formula. The result of executing an action in world w is described by taking w's state description and adding each literal from the action's effect conjunction in turn, eliminating contradictory literals along the way.

This defines the so called "classical" planning problem which makes many simplifying assumptions: atomic time, no exogenous events, deterministic action effects, omniscience on the part of the agent, *etc.*. We relax some of these assumptions later in the paper.

1.2 Available Implementations

Many readers will find it helpful to experiment with implementations of the ideas discussed in this paper. Fortunately, there are a variety of freely distributed alternatives, and most accept domains expressed in PDDL[2] syntax, the language used for the AIPS planning competition[3] which we expect will be widely adopted as a standard for teaching purposes and collaborative domain interchange for performance comparison.

- Graphplan and its descendants:

[1]The acronym STRIPS stands for "STanford Research Institute Problem Solver" a very famous and influential planner built in the 1970s to control an unstable mobile robot affectionately known as "Shakey" [28].

[2]See `ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz` for the PDDL specification.

[3]See `ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html` for competition results.

- Graphplan — the original, somewhat dated, C implementation [6] is still available from `www.cs.cmu.edu/afs/cs.cmu.edu/user/avrim/www/graphplan.html`.
- IPP [64] is a highly optimized C implementation of Graphplan, extended to handle expressive actions (*e.g.*, universal quantification and conditional effects); download from `www.informatik.uni-freiburg.de/~koehler`.
- STAN is another highly-optimized C implementation which uses an in-place graph representation and performs sophisticated type analysis to compute invariants. Download from `www.dur.ac.uk/~dcs0www/research/stanstuff/stanpage.html`.
- SGP [109] is a simple, pedagogical Lisp implementation of Graphplan, extended to handle universal quantification, conditional effects, and uncertainty; see `www.cs.washington.edu/research/projects/ai/www/sgp.html`.

- Systems based on compilation to SAT:
  - The highest performance SAT compiler is Blackbox [55] available `www.research.att.com/~kautz/blackbox/index.html`.
  - The Medic planner [22] is a flexible testbed, implemented in Lisp, allowing direct comparison of over a dozen different SAT encodings see `ftp://ftp.cs.washington.edu/pub/ai/medic.tar.gz`.

2 Graphplan & Descendants

Blum and Furst's Graphplan algorithm [6, 7] is one of the most exciting recent developments in AI planning for two reasons:

- Graphplan is a simple, elegant algorithm that yields an extremely speedy planner — in many cases orders of magnitude faster than previous systems such as SNLP [78], Prodigy [82], or UCPOP [88].
- The representations used by Graphplan form the basis of the most successful encodings of planning problems into propositional SAT; hence familiarity with Graphplan aids in understanding SAT-based plannning systems (section 3).

Graphplan alternates between two phases: *graph expansion* and *solution extraction*. The graph expansion phase extends a *planning graph* forward in "time" until it has achieved a necessary (but insufficient) condition for plan existence. The solution extraction phase then performs a backward-chaining search on the graph, looking for a plan that solves the problem; if no solution is found, the cycle repeats by further expanding the planning graph.

We start our discussion by considering the initial formulation of Graphplan, thus restricting our attention to STRIPS planning problems in a deterministic, fully-specified world. In other words, both the preconditions and effects of actions are conjunctions of literals (*i.e.*, positive literals denoting entries in the add lists and negative literals correspond to elements of the delete list). After covering the basics, we describe optimizations and explain how to handle more expressive action languages.

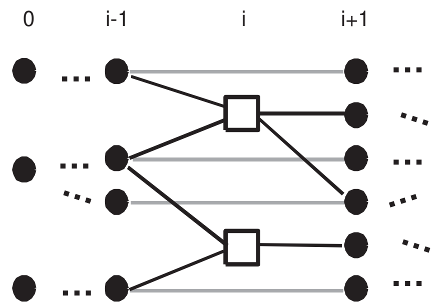

Figure 1: The planning graph alternates proposition (circle) and action (square) layers. Horizontal grey lines between proposition layers represent "maintenance actions," which encode the possibility that unaffected propositions will persist until the next layer.

2.1 Expanding the Planning Graph

The planning graph contains two types of nodes, proposition nodes and action nodes, arranged into levels. Even-numbered levels contain proposition nodes (*i.e.*, ground literals), and the zeroth level consists precisely of the propositions that are true in the initial state of the planning problem. Nodes in odd-numbered levels correspond to action instances; there is one such node for each action instance whose preconditions are present (and are mutually consistent) at the previous level. Edges connect proposition nodes to the action instances (at the next level) whose preconditions mention those propositions, and additional edges connect from action nodes to subsequent propositions made true by the action's effects.

Note that the planning graph represents "parallel" actions at each action level. This means that a planning graph with k action levels can represent a plan with more than k actions. However, just because two actions are included in the planning graph at some level, doesn't mean that it is possible to execute both at once. Central to Graphplan's efficiency is inference regarding a binary mutual exclusion relation ("mutex") between nodes at the same level. We define this relation recursively as follows (see also figure 2):

- Two action instances at level i are mutex if either
  - *Inconsistent Effects:* the effect of one action is the negation of another action's effect, or
  - *Interference:* one action deletes the precondition of another, or
  - *Competing Needs:* the actions have preconditions that are mutually exclusive at level $i-1$.

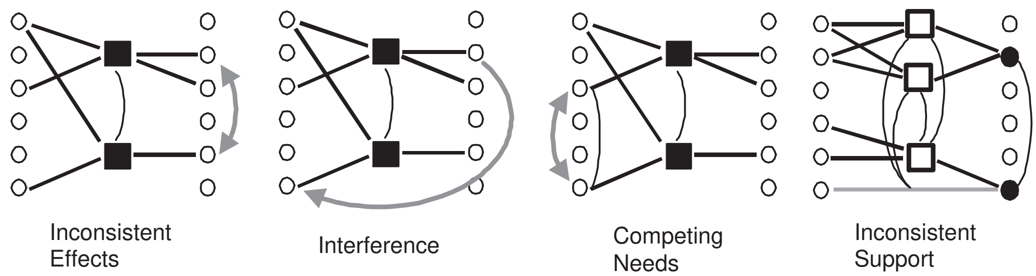

Figure 2: Graphical depiction of the mutex definition (devised by David Smith). Circles denote propositions, squares represent actions, and thin, curved lines denote mutex relations. The first three parts illustrate deduction of a new action-action mutex (between the dark boxes), and rightmost part depicts the discovery of a new mutex between propositions (the dark circles).

```
Initial Conditions:  (and (garbage) (cleanHands) (quiet))
Goal:  (and (dinner) (present) (not (garbage)))
Actions:
   cook   :precondition (cleanHands)
          :effect (dinner)
   wrap   :precondition (quiet)
          :effect (present))
   carry  :precondition
          :effect      (and (not (garbage)) (not (cleanHands)))
   dolly  :precondition
          :effect (and (not (garbage)) (not (quiet)))
```

Figure 3: STRIPS specification of the dinner-date problem.

- Two propositions at level i are mutex if one is the negation of the other, or if all ways of achieving the propositions (*i.e.*, actions at level $i-1$) are pairwise mutex (*Inconsistent Support*).

For example, consider the problem of preparing a surprise date for one's sleeping sweetheart (figure 3). The goal is to take out the `garbage`, fix `dinner`, and wrap a `present`. There are four possible actions: `cook`, `wrap`, `carry`, and `dolly`. `Cook` requires `cleanHands` and achieves `dinner`. `Wrap` has precondition `quiet` (since the gift is a surprise, one mustn't wake the recipient) and produces `present`. `Carry` eliminates the `garbage`, but the intimate contact with a smelly container negates `cleanHands`. The final action, `dolly`, also eliminates the `garbage`, but because of the noisy handtruck it negates `quiet`. Initially, you have `cleanHands` while the house has `garbage` and is `quiet`; all other propositions are false.

Figure 4 shows the planning graph for the dinner date problem expanded

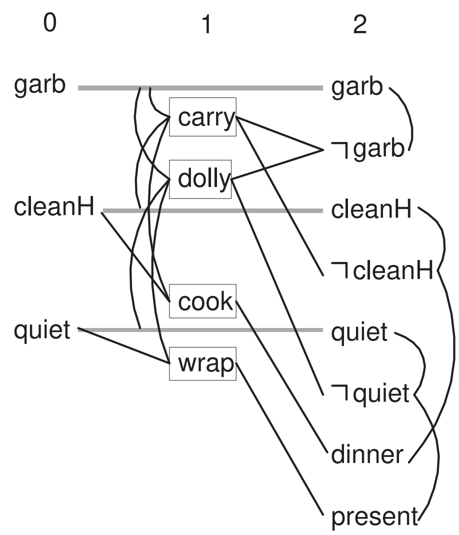

Figure 4: Planning graph for the dinner date problem, expanded out to level two. Action names are surrounded by boxes, and horizontal grey lines between proposition layers represent maintenance actions that encode persistence. Thin, curved lines between actions and propositions at a single level denote mutex relations.

from level zero through one action and proposition level. Note that the `carry` action is mutex with the persistence of `garbage` because they have inconsistent effects. `Dolly` is mutex with `wrap` because of interference, since `dolly` deletes `quiet`. At proposition level two, ¬`quiet` is mutex with `present` because of inconsistent support. Recall that the goal of the dinner date problem is to achieve ¬`garbage` ∧ `dinner` ∧ `present`. Since all of these literals are present at proposition level two, and since none of them are mutex with each other, there is a chance that a plan exists. In this case, the second phase of Graphplan is executed: solution extraction.

2.2 Solution Extraction

Suppose that Graphplan is trying to generate a plan for a goal with n "subgoal" conjuncts, and (as in our example) it has extended the planning graph to an even level, i, in which all goal propositions are present and none are pairwise mutex. This is a necessary (but insufficient) condition for plan existence, so Graphplan performs solution extraction — a backward chaining search to see if a plan exists in the current planning graph.

Solution extraction searches for a plan by considering each of the n subgoals in turn. For each such literal at level i, Graphplan chooses an action a at level

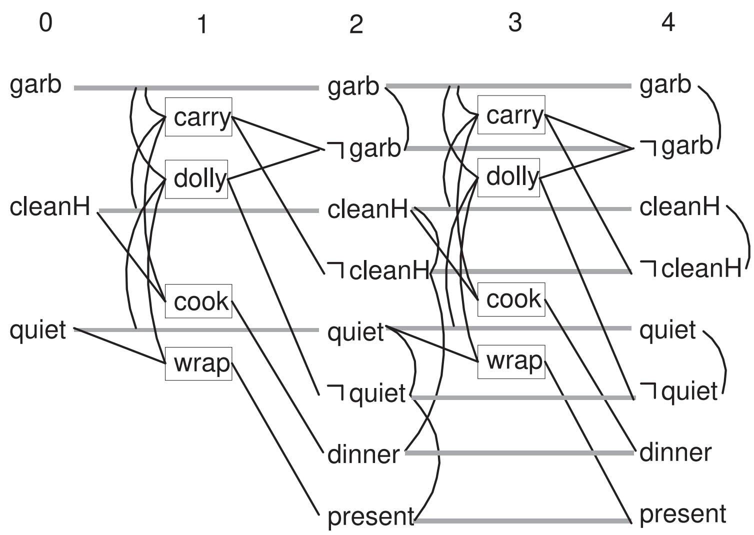

Figure 5: Planning graph for the dinner date problem, expanded out to level four. Although no new literals are present at this proposition level, both `dinner` and `present` have additional support from persistence actions and as a result Graphplan's solution extraction search can find a plan.

$i-1$ that achieves the subgoal. This choice is a backtrack point: if more than one action produces a given subgoal, then Graphplan must consider all of them in order to ensure completeness. If a is consistent (i.e., nonmutex) with all actions that have been chosen so far at this level, then Graphplan proceeds to the next subgoal, otherwise if no such choice is available Graphplan backtracks to a previous choice.

After Graphplan has found a consistent set of actions at level $i-1$ it recursively tries to find a plan for the set formed by taking the union of all of the preconditions of those actions at level $i-2$. The base case for the recursion is level zero — if the propositions are present there, then Graphplan has found a solution. Otherwise, if backtracking fails on all combinations of the possible supporting actions for each subgoal (at each level), then Graphplan extends the planning graph with additional action and proposition levels and then tries solution extraction again.

In the dinner date example, there are three subgoals at level two. ¬`garbage` is supported by `carry` and by `dolly`; `dinner` is supported by `cook`, and `present` is supported by `wrap`. Thus Graphplan must consider two sets of actions at level one: {`carry`, `cook`, `wrap`} and {`dolly`, `cook`, `wrap`}, but unfortunately neither of these sets is consistent because `carry` is mutex with `cook` while `dolly` is mutex with `wrap`. Thus solution extraction fails, and Graphplan extends the planning graph to level four as shown in figure 5.

Note the difference between levels two and four of the planning graph. Although there are no new literals present at level four, there are fewer mutex relations. For example, there is no mutex between `dinner` and `cleanHands` at level four. The most important difference is at level three — where there are five additional maintenance actions encoding the possible persistence of literals achieved by level two. This means that each of the subgoals have additional supporting actions for consideration during the backward chaining process of solution extraction. Specifically,

- ¬`Garbage` is supported by `carry`, `dolly`, and a maintenance action.
- `Dinner` is supported by `cook` and a maintenance action.
- `Present` is supported by `wrap`, and a maintenance action.

so solution extraction needs to consider $3 \times 2 \times 2 = 12$ combinations of supporting actions at level three instead of the $2 \times 1 \times 1 = 2$ combinations during the previous attempt at solution extraction. And, indeed, this increased flexibility allows solution extraction to find a plan. There are actually several combinations which work; we illustrate one below.

Support ¬`garbage` with `carry`, support `dinner` with the maintenance action, and support `present` with `wrap`. None of these actions is mutex with another, so the choices for level three are consistent. The selection of these actions lead to the following subgoals for level two: `dinner` (precondition of the maintenance action) and `quiet` (precondition of `wrap`); since `carry` has no preconditions, there are only two level-two subgoals. Solution extraction recurses, and chooses `cook` to support `dinner` and the maintenance action to support `quiet`; these two actions aren't mutex so the selections for level one are consistent. The preconditions of these actions create two subgoals for level zero: `cleanHands` and `quiet`. Since these propositions are present in the initial conditions, the selection is consistent and a solution plan exists!

Figure 6 illustrates the results of solution extraction. Note that Graphplan generates an inherently parallel (partially ordered) plan. The actions selected for level three, `carry` and `wrap`, can be executed in either order and will achieve the same effect. Thus, if one wishes a totally ordered sequence of actions for one's plan, one may choose arbitrarily: `cook`; `carry`; `wrap`.

2.3 Optimizations

So far we have covered the basic Graphplan algorithm, but there are several optimizations that have a huge effect on efficiency. The first improvements speed solution extraction: forward checking, memoization, and explanation-based learning. The second set of optimizations concern the graph expansion process: handling the closed world assumption, compilation of action schemata to remove static fluents via type analysis, regression focussing, and in-place graph expansion.

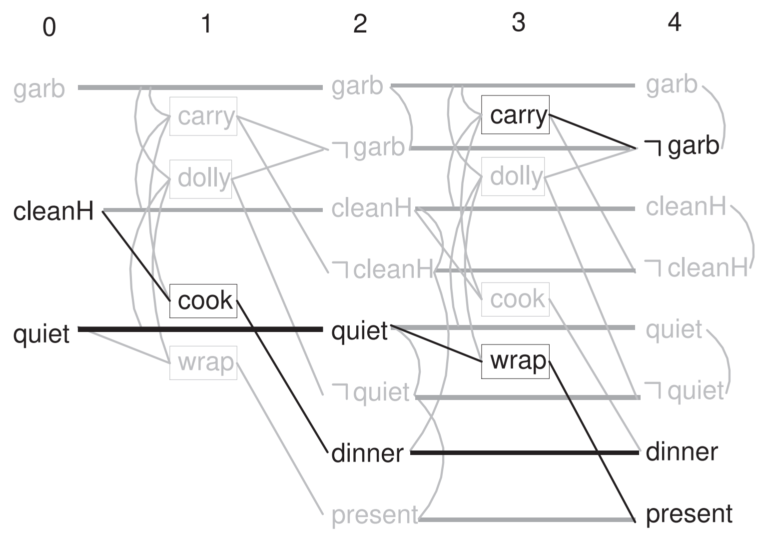

Figure 6: One of four plans that might be found by solution extraction. Actions in black are to be executed; all others are not.

The benefit achieved by each of these optimizations depends on the specific planning problem to be solved. In the worst case, planning graph expansion is polynomial time while solution extraction is exponential [6]. However, in many planning problems it is expansion time which dominates, so each of the optimizations described below is important.

2.3.1 Solution Extraction as Constraint Satisfaction

By observing the connection between the Graphplan solution extraction process and constraint satisfaction problems, we can transfer many insights from the CSP field to planning.[4] There are many possible formulations, but the simplest is in terms of a *dynamic* CSP [27], *i.e.* a constraint satisfaction problem in which the set of variables and associated constraints changes based on the selection of values to earlier variables. There is a CSP *variable* for subgoal literals at each proposition level after level zero. The *domain* of a variable (*i.e.*, its set of possible *values*) is the set of supporting actions at the previous level. The

[4] Although there is a long history of research applying ideas from constraint satisfaction to planning, we focus on applications to Graphplan in this paper (although compilation of planning to SAT can be viewed as taking the constraint satisfaction perspective to its logical conclusion). See MOLGEN [102] for seminal work on constraint-posting planning. TWEAK [13], SNLP [78], and UCPOP [88] manipulated explicit codesignation and ordering constraints. [44] describes a planner that represented all of its decisions as constraints. [48] provides a formal framework of planning that compares different planners in terms of the way they handle constraints. GEMPLAN [70] is a modern constraint-posting planner.

set of constraints are defined by the mutex relations. For example, consider the process of solution extraction from the level four dinner date graph shown in figure 5. Initially, we create a CSP variable for each subgoal at level four: $V_{4,\neg\text{garbage}}$ takes a value from {`carry, dolly, maintain`}, $V_{4,\text{dinner}}$ takes a value from {`cook, maintain`}, and $V_{4,\text{present}}$ from {`wrap, maintain`}. The assignments

$$V_{4,\neg\text{garbage}} = \texttt{carry}$$

$$V_{4,\text{dinner}} = \texttt{maintain}$$

$$V_{4,\text{present}} = \texttt{wrap}$$

corresponds to the first part of the solutiom shown in 6. Once a solution is found for the variables at proposition level four, the actions corresponding to the variable values define a CSP problem at level two. Note that there is no requirement to perform this search level by level. In other words, our previous description of solution extraction dictated finding a consistent set of actions at level i before performing any search at level $i-2$. However, this methodical order is unnecessary and potentially inefficient. For example, the BLACKBOX planner [55] takes the planning graph, compiles it to SAT, and uses fast stochastic methods to perform the equivalent of solution extraction in which search jumps around from level to level in a greedy fashion. Rintanen [91] describes an opportunistic, non-directional search strategy which bypasses conversion to SAT.

By itself, this CSP formulation of solution extraction is unremarkable, but it suggests certain strategies for speeding the search, such as *forward checking*, *dynamic variable ordering*, *memoization*, and *conflict-directed backjumping*.

- When assigning a value to a variable, simple CSP solvers check to ensure that this choice is consistent with all values previously chosen. A better strategy, called *forward checking* [41], checks *unassigned* variables in addition, shrinking their domain by eliminating any values that are inconsistent with the recent choice. If the domain of any unassigned variable collapses (*i.e.*, it shrinks to the empty set), then the CSP solver should backtrack. [66] shows analytically that forward checking is an excellent strategy, strengthening previous empirical support.

- *Dynamic variable ordering* refers to a class of heuristics for choosing which CSP variable should next be assigned a value [3]. Of course, eventually *all* variables must have values assigned but the order in which they are selected can have a huge impact on efficiency [67, 4]. Note that if a variable has only one choice, then it is clearly best to make that assignment immediately. In general, a good heuristic is to select the variable with the fewest remaining (nonconflicting) values and this information is readily available if forward checking is employed.[5] While not astounding, these

[5] Similar heuristics have been investigated in the context of causal-link planners; see [97, 43, 113, 101, 33, 90].

techniques lead to significant (*e.g.*, 50%) performance improvements in Graphplan [49].

Another method for determining a good subgoal ordering is through structural analysis of subgoal interactions [42, 14, 96]. Precomputation aimed at calculating speedy subgoal orderings is closely related to the use of abstraction in planning [59, 103, 114]. In general, one can distinguish between *domain-specific* approaches (which are based on action definitions alone) and *problem-specific* approaches (which additionally use the goal and initial state specifications); problem-specific approaches typically provide more leverage, but the cost of domain-specific precomputation can be amortized amongst many planning problems. Koehler [63] describes a problem-specific method which speeds Graphplan by orders of magnitude on problems from many domains.[6]

- The original Graphplan paper [6] describes a technique called *memoization* which caches for future use the results learned from exhaustive search about inconsistent subgoal sets. Suppose that solution extraction is called at level i and in the course of search attempts to achieve subgoals P, Q, R, and S at level k (where $k \leq i$). If none of the combinations of supporting actions for these subgoals proves consistent (regardless of the level at which this inconsistency is detected), then Graphplan records the set $\{P, Q, R, S\}$ to be a *nogood at level* k. Later, if Graphplan extends the planning graph to level $i+2$ and once again attempts solution extraction, it might attempt to achieve the same four goals at level k but this time it will backtrack immediately rather than performing exhaustive search. The memoization process trades space for time, and although the space requirements can be large, the resulting speedups are significant.

  As stated, this memoization process is rather simplistic, and since more sophisticated approaches have proven effective in systematic SAT solvers [5], one might suspect memoization improvements are possible. Indeed, recent work by Kambhampati [49] demonstrates dramatic speedups (*e.g.*, between 1.6 and 120 times faster depending on domain). The basic idea is to determine which *subset* of goals is responsible for failure at a given level and record only that subset; if solution extraction ever returns to the level with that set *or a superset* then failure is justified. This approach leads to much smaller (and hence more general) nogoods; for example, it might be the case that subgoals P, Q, and S are together unachievable, regardless of R.

  An additional idea (also described in [49]) is the *regression* of level k failure explanations through the action definitions for level $k+1$ to calculate failure conditions for level $k+2$. When these level-$k+2$ conditions are short, then many searches can be terminated very quickly. These methods are

[6] See [25, 97, 98] for additional uses of precomputation based on analysis of action interactions

```
(defschema (drive)
   :parameters   (?v ?s ?d)
   :precondition (and (vehicle ?v)
                      (location ?s)
                      (location ?d)
                      (road-connected ?s ?d)
                      (at ?v ?s))
   :effect       (and (not (at ?v ?s))
                      (at ?v ?d)))
```

Figure 7: Parameterized specification of the action of driving a vehicle from a source location to a destination.

> based on Kambhampati's earlier work on the relationship between traditional planning-based speedup methods (*e.g.*, explanation-based learning) and CSP methods [50].

2.3.2 Closed World Assumption

The closed world assumption says that any proposition not explicitly known to be true in the initial state can be presumed false. A simple way of implementing the closed world assumption in Graphplan would be to explicitly *close* the zeroth level of the planning graph — i.e. to add negative literals for all possible propositions that were not explicitly stated to be true. Because there are an infinite number of *possible* propositions, one should restrict this approach to the *relevant* subset — i.e. those that were mentioned in the preconditions of some action or in the goal; other literals can't affect any solution.

A better solution is to handle the closed world assumption *lazily*, since this shrinks the size of the planning graph and diminishes cost of graph expansion. Create the zeroth level of the planning graph by adding only the propositions known to be true in the initial state (as shown in figure 4). When expanding the planning graph to action level i one does the following. Suppose action A requires $\neg P$ as precondition. If $\neg P$ is in the planning graph at level $i-1$ simply link to it as usual. However, if $\neg P$ is missing from level $i-1$ one must check to see if its negation (i.e. proposition P) is present at level zero. If P is absent from level zero, then add $\neg P$ to level zero and add maintenance actions and mutexes to carry $\neg P$ up to the current level. With this simple approach, no changes are necessary for solution extraction.

Note that the issue of the closed world assumption never rose with respect to the dinner date example because none of the actions had a negative precondition. And while the goal *did* include a negative literal ($\neg$`garbage`) the positive proposition `garbage` was present in the initial conditions thus voiding the closed world assumption.

```
drive-truck37-Seattle-Tacoma
       :precondition (at truck37 Seattle)
       :effect       (and (not (at truck37 Seattle))
                          (at truck37 Tacoma))
```

Figure 8: Ground instance of `drive` after type analysis and elimination of "timeless" (eternally true, static) preconditions. Compare with the schema in figure 7.

2.3.3 Action Schemata, Type Analysis & Simplification

In the dinner date example all of the actions were propositional, but in realistic domains it is much more convenient to define parameterized action schemata. For example, in a logistics domain one might define the operation of driving a truck as shown in figure 7. The intuition is simply that at this level of abstraction driving one vehicle has the same preconditions and effects as driving another, so one should only write it once.

The use of action schemata requires a few changes to the planning graph expansion routine at action levels: the planner must *instantiate* each parameterized schema to create possible ground actions by considering all combinations of appropriately typed constants. For example, in order to handle the `drive` schema, the system must create $O(n^3)$ ground `drive` actions, assuming that there are n constants defined in the initial state of the world.

Many of these ground combinations will be irrelevant, because the selection of constants to parameters will never satisfy the preconditions. For example, if `?v` is bound to `Seattle` then presumably the ground precondition `(vehicle Seattle)` will never be satisfied. So an important optimization involves *type analysis* which determines which predicates represent types, calculates the set of constants that form the *extent* of each type, and then instantiates ground actions only for plausibly typed combinations of constants.

The simplest form of type analysis scans the set of predicates present in the initial conditions which are *absent* from the effects of any action schemata. These predicates (*e.g.* `location` and `vehicle`) are static, so terms formed with these predicates (*e.g.*, `(location Seattle)` and `(vehicle truck37)`) will never change. Thus the planner may conclude that `Seattle` is in the extent of the `location` type and similarly reason about `vehicle`.

Since one can evalute static terms at instantiation time, there is no need to include these terms in the planning graph at all; action schemata that include these terms as preconditions can be simplified (during instantiation) by eliminating static preconditions. Furthermore, this simplification is not limited to unary predicates. For example, if `vehicle`, `location` and `road-connected` are all static, and if instances of these actions are only instantiated for constants which obey these preconditions, then planning graph expansion may add ground action instances (such as the one shown in figure 8) and eliminate static terms from proposition levels in the planning graph.

Fox *et al.* have devised more sophisticated, polynomial-time, planner-independent type-inference methods that deduce state invariants, and they demonstrate that this analysis can significantly speed their version of Graphplan on some domains [29].[7] Their method is based on the observation that a planning domain can be viewed as a collection of finite-state machines where domain constants traverse between states corresponding to predicates.

2.3.4 Regression Focussing

As described previously, the planning graph with d proposition levels contains only[8] those actions which could possibly be executed in the initial state or in a world reachable from the initial state. But many of the actions in the planning graph may be irrelevant to the goal at hand. In other words, the graph expansion algorithm is *uninformed* by the goal of the planning problem, and as a result time may be wasted by adding useless actions and their effects into the graph and by reasoning about mutex relations involving these irrelevant facts.

Two optimizations have been proposed to make graph expansion more goal-directed: heuristically filtering facts from the initial state with a *fact-generation graph* [84] and *backward expansion of the planning graph* [45].

A fact-generation graph is an AND-OR-graph created from a problem goal and domain actions as follows. The root of the graph is an AND node corresponding to the goal and its children are the conjunctive subgoals. Each subgoal P is an OR node whose children correspond to the different ground actions that have P as an effect. This structure would be a tree except in order to avoid exponential blowup, nodes are reused within levels of the graph. We say that an OR-node is *solved* if it is in the initial conditions or if an immediate child is solved; an AND-node is solved if all of its children are solved. Because the fact generation graph ignores subgoal interactions from negative literals, solution of a depth d fact-generation graph is a necessary but insufficient condition for solution of a depth d planning graph. At the risk of incompleteness, one may try to speed planning by eliminating initial conditions (or ground actions) that don't appear (or appear infrequently) in the fact-generation graph [84]. Note that this approach is similar to (and motivated by) McDermott's *greedy regression graph* heuristic [81].

A similar approach, due to [45], provides speedup without sacrificing completeness. Recall that Graphplan follows a simple loop: expand the planning graph with action and proposition levels, then attempt solution extraction, if no plan is found then repeat. Kambhampati modified the loop to first grow the planning graph backwards from the subgoals by an action and proposition level, then grow the graph forwards from the intersection of the initial state and the backward propositional fringe, including only ground actions that were added during backwards propagation and adding in mutex relations, then per-

[7] Fox *al.* also point out that their method can dramatically improve the software-engineering process of designing, debugging and maintaining complex planning domains.

[8] Our use of the word *only* is too strong, since the planning graph might contain some actions which can't ever be executed. Strictly speaking, the planning graph contains a proper superset of the executable actions which is a close approximation of that set.

forming solution extraction if necessary. If solution extraction failed to find a plan, then Kambhampati's system would grow the graph backwards for another pair of levels, compute a new (larger) intersection with the initial state, and resume forward growth. Although Kambhampati's implementation regenerated the graphs from scratch at each stage (duplicating discovery of mutex relations), the resulting planning graph was so much smaller that his system outperformed the basic Graphplan on most problems.

2.3.5 In-Place Graph Expansion

One can avoid duplicated work during regression focussing by exploiting the following observations concerning monotonicity in the planning graph.

- *Propositions are monontonically increasing:* if proposition P is present at level i it will appear at level $i+2$ and in all *subsequent* proposition levels.
- *Actions are monontonically increasing:* if action A is present at level i it will appear at level $i+2$ and in all *subsequent* action levels.
- *Mutexes are monotonically decreasing:* if mutex M between actions A and B is present at level i then M is present at all *previous* action levels in which both A and B appear. The same is true of mutexes between propositions.[9]
- *Nogoods are monotonically decreasing:* If subgoals P, Q, and R are unachievable at level i then they are unachievable at all previous proposition levels. [10]

These observation suggest that one can dispense with a multi-level planning graph altogether. Instead, all one needs is a bipartite graph with action and proposition nodes. Arcs from propositions to actions denote the precondition relation and arcs from actions to propositions encode effects. Action, proposition, mutex, and nogood structures are all annotated with an integer label field; for proposition and action nodes this integer denotes the *first* planning graph level at which the proposition (or action) appears. For mutex or nogood nodes, the label marks the *last* level at which the relation holds. By adding an additional set of labels one may interleave forward and backward expansion of the planning graph. Using this scheme, the time and space costs of the expansion phase are vastly decreased, but the bookkeeping required is surprisingly tricky; see [100] for details and see also the STAN planner's "wavefront" representation.

[9] Proof sketch: if A and B appear at both level i and $i-2$ and are mutex at level i then by definition this mutex must be due to inconsistent effects, interference, or competing needs. If the mutex is due to the first two reasons then the mutex will occur at *every* level containing A and B. However, if the mutex is due to competing needs then there are preconditions P and Q of A and B respectively such that P is mutex with Q at level $i-1$. This propositional mutex can only result from the fact that all level $i-3$ actions supporting P and Q are pairwise mutex, so an inductive argument (combined with action monotonicity) completes the proof.

[10] Proof sketch: if they were achievable at level $i-2$ then adding a level of maintenance actions would achieve them at level i.

2.4 Handling Expressive Action Languages

Until now, our discussion has been restricted to the problem of planning with the STRIPS representation in which actions are limited to quantifier-free, conjunctive preconditions and effects. Since this representation is severely limited, this section discusses extensions to more expressive representations aimed at complex, real-world domains. We focus on disjunctive preconditions, conditional effects, and universally-quantified preconditions and effects because these areas have received the most attention. Koehler has developed methods for handling resource constraints [62], and we discuss work on uncertainty at the end of this paper (after describing methods for compiling planning problems to SAT). But other capabilities such as domain axioms, procedural attachment, numeric fluents, exogenous events, and actions with temporal duration beg for exploration.

2.4.1 Disjunctive Preconditions

It is easy to extend Graphplan to handle disjunctive preconditions. Conceptually, the precondition (which may contain nested ands and ors) is converted to disjunctive normal form (DNF). Then when the planning graph is extended with an action schema whose precondition contains multiple disjuncts, an action instance may be added if *any* disjunct has all of its conjuncts present (nonmutex) in the previous level. During the solution extraction phase, if the planner at level i considers an action with disjunctive preconditions, then it must consider all possible precondition disjuncts at level $i-1$ to ensure completeness.

Disjunctive effects are much harder since they imply nondeterminism — one cannot predict the precise effect of execution in advance. As a result, they require a general approach to uncertainty, which we discuss near the end of this paper.

2.4.2 Conditional Effects

Conditional effects are used to describe actions whose effects are context-dependent. The basic idea is simple: we allow a special `when` clause in the syntax of action effects. `When` takes two arguments, an *antecedent* and a *consequent*; execution of the action will have the consequent's effect just in the case that the antecedent is true immediately before execution (*i.e.*, much like the action's precondition determines if execution itself is legal — for this reason the antecedent is sometimes referred to as a secondary precondition [86]). Note also that, like an action precondition, the antecedent part refers to the world *before* the action is executed while the consequent refers to the world *after* execution. For now, we assume that the consequent to be a conjunction of positive or negative literals. Figure 9 illustrates how conditional effects allow one to define a single action schema that accounts for driving a vehicle that may possibly contain a spare tire and/or cargo.

Three methods have been devised for allowing Graphplan-derivative planners to handle action schemata with conditional effects: *full expansion* [30], *factored*

```
(defschema (drive)
   :parameters   (?v ?s ?d)
   :precondition (and (vehicle ?v) (at ?v ?s)
                      (location ?s) (location ?d)
                      (road-connected ?s ?d))
   :effect       (and (at ?v ?d) (not (at ?v ?s))
                      (when (in cargo ?v)
                            (and (at cargo ?v)) (not (at cargo ?s)))
                      (when (in spare-tire ?v)
                            (and (at spare-tire ?d)) (not (at spare-tire ?s)))))
```

Figure 9: Conditional effects allow the same `drive` schema to be used when the vehicle is empty or contains cargo and/or a spare tire.

expansion [1], and *partially factored* [64]. The simplest approach, full expansion, rewrites an action schema containing conditional effects into a number of mutually exclusive STRIPS schemata by considering all minimal consistent combinations of antecedents in the conditional effects. For example, the action schema in Figure 9 would be broken up into four separate STRIPS schemata (as shown in Figure 10): one for the empty vehicle, one for the vehicle with cargo, one for the vehicle with spare tire, and one for the vehicle with both cargo and spare.

Although full-expansion has the advantage of simplicity, it can result in an exponential explosion in the number of actions. If a spare fuel drum could also be in the vehicle, then full expansion would generate eight STRIPS schemata. In general, if an action has n conditional effects, each containing m antecedent conjuncts, then full expansion may produce as many as n^m STRIPS actions [30]. This explosion is common when the conditional effects are universally quantified as in Figure 11. In essence, this schema has one conditional effect for each object that could possibly be put in the truck. If there were only twenty such cargo items, full expansion would yield over a million STRIPS schemata.

The other two approaches for dealing with conditional effects consider the conditional effects themselves as the primitive elements handled by Graphplan.[11] Note that in contrast to the STRIPS actions produced by full expansion, an action's conditional effects are not mutually exclusive, but neither are they independent since the antecedent of one effect may imply that of another. The advantage of *factored expansion* is an increase in performance. By avoiding the need to expand actions containing conditional effects into an exponential number of plain STRIPS actions, factored expansion yields dramatic speedup. But this increased performance comes at the expense of complexity:

[11] In essence, this makes *all* effects conditional since the action preconditions are added into the antecedent for each conditional effect, and the unavoidable effects (*e.g.*, changing the vehicle's location) form a new conditional effect with *just* the action's preconditions as antecedent.

```
(defschema (drive-empty)
   :parameters   (?v ?s ?d)
   :precondition (and (vehicle ?v) (at ?v ?s)
                      (location ?s) (location ?d)
                      (road-connected ?s ?d)
                      (not (in cargo ?v)) (not (in spare-tire ?v)))
   :effect       (and (at ?v ?d) (not (at ?v ?s))))
(defschema (drive-cargo)
   :parameters   (?v ?s ?d)
   :precondition (and (vehicle ?v) (at ?v ?s)
                      (location ?s) (location ?d)
                      (road-connected ?s ?d)
                      (in cargo ?v) (not (in spare-tire ?v)))
   :effect       (and (at ?v ?d) (not (at ?v ?s))
                      (and (at cargo ?v)) (not (at cargo ?s))))
(defschema (drive-spare)
   :parameters   (?v ?s ?d)
   :precondition (and (vehicle ?v) (at ?v ?s)
                      (location ?s) (location ?d)
                      (road-connected ?s ?d)
                      (not (in cargo ?v)) (in spare-tire ?v))
   :effect       (and (at ?v ?d) (not (at ?v ?s))
                      (and (at spare-tire ?v)) (not (at spare-tire ?s))))
(defschema (drive-both)
   :parameters   (?v ?s ?d)
   :precondition (and (vehicle ?v) (at ?v ?s)
                      (location ?s) (location ?d)
                      (road-connected ?s ?d)
                      (not (in cargo ?v)) (in spare-tire ?v))
   :effect       (and (at ?v ?d) (not (at ?v ?s))
                      (and (at cargo ?v)) (not (at cargo ?s))
                      (and (at spare-tire ?v)) (not (at spare-tire ?s))))
```

Figure 10: The four STRIPS schemas for driving with possible contents.

- Because factored expansion reasons about individual effects of actions (instead of complete actions), more complex rules are required in order to define the necessary mutual exclusion constraints during planning graph construction. The most tricky extension stems from the case when one conditional effect is *induced* by another — *i.e.*, when it is impossible to execute one effect without causing the other to happen as well [1].

- Factored expansion also complicates the solution extraction, because of the need to perform the analog of *confrontation* [88, 107], *i.e.*, prevent the antecedent of undesirable effects from occuring.

The IPP planner [65] uses a third method for handling conditional effects which we call *partially factored expansion*. The primary difference stems from IPP's mutex rules which state that two actions are marked as mutex only if their *unconditional* effects and preconditions are in conflict. This difference allows IPP to do less computation during graph expansion, but reduces the number of mutex constraints that will be found. For most domains, the difference doesn't matter, but in some cases (*e.g.*, the movie watching domain [1]) factored expansion performs exponentially better than IPP.

```
(defschema (drive)
   :parameters   (?v ?s ?d)
   :precondition (and (vehicle ?v) (at ?v ?s)
                      (location ?s) (location ?d)
                      (road-connected ?s ?d))
   :effect       (and (at ?v ?d) (not (at ?v ?s))
                      (forall (object ?o)
                          (when (in ?o ?v)
                                (and (at ?o ?v)) (not (at ?o ?s))))))
```

Figure 11: Universally quantified conditional schemata for driving.

2.4.3 Universal Quantification

The Graphplan descendants IPP [64] and SGP [1] each allow action schemata with universal quantification. In preconditions, universal quantification lets one conveniently describe real world actions like the UNIX `rmdir` command which deletes a directory only if *all* files inside it have already been deleted. Universally quantified effects allow one to describe actions like `chmod *` which set the protection of *all* files in a given directory. Naturally, universal quantification is equally useful in describing physical domains. As shown in figure 11, one can use a universally quantified conditional effect to say that all objects on the vehicle will change loaction as a result of driving.

To add universal quantification to Graphplan, it helps to make several simplifying assumptions. Specifically, assume that the world being modeled has a *finite, static universe of typed objects*. For each object in the universe, the initial state description must include a unary atomic sentence declaring its type.[12] For example, the initial description might include sentences of the form `(vehicle truck37)` and `(location Renton)` where `vehicle` and `location` are types.[13] The assumption of a static universe means that action effects may not assert type information. For example, if an action were allowed to assert `(not (vehicle truck37))` as an effect, then that would amount to the destruction of an object; the assumption forbids destruction or creation of objects.

To assure systematic establishment of goals and preconditions that have universally quantified clauses, one must modify the graph expansion phase to map these formulae into a corresponding ground version. The *Herbrand base* Υ of a first-order, function-free sentence, Δ, is defined recursively as follows:

$$\begin{aligned} \Upsilon(\Delta) &= \Delta \text{ if } \Delta \text{ contains no quantifiers} \\ \Upsilon(\forall_{\mathtt{t1}} x\ \Delta(x)) &= \Upsilon(\Delta_1) \wedge \ldots \wedge \Upsilon(\Delta_n) \end{aligned}$$

[12] See the previous section on Action Schemata for further explanation of types.

[13] It's fine for a given object to have multiple types, but this must be stated explicitly or else some form on inheritance reasoning must be added to the graph expansion process.

where the Δ_i correspond to each possible interpretation of $\Delta(x)$ under the universe of discourse, $\{C_1, \ldots, C_n\}$, *i.e.* the possible objects of type `t1` [31, p. 10]. In each Δ_i, all references to x have been replaced with the constant C_i. For example, suppose that the universe of `vehicle` is {`truck37`, `loader55`, `plane7`}. If Δ is `(forall ((vehicle ?v)) (at ?v Seattle))` then the Herbrand base $\Upsilon(\Delta)$ is the following conjunction:

```
(and (at truck37 Seattle) (at loader55 Seattle) (at plane7 Seattle))
```

Under the static universe assumption, if this goal is satisfied, then the universally quantified goal is satisfied as well. Note that the Herbrand base for a formula containing only universal quantifiers will always be ground, so one may use formulae of this form as action effects. It's easy to handle existential quantifiers interleaved arbitrarily with universal quantification, when the expression is used as a goal, action precondition or the antecedent of a conditional effect. Existential quantifiers are not allowed in action effects because they are equivalent to disjunctive effects and (as described above) imply nondeterminism and hence require reasoning about uncertainty.

In order to handle existential quantification in goals, one needs to extend the definition of Herbrand base as follows.

$$\begin{aligned}
\Upsilon(\exists_{\texttt{t1}} y\ \Delta(y)) &= \texttt{t1}(y) \wedge \Upsilon(\Delta(y)) \\
\Upsilon(\forall_{\texttt{t1}} x\ \exists_{\texttt{t2}} y\ \Delta(x, y)) &= \texttt{t2}(y_1) \wedge \Upsilon(\Delta_1) \wedge \ldots \wedge \texttt{t2}(y_n) \wedge \Upsilon(\Delta_n)
\end{aligned}$$

Once again the Δ_i correspond to each possible interpretation of $\Delta(x, y)$ under the universe of discourse for type `t1`: $\{C_1, \ldots, C_n\}$. In each Δ_i all references to x have been replaced with the constant C_i. In addition, references to y have been replaced with Skolem constants (*i.e.*, the y_i).[14] All existential quantifiers are eliminated as well, but the remaining, free variables (which act as Skolem constants) are implicitly existentially quantified; they will be treated just like action schemata parameters during graph expansion. Since we are careful to generate one such Skolem constant for each possible assignment of values to the universally quantified variables in the enclosing scope, there is no need to generate and reason about Skolem functions. In other words, instead of using $y = f(x)$, we enumerate the set $\{f(C_1), f(C_2), \ldots, f(C_n)\}$ for each member of the universe of x and then generate the appropriate set of clauses Δ_i by substitution and renaming. Since each type's universe is assumed finite, the Herbrand base is guaranteed finite as well. Two more examples illustrate the handling of existential quantification. Suppose that the universe of `location` is {`Seattle`, `Renton`} and that Δ is

```
(exists ((location ?l))
   (forall ((vehicle ?v)) (at ?v ?l)))
```

[14] Note that this definition relies on the fact that type `t1` has a finite universe; as a result n Skolem constants are generated. If there were two leading, universally quantified variables of the same type, then n^2 Skolem constants ($y_{i,j}$) would be necessary.

then the Herbrand base is the following:

```
(and (location ?l) (at truck37 ?l) (at loader55 ?l) (at plane7 ?l))
```

As a final example, suppose Δ is

```
(forall ((location ?l))
   (exists ((vehicle ?v)) (at ?v ?l)))
```

Then the universal base contains two Skolem constants (`?v1` and `?v2`) which are treated as parameters:

```
(and (vehicle ?v1) (at ?v1 Seattle) (vehicle ?v2) (at ?v2 Renton))
```

Since there are only two locations, the Skolem constants `?v1` and `?v2` exhaust the range of the Skolem function whose domain is the universe of vehicles. Because of the finite, static universe assumption, one can always do this expansion when creating the Herbrand base.

In summary, we only allow universal quantifiers in action effects, but goals, preconditions, and effect antecedents may have interleaved universal and existential quantifiers. Quantified formulae are compiled into the corresponding Herbrand base and all remaining variables are treated like action schemata parameters during graph expansion. Since the resulting planning graph contains quantifier-free ground action instances, no changes are required during solution extraction.

3 Compilation of Planning to SAT

Despite the early formulation of planning as theorem proving [39], most researchers have long assumed that special-purpose planning algorithms are necessary for practical performance. Algorithms such as TWEAK [13], SNLP [78], UCPOP [88], and Graphplan [6] may all be viewed as special purpose theorem provers aimed at planning problems.

However, recent improvements in the performance of propositional satisfiability methods [15] call this whole endeavor in doubt. Initial results for compiling bounded-length planning problems to SAT were unremarkable [53], but recent experiments [54] suggest that compilation to SAT might yield the world's fastest STRIPS-style planner.

Figure 12 shows the architecture of a typical SAT-based planning system, *e.g.* MEDIC [22] or Blackbox [55]. The *compiler* takes a planning problem as input, guesses a plan length, and generates a propositional logic formula, which if satisfied, implies the existence of a solution plan; a symbol table records the correspondence between propositional variables and the planning instance. The *simplifier* uses fast (linear time) techniques such as unit clause propagation and pure literal elimination (*e.g.*, [105]) to shrink the CNF formula. The *solver* uses systematic or stochastic methods to find a satisfying assignment which

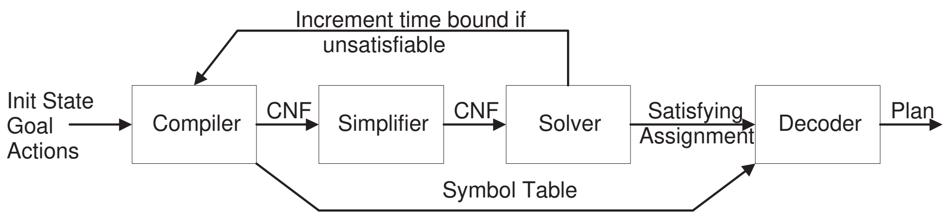

Figure 12: Architecture of a typical SAT-based planning system.

the *decoder* translates (using the symbol table) into a solution plan. If the solver finds that the formula is unsatisfiable, then the compiler generates a new encoding reflecting a longer plan length.

3.1 The Space of Encodings

Compilers for high-level programming languages (*e.g.*, Lisp) are compared on the basis of speed and also on the quality of the machine code they produce. These same notions carry over to SAT compilers as well. One wishes a compiler to quickly produce a small SAT encoding, since solver speed can be exponential in the size of the formula being tested. But this measure of size is complicated by the fact that a propositional formula can be measured in terms of the number of variables, the number of clauses, or the total number of literals summed over all clauses; often a decrease in one parameter (variables, say) will increase another (*e.g.*, clauses). Two factors determine these sizes: the *encoding* and *optimizations* being used. Since the encoding is the more fundamental notion, we focus on it first, presenting a parameterized space of possibilities (developed in [22]) with two dimensions

- The choice of a regular, simply split, overloaded split, or bitwise *action representation* specifies the correspondence between propositional variables and ground (fully-instantiated) plan actions. These choices represent different points in the tradeoff between the number of variables and the number of clauses in the formula.

- The choice of classical or explanatory *frame axioms* varies the way that stationary fluents are constrained.

Each of the encodings uses a standard fluent model in which time takes non-negative integer values. State-fluents occur at even-numbered times and actions at odd times. For example, in the context of the dinner-date problem described previously, the propositional variable $\mathtt{garb}_0$ means that there is garbage in the initial state, $\neg\mathtt{garb}_2$ signifies that there is no garbage after executing the first set of parallel actions, and $\mathtt{carry}_1$ means that the `carry` action is executed at time one.

Each of the encodings uses the following set of *universal axioms*:

INIT The initial state is completely specified at time zero, including all properties presumed false by the closed-world assumption. For the dinner date problem, one gets:

$$\texttt{garb}_0 \wedge \texttt{cleanH}_0 \wedge \texttt{quiet}_0 \wedge \neg\texttt{dinner}_0 \wedge \neg\texttt{present}_0$$

GOAL In order to test for a plan of length n, all desired goal properties are asserted to be true at time $2n$, (but the goal state need not be fully specified). Assuming a desired dinner-date plan length of $n = 1$, one gets:

$$\neg\texttt{garb}_2 \wedge \texttt{dinner}_2 \wedge \texttt{present}_2$$

A$\Rightarrow$P,E Actions imply their preconditions and effects. For each odd time t between 1 and $2n-1$ and for each consistent ground action, an axiom asserts that execution of the action at time t implies that its effects hold at $t+1$ and its preconditions hold at $t-1$. The A$\Rightarrow$P,E can generate numerous clauses when applied to action schemata in the context of a world with many objects, but for the simple non-parameterized dinner-date `cook` action, one gets:

$$(\neg\texttt{carry}_1 \vee \texttt{dinner}_2) \wedge (\neg\texttt{carry}_1 \vee \texttt{cleanH}_0)$$

3.1.1 Action Representation

The first major encoding choice is whether to represent the names of ground action instances in regular, simply split, overloaded split, or the bitwise format. This choice is irrelevant for purely propositional planning problems (scuh as the dinner date example), but becomes crucial given parameterized action schemata (*e.g.*, the STRIPS `drive` schema shown in figure 7). In the **regular** representation, each ground action is represented by a different logical variable, for a total of $n|Schemata||Dom|^{P_s}$ such variables, where n denotes the number of odd time steps, $|Schemata|$ is the number of action schema, P_s denotes the maximum number of parameters per action schemata, and $|Dom|$ is the number of objects in the domain. Since systematic solvers take worst-case time which is exponential in the number of variables, and large numbers of variables also slow stochastic solvers, one would like to reduce this number.

In order to do this, [54] introduced **simple action splitting**, which replaces each n-ary action fluent with n unary fluents throughout the encoding. For example, variables of the form `Drive(Truck37, Seattle, Renton,` t`)` are replaced with the conjunction of `DriveArg1(Truck37,` t`)`, `DriveArg2(Seattle,` t`)`, `DriveArg3(Renton,` t`)`.[15] Doing this for each action reduces the number of

[15] Note that we are using nonstandard notation here in order to emphasize the combinatorics. When we write `DriveArg3(Renton,` t`)` we denote a *propositional variable*, not a functional term from first order predicate calculus. Thus `DriveArg3(Renton,` t`)` is treated as if it has no substructure. To make this aspect clear, we might better write the symbol $\texttt{DriveArg3Renton}_t$, but we prefer our notation because it more clearly illustrates the effects of representational differences on CNF size.

variables needed to represent all actions to $n|Schemata||Dom|P_s$, but each action (formerly a single variable) is now described by a conjunction of P_s variables.

With the simple splitting representation, only instances of the same action schemata share propositional variables. An alternative is **overloaded splitting**, whereby all operators share the same split fluents. Overloaded splitting replaces `Drive(Truck37, Seattle, Renton,` t`)` by the conjunction of `Act(Drive,` t`)`, `Arg1(Truck37,` t`)`, `Arg2(Seattle,` t`)`, `Arg3(Renton,` t`)`, while a different action `Load(Truck37, Drum9,` t`)` is replaced with `Act(Load,` t`)` $\wedge$ `Arg1(Truck37,` t`)` $\wedge$ `Arg2(Drum9,` t`)`. This technique further reduces the number of variables needed to represent all actions to $n(|Schemata| + |Dom|P_s)$.

The **bitwise** representation shrinks the number of variables even more, by representing the action instances with only $\lceil \log_2 |Schemata||Dom|^{P_s} \rceil$ propositional symbols (per odd time step), each such variable representing a bit. The ground action instances are numbered from 0 to $(|Schemata||Dom|^{P_s}) - 1$. The number encoded by the bit symbols determines the ground action which executes at each odd time step. For instance, if there were four ground actions, then ($\neg$`bit1(`t`)`$\wedge\neg$`bit2(`t`)`) would replace the first action, ($\neg$`bit1(`t`)`$\wedge$`bit2(`t`)`) would replace the second, and so forth.

Which action representation is the best? While more experiments need to be performed, preliminary results suggest that the regular and simply split representations are good choices [22]. In contrast, bitwise and overloaded result in convoluted encodings that resist simplification and type analysis. For example, although the bitwise encoding yields the smallest number of propositional variables *before* simplification, the linear-time procedure described in [105] shrunk the CNF formulae from the other representations so much that afterwards bitwise had the most variables.

3.1.2 Frame Axioms

Every encoding requires axioms to confront the frame problem[80].

FRAME Frame axioms constrain unaffected fluents when an action occurs. There are two alternatives: classical or explanatory frames.

Classical frame axioms [80] state which fluents are left unchanged by a given action. For example, one classical frame axiom for the STRIPS `drive` schemata (Figure 7) would say "Driving vehicle `Truck37` from `Seattle` to `Renton` leaves `Truck9`'s location (`Kent`) unchanged,"

(`At(Truck9, Kent,` $t-1$`)`$\wedge$`Drive(Truck37, Seattle, Renton,` t`))` $\Rightarrow$ `At(Truck9, Kent,` $t+1$`)`

Since the encoding is propositional, one must write a version of this axiom for each combination of (1) possible location of `Truck9`, (2) source location for `Truck37`, and (3) destination for `Truck37`. If these aren't the only two trucks, then there will be even more combinations. Note also the use of the regular action representation implied by our choice of variable `Drive(Truck37, Seattle, Renton,` t`)`;

if a different representation is desired, then the frame axiom might contain more literals.

Adding classical frame axioms for each action and each odd time t to the universal axioms almost produces a valid encoding of the planning problem. However, if no action occurs at time t, the axioms of the encoding can infer nothing about the truth value of fluents at time $t+1$, which can therefore take on arbitrary values. The solution is to add AT-LEAST-ONE axioms for each time step.

AT-LEAST-ONE A disjunction of every possible, fully-instantiated action ensures that some action occurs at each odd time step. (A maintenance action is inserted as a preprocessing step.) Note that action representation has a huge effect on the size of this axiom.

The resulting plan consists of a totally-ordered sequence of actions; indeed it corresponds roughly to a "linear" encoding in [52], except that they include exclusion axioms (see below) to ensure that at most one action is active at a time. However, exclusion axioms are unnecessary because the classical FRAME axioms combined with the A⇒P,E axioms ensure that any two actions occurring at time t lead to an identical world-state at time $t+1$. Therefore, if more than one action does occur in a time step, then either one can be selected to form a valid plan.

Explanatory frame axioms [40] enumerate the set of actions that could have occurred in order to account for a state change. For example, an explanatory frame axiom would say which actions could have caused `truck9` to have left `Seattle`.

$$
\begin{aligned}
&(\texttt{At(Truck9, } t{-}1) \wedge \neg\texttt{At(Truck9, } t{+}1)) \Rightarrow \\
&\quad (\texttt{Drive(Truck9, Seattle, Renton, } t) \vee \\
&\quad \texttt{Drive(Truck9, Seattle, Kent, } t) \vee \ldots \vee \\
&\quad \texttt{Drive(Truck9, Seattle, Tacoma, } t))
\end{aligned}
$$

Note (again) that the choice of action representation affects the length of the frame axioms. Furthermore, note that the axiom can be simplified dramatically if a different representation is chosen. For example, if we use the simply split representation then a straight translation yields.

$$
\begin{aligned}
&\texttt{At(Truck9, } t{-}1) \wedge \neg\texttt{At(Truck9, } t{+}1) \Rightarrow \\
&\quad ((\texttt{DriveArg1(Truck9, } t) \wedge \texttt{DriveArg2(Seattle, } t) \wedge \texttt{DriveArg3(Renton, } t)) \vee \\
&\quad (\texttt{DriveArg1(Truck9, } t) \wedge \texttt{DriveArg2(Seattle, } t) \wedge \texttt{DriveArg3(Kent, } t)) \vee \ldots \vee \\
&\quad (\texttt{DriveArg1(Truck9, } t) \wedge \texttt{DriveArg2(Seattle, } t) \wedge \texttt{DriveArg3(Tacoma, } t)))
\end{aligned}
$$

But this disjunction is really just enumerating all the possible destinations, which is silly, so the compiler can do a *factoring* optimization [22] by recognizing which parameters affect which literals, and generating simplified frames axioms[16] For this example, the compiler should generate (the vastly simpler):

[16] In fact, the factoring optimization should be applied to all axiom types — not just frame axioms.

$\texttt{At(Truck9, } t{-}1\texttt{)} \wedge \neg \texttt{At(Truck9, } t{+}1\texttt{)} \Rightarrow (\texttt{DriveArg1(Truck9, } t\texttt{)} \wedge \texttt{DriveArg2(Seattle, } t\texttt{)})$

As a supplement to the universal axioms, explanatory frame axioms must be added for each ground fluent and each odd time t to produce a reasonable encoding. With explanatory frames, a change in a fluent's truth value implies that some action occurs, so (contrapositively) if no action occurs at a time step, this will be correctly treated as a maintenance action. Therefore, no AT-LEAST-ONE axioms are required.

The use of explanatory frame axioms brings an important benefit: since they do not explicitly force the fluents unaffected by an executing action to remain unchanged, explanatory frames permit *parallelism*. Specifically, any actions whose preconditions are satisfied at time t and whose effects do not contradict each other can be executed in parallel. Parallelism is important because it allows one to encode an n step plan with less than n odd time steps, and small encodings are good. But uncontrolled parallelism is problematic because it can create valid plans which have no linear solution. For example, suppose action α has precondition X and effect Y, while action β has precondition $\neg Y$ and effect $\neg X$. While these actions might be executed in parallel (because their effects are not contradictory) there is no legal total ordering of the two actions. Hence, one must explicitly rule out this type of pathologic behavior with more axioms:

EXCLUSION Linearizability of resulting plans is guaranteed by restricting which actions may occur simultaneously.

Two kinds of exclusion enforce different constraints in the resulting plan:

- **Complete** exclusion: For each odd time step, and for all distinct, fully-instantiated action pairs α, β, add clauses of the form $\neg\alpha_t \vee \neg\beta_t$. Complete exclusion ensures that only one action occurs at each time step, guaranteeing a totally-ordered plan.
- **Conflict** exclusion: For each odd time step, and for all distinct, fully-instantiated, conflicting action pairs α, β, add clauses of the form $\neg\alpha_t \vee \neg\beta_t$. In our framework, two actions conflict if one's precondition is inconsistent with the other's effect.[17] Conflict exclusion results in plans whose actions form a partial order. Any total order consistent with the partial order is a valid plan.

Note that conflict exclusion cannot be used in isolation given a split action representation, because splitting causes there not to be a unique variable for each fully-instantiated action. For example, with simple splitting, it would be impossible to have two instantiations of the same action schema execute at the same time, because their split fluents would interfere. Overloaded splitting

[17] Contrast our definition of conflict with that of Graphplan [6] and [54]. Unlike Kautz and Selman's parallel encoding, but like their linear one, our encodings have axioms stating that actions imply their effects; their parallel encoding prohibits effect-effect conflicts instead.

further disallows two instantiations of different actions to execute at the same time, so it requires complete exclusion. Simple splitting may be used with conflict exclusion when augmented with additional axioms that ban multiple instances of a single schema from executing.

The bitwise action representation requires no action EXCLUSION axioms. At any time step, only one fully-instantiated action's index can be represented by the bit symbols, so a total ordering is guaranteed.

What is the best way to represent frame axioms? Experience [54, 22] shows that explanatory frame axioms are clearly superior to classical frames in almost every case. Since parallel actions encode longer plans with the same number of time steps, conflict exclusion should be used whenever possible (*e.g.*, with the regular action representation or with the minimal additional exclusions necessary for the simply split representation).

3.1.3 Other Kinds of Encodings

The MEDIC planning compiler [22] uses the taxonomy described above to generate any of twelve different encodings. In addition, MEDIC incorporates many switch-selectable optimizations such as type analysis; these features make MEDIC a powerful testbed for research in SAT-based planning. But there are several encodings which do not fit in our taxonomy and hence cannot be generated by MEDIC.

The *causal encoding* [52] is based on the causal-link representation used by partial-order planners such as SNLP [78]. While this encoding has been shown to have the smallest encoding when measured asymptotically, the constant factors are large, and despite several efforts no one has succeeded in building a practical compiler based on the idea.

Work has also been done exploring ways of encoding hierarchical task network (HTN) planning [23] as a SAT problem [75].

3.1.4 Comparison with Graphplan

Note the strong similarities between Graphplan-derivative and SAT-based planning systems.

- Both approaches convert parameterized action schemata into a finite propositional structure (*e.g.*, the planning graph and a CNF formula) representing the space of possible plans up to a given length.

- Both approaches use local consistency methods (*e.g.*, mutex propagation and propositional simplification) before resorting to exhaustive search.

- Both approaches iteratively expand their propositional structure until they find a solution.

Indeed, Kautz and Selman [54] showed that the planning graph can be automatically converted into CNF notation for solution with SAT solvers, by constructing propositional formulae stating:

1. The (fully specified) initial state holds at level zero and the goal holds at the highest level (*i.e.*, our INIT and GOAL axioms).

2. Conflicting actions are mutually exclusive (*i.e.*, conflict-based EXCLUSION axioms).

3. Actions imply their preconditions (*i.e.*, the precondition part of our A⇒P,E axioms).

4. Each fact at positive even levels implies the disjunction of all actions at the previous levels (including maintenance actions). For example, consider the dinner date proposition $\neg\texttt{garb}$ at level 4 in figure 5; one obtains:

$$\neg\texttt{garb}_4 \Rightarrow (\texttt{dolly}_3 \vee \texttt{carry}_3 \vee \texttt{maintain-no-garb}_3)$$

Kautz et al. [52] observe that this encoding is very close to the combination of explanatory frames with a regular action representation; there are two differences. First, this encoding does not explicitly include explanatory frame axioms, but they may be generated by resolving axioms of type (4) with the "actions imply their preconditions" axioms for the maintenance actions. Second, there are no axioms stating that actions imply their effects, so spurious actions may be included in the solution (these can be removed later by the decoder). Fortunately, the conflict exclusion axioms prevent these spurious actions from interfering with the rest of the plan.

The BLACKBOX system [55] uses this Graphplan-based encoding to provide a very fast planner. BLACKBOX uses the graph expansion phase of IPP [64] to create the planning graph, then converts the graph into CNF rather than performing traditional solution extraction. One of the keys to BLACKBOX's performance is the observation that the simplification algorithm employed by Graphplan is more powerful than the unit propagation used in their previous SAT planning system [47, 55]. Specifically, Graphplan employs negative binary propagation in a limited way: binary exclusion clauses corresponding to mutex relations (*e.g.*, $\{\neg p \vee \neg q\}$) are resolved against proposition support sets (*e.g.*, $\{p \vee r \vee s \vee \ldots\}$) to infer $\{\neg q \vee r \vee s \vee \ldots\}$.

3.2 Optimizations

There are several ways to improve the encodings discussed above; in this section we discuss compile-time type optimization and the addition of domain-specific information.

The principles and objectives underlying type analysis for SAT-compilation are the same as previously discussed in the context of Graphplan. Graphplan-based approaches (*e.g.*, inertia optimization [64] and TIM [29]) aimed to shrink the size of the planning graph by eliminating static fluents and by avoiding nonsensical action-schemata instantiations. The same approaches can be used to shrink the size of the CNF formula that a SAT-compiler generates. The

MEDIC compiler performs optimizations that reduce CNF size by as much at 69% on a variety of problems [22].

Another way to optimize the CNF formula produced by a compiler is to add domain specific information. Typically this knowledge is impossible to express in terms of STRIPS actions but is natural when writing general logical axioms, and can be induced when processing action schemata and initial state specifications. For example, in the blocks world one might state axioms to the effect that the relation `On` is both non-commutative and irreflexive, only one block may be on another at any time, *etc.*. Ernst *et al.* show that *adding* these types of axioms increased the clause-size of the resulting CNF formulae, but *decreased* the number of variables (after simplification) by 15% and speeded solver time significantly.

Domain axioms may be classified in terms of the logical relationship between the knowledge encoded and the original problem statement [56]:

- Action *conflicts* and *derived effects* are entailed solely by the preconditions and effects of the domain's action schemata.
- Heuristics which are entailed by the initial state in conjunction with the domain's action schemata include *state invariants*. For example, a vehicle can only be in one location at a time.
- *Optimality* heuristics restrict plans by disallowing unnecessary subplans. For example, in a package delivery domain one might specify that packages should never be returned to their original location.
- *Simplifying assumptions* are not logically entailed by the domain definition or goal, but may restrict search without eliminating plans. For example, one might specify that once trucks are loaded they should immediately move.

DISCOPLAN [34] is a preprocessing system that infers state constraints from domain definitions. The basic idea is to look for four general axiom patterns (which can be discovered with low-order polynomial effort). For example, a "single-valuedness constraint" would be discovered for the logistics world, saying that each vehicle can be in only one place at a time. IPP's planning graph is used to discover some of these constraints, while others are deduced using special purpose analysis. No attempt is made to deduce optimality heuristics or simplifying assumptions — all constraints are completeness preserving. Never the less, the CNF formulae with DISCOPLAN-inferred axioms were solved many times faster than plain MEDIC or SATPLAN [54] formulae, regardless of SAT solver used. Indeed, in many cases the plain encodings were unsolvable in the alloted time, while the DISCOPLAN-augmented encodings quickly yielded a plan.

Other researchers have devised alternative methods for detecting constraints. For example, [2] describes a method similar to DISCOPLAN which, in addition, uses regression search to further restrict the predicate domains. Rintanen modified algorithms from computer-aided verification to discover binary invariants [91]. Earlier work on the subject is presented in [57]. Despite these promising first-efforts, much more exciting work remains to done in the area of optimizing SAT encodings for speedy solution.

3.3 SAT Solvers

Without an efficient solver, a planning-to-SAT compiler is useless; in this section we review the state of the art. Perhaps the best summary is that this area of research is highly dynamic. Each year seems to bring a new method which eclipses the previous leader. Selman *et al.* [95] present an excellent summary of the state of the art in propositional reasoning, and sketches challenges for coming years. Our discussion is therefore brief.

SAT solvers are best distinguished by the type of search they perform: *systematic* or *stochastic*.

3.3.1 Systematic SAT Solvers

Although it was discovered many years ago, the DPLL algorithm [17] remains a central algorithm, and it can be summarized with a minimum of background. Let ϕ be a CNF formula, *i.e.*, a conjunction of clauses (disjunctions). If one of the clauses is just a single literal P, then clearly P must be true in order to satisfy the conjunction; P is called a *unit clause*. Furthermore, if there exists some other literal Q such that every clause in ϕ which refers to Q or $\neg Q$, references Q in the same polarity *e.g.*, all references are true (or all are false) then Q (or $\neg Q$) is said to be a *pure literal*. For example, in the CNF formula below

$$\psi = (A \vee B \vee \neg E) \wedge (B \vee \neg C \vee D) \wedge (\neg A) \wedge (B \vee C \vee E) \wedge (\neg D \vee \neg E)$$

$\neg A$ is a unit clause and B is a pure literal. We use the notation "$\phi(u)$" to denote the result of setting literal u true and then simplifying. For example, $\psi(\neg A)$ is

$$(B \vee \neg E) \wedge (B \vee \neg C \vee D) \wedge (B \vee C \vee E) \wedge (\neg D \vee \neg E)$$

and $\psi(B)$ is

$$(\neg A) \wedge (\neg D \vee \neg E)$$

We can now describe DPLL in simple terms; it performs a backtracking, depth-first search through the space of partial truth assignments, using unit-clause and pure-literal heuristics (figure 13). Tableau [16] and Satz [71] are tight implementations of DPLL with careful attention to datastructures and indexing. Many additional heuristics have been proposed to guide the choice of a splitting variable in preparation for the divide and conquer recursive call. For example, Satz selects variables by considering how much unit propagation is facilitated if it branches on that variable [71]. See [15] for a discussion of other heuristics.

Procedure DPLL(CNF formula: ϕ)
  If ϕ is empty, return `yes`.
  Else if there is an empty clause in ϕ return `no`.
  Else if there is a pure literal u in ϕ return DPLL($\phi(u)$)
  Else if there is a unit clause $\{u\}$ in ϕ return DPLL($\phi(u)$)
  Else
    Choose a variable v mentioned in ϕ.
    If DPLL($\phi(v)$) = `yes` then return `yes`.
    Else return DPLL($\phi(\neg v)$).

Figure 13: Backtracking, depth-first search through the space of partial truth assignments.

Procedure GSAT(CNF formula: ϕ, integer: N_{restarts}, N_{flips})
  For i equals 1 to N_{restarts},
    Set A to a randomly generated truth assignment.
    For j equals 1 to N_{flips},
      If A satisfies ϕ then return `yes`.
      Else
        Set v to be a variable in ϕ whose change gives the largest increase
          in the number of staisfied clauses; break ties randomly.
        Modify A by flipping the truth assignment of v.

Figure 14: Random-restart, hill-climbing search through the space of complete truth assignments.

By incorporating CSP "Look Back" techniques such as conflict-directed backjumping and and its generalization, relevance-bounded learning, solver speed was increased substantially [5].

Another interesting direction is the construction of special-purpose SAT solvers, optimized for CNF encodings of planning problems. A first effort in this direction is based on the insight that propositional variables corresponding to action choices are more important than other variables (*e.g.*, those corresponding to fluent values) which follow deterministically from action choices. This insight suggests a small change to DPLL: restrict the choice of splitting variables to action variables. Interestingly, the result of this restriction is dramatic: up to four orders of magnitude speedup [35]. The MODOC solver [104, 85] also uses the high-level structure of the planning problem to speed the SAT solver, but MODOC uses knowledge of which propositions correspond to goals (rather than to actions) to guide its search; the resulting solver is competitive with Walksat (described below).

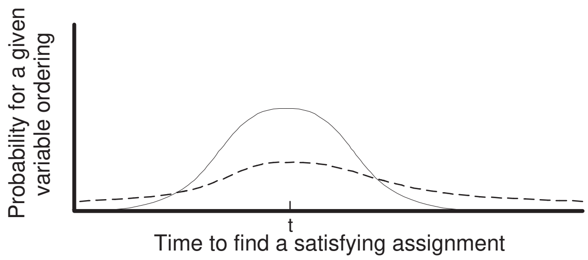

Figure 15: The time required by DPLL to find a satisfying assignment is highly dependent on the order in which variables are chosen. While the distribution of times as a function of variable order varies from problem to problem, many problems show a heavy tailed distribution (dashed curve) instead of Gaussian (hairline curve). Note that the mean value for a heavy tailed distribution can be infinite, because probability mass stretches rightwards without bound. However, since there is a sizable probability mass to the left of time t, one is likely to land in that area after a small number of restarts sample different orderings.

3.3.2 Stochastic SAT Solvers

In contrast to systematic solvers, stochastic methods search locally using random moves to escape from local minima. As a result, stochastic methods are incomplete — when called on hard problems a stochastic solver may simply report that it is unable to find a satisfying assignment in the allotted time. This output leaves the observer uninformed since there is no sure way to distinguish an unsatisfiable formula from one whose satisfying assignment is difficult to find. On the other hand, stochastic solvers are *frequently* much faster at finding satisfying assignments when they exist.

The simple and popular GSAT solver is a random-restart, hill-climbing search algorithm (figure 14) [94]. The successors of a truth assignment are assignments that differ only in the value assigned to a single variable. GSAT performs a greedy search, prefering one assignment over another based on the number of satisfied clauses. Note that the algorithm may move sideways (no change in the number of satisfied clauses) or make negative progress. After hill climbing for a fixed amount of flips (as directed by N_{flips}), GSAT starts anew with a freshly generated, random assignment. After N_{restarts} many of these restarts, GSAT gives up.

WALKSAT[18] [92, 93] improves upon GSAT by adding additional randomness akin to simulated annealing. On each flip, WALKSAT does one of two things; with probability p it chooses the same variable GSAT would have chosen, otherwise it selects a random variable from an unsatisfied clause. Many variants on these algorithms have been constructed and compared, *e.g.* [32, 79].

An especially promising new method, reported in [38], exploits the fact that the time required by the DPLL procedure is highly dependent on the choice of

[18] Download from `http://www.informatik.tu-darmstadt.de/AI/SATLIB`.

splitting variable, producing a heavy-tailed distribution of running times (figure 15). They augmented a version of DPLL by (1) adding randomization to the choice of splitting variable, and (2) causing the algorithm to quit and restart if it failed to find a solution after a very small time limit t. These restarts curtail unpromising choices (*i.e.*, ones that might lead to extremely long running times) before they consume much time. After a number of restarts the odds are high that the algorithm will stumble on a good choice that leads to a quick solution (*i.e.*, one with a running time less than the time limit t). Since very little time is wasted on the restarts, the result is a speedup of several orders of magnitude.

While stochastic methods can perform extremely well, their performance is usually sensitive to a variety of parameters: random noise p, N_{restarts}, N_{flips}, *etc.*. Since the optimal values for these parameters are a function of the problem being solved and the specific algorithm in question, it can take considerable experimentation to "tune" these parameters for a specific problem distribution. For stochastic methods to reach their potential, automated tuning methods (which don't require solving complete problem instances!) must be developed; [79] reports on work in this direction.

3.3.3 Incremental SAT Solving

The problem of propositional satisfiability is closely related to that of truth maintenance [20, 77, 18]; we focus on LTMS-style truth maintenance systems [76]. Both problems concern a CNF formula represented as a set of clauses Σ over a set of propositional variables $\mathcal{V}$. A SAT solver seeks to find a truth assignment (*i.e.*, a function from $\mathcal{V}$ to $\{\texttt{true}, \texttt{false}\}$) that makes Σ true. An LTMS has two differences:

- An LTMS manipulates a function from $\mathcal{V}$ to $\{\texttt{true}, \texttt{false}, \texttt{unknown}\}$, which is more general than a truth assignment.
- An LTMS doesn't just *find* this mapping, it *maintains* it during incremental changes (additions and deletions) to the set of clauses Σ.

An LTMS uses unit propagation to update its mapping. Any unit clauses can be assigned values immediately, and a clause with a single `unknown` literal and all remaining literals labeled `false` can also be updated. If a new clause is added to Σ it may enable additional inference, and dependency records allow the LTMS to retract inferences that depended on clauses later removed from Σ. Nayak and Williams [83] describe an especially efficient method for maintaining this mapping (which they call an ITMS), and the resulting algorithm is a powerful foundation for building real-time planning and execution systems, as we describe below.

4 Interleaved Planning & Execution Monitoring

One of the most exciting recent developments is a partially SAT-based reactive control system that will command the NASA *Deep Space One* autonomous

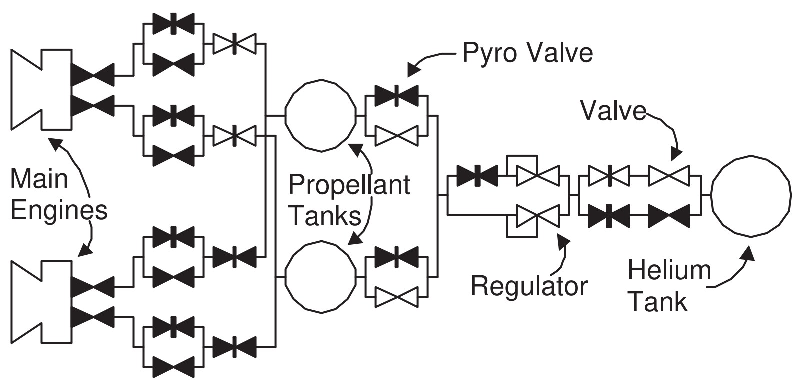

Figure 16: Schematics for spacecraft engine (adapted from [111]). Closed valves are shown filled black. Pyro values may be opened (or closed) only once.

spacecraft which is to be launched in autumn 1998. As one would expect given the magnitude of the task, the complete agent is quite complex [87]; we focus on the configuration planning and execution subsystem [110, 111] which are best explained with an example. Figure 16 shows a simplified schematic for the main engines of a spacecraft. The helium tank pressurizes the fuel and oxidizer tanks, so that they are forced through valves (if open), to combine in the engines where these propellants ignite to produce thrust. Valves are opened by valve drivers by sending commands through a control unit. During its long voyage towards its destination (perhaps Saturn), as many components as possible are turned off in order to save energy, so these control units and drivers must be both turned-on and operational before the valves can be adjusted. Radiation makes space a harsh environment that can damage both electronic and physical components. Valves may jam open or shut, and control units may fail in either a soft (resettable) or permanent fashion. To counteract these problems, the engines have a high degree of redundancy (figure 16). However, some of these propellant paths are more flexible than others — *e.g.*, pyro valves are less likely to fail, but may be switched only once.

The spacecraft's configuration management system must satisfy high-level goals (*e.g.*, "achieve thrust" before orbital insertion) by identifying when failures have occurred and executing actions (*e.g.*, powering on control units and switching valves) so that these goals are quickly achieved at minimum cost in terms of power and spent pyro valves. As shown in figure 17, these decisions are made by a pipeline of three major components:

- The **execution monitor** (MI)[19] interprets limited sensor readings to de-

[19]For clarity and consistency, we use different terminology than Williams and Nayak's original papers [110, 111], and we include their acronyms to facilitate correspondence for reader recourse to primary literature. Williams and Nayak name the process of execution monitoring *mode identification*, hence the abbreviation "MI." The intuition is that the system's "mode" is its state and hence execution monitoring determines whether the system is in the expected

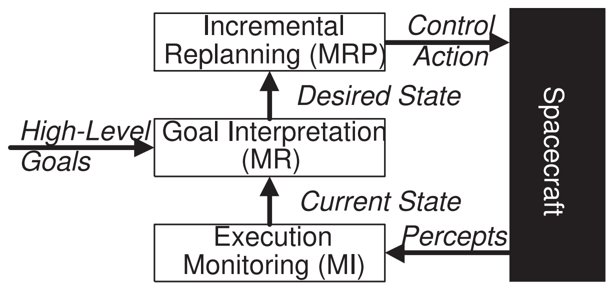

Figure 17: Architecture of the *Deep Space One* spacecraft configuration planning and execution system.

termine the current physical state of the spacecraft; this includes recognizing when an execution failure has occurred. Frequently, there will be several possible states consistent with previous values and current sensors, and in this case the execution monitor returns the mostly likely single state.

- The **goal interpreter** (MR) determines the set of spacecraft states that achieve the high-level goals and are reachable from the current state. It returns the lowest cost such state, *e.g.* one with minimal power consumption and the fewest blown pyro valves.
- The **incremental replanner** (MRP) calculates the first action of a plan to reach the state selected by goal interpretation.

4.1 Propositional Encoding of Spacecraft Capabilities

Each of these modules utilizes a propositional encoding of spacecraft capabilities which (despite superficial differences) is quite similar to the STRIPS domain theories considered earlier in this paper. For example, each valve in the engine is described in terms of the following *modeling variables*: `valve-mode`, $\mathtt{f_{in}}$, $\mathtt{f_{out}}$, $\mathtt{p_{in}}$, $\mathtt{p_{out}}$. These variables have the domains shown below:

$$\begin{array}{l}
\mathtt{valve\text{-}mode} \in \{\mathtt{open}, \mathtt{closed}, \mathtt{stuck\text{-}open}, \mathtt{stuck\text{-}closed}\} \\
\mathtt{f_{in}}, FOUT \in \{\mathtt{positive}, \mathtt{zero}, \mathtt{negative}\} \\
\mathtt{p_{in}}, POUT \in \{\mathtt{high}, \mathtt{nominal}, \mathtt{low}\}
\end{array}$$

Note the use of a discretized, *qualitative representation* [9, 108] of the real-valued flow and pressure variables. The cross product of these five variables defines

state or if a failure has occurred. The process of goal interpretation was called *mode reconfiguration* (hence "MR"), and what we call incremental replanning was called "model-based reactive planning" (MRP).

a space of possible valve states, but many of these states are not physically attainable. For example, if the valve is open, there will be equal pressure on both sides, so $\mathtt{p_{in}} = \mathtt{p_{out}}$. Infeasible states are eliminated by writing a set of propositional logic formulae in which the underlying propositions are of the form "modeling variable = value." For the valve example, one may write:

$$
\begin{aligned}
&((\texttt{valve-mode} = \texttt{open}) \vee (\texttt{valve-mode} = \texttt{stuck-open})) \Rightarrow \\
&\quad ((\mathtt{p_{in}} = \mathtt{p_{out}}) \wedge (\mathtt{f_{in}} = \mathtt{f_{out}})) \\
&((\texttt{valve-mode} = \texttt{closed}) \vee (\texttt{valve-mode} = \texttt{stuck-closed})) \Rightarrow \\
&\quad ((\mathtt{f_{in}} = \texttt{zero}) \wedge (\mathtt{f_{out}} = \texttt{zero}))
\end{aligned}
$$

Note that these descriptions are implicitly parameterized. Each valve has it's own variables and thus its own propositions. Augmenting these domain axioms, control actions are specified in a temporal, modal logic that adds a "next" operator, $\bigcirc$, to propositional logic. For example, the behavior of a valve driver[20] may be partially described with the following formulae:

$$
\begin{aligned}
&((\texttt{driver-mode} \neq \texttt{failed}) \wedge (\mathtt{cmd_{in}} = \texttt{reset})) \Rightarrow \\
&\quad \bigcirc(\texttt{driver-mode} = \texttt{on}) \\
&((\texttt{driver-mode} = \texttt{on}) \wedge (\mathtt{cmd_{in}} = \texttt{open}) \wedge (\texttt{valve-mode} \neq \texttt{stuck-closed})) \Rightarrow \\
&\quad \bigcirc(\texttt{valve-mode} = \texttt{open})
\end{aligned}
$$

Each of these *transition equations* is akin to a STRIPS operator — the antecedent (*i.e.*, left hand side) of the implication corresponds to the action name and precondition, while the consequent (right-hand side) equations that follow the $\bigcirc$ are effects that take place in the next time step. The name/parameter distinction is a bit subtle, because it stems from the fact that modeling variables are partitioned into disjoint sets: *state variables* (*e.g.*, `driver-mode`), *dependent variables* (*e.g.*, $\mathtt{f_{in}}$), and *control variables* (*e.g.*, $\mathtt{smd_{in}}$). The subexpression of the antecedent that consists solely of propositions referring to control variables[21] corresponds to the *name* of a STRIPS action, and the remainder of the antecedent (*i.e.*, propositions referring only to dependent and state variables) corresponds to the STRIPS action precondition. In summary, therefore, Williams and Nayak model the spacecraft with a combination of STRIPS actions and propositional constraints defining the space of feasible states.

4.2 Real-Time Inference

Suppose that the agent knows the state of all modeling variables (and hence all propositions) at time zero. This information suffices to predict which actions

[20] Again, we ignore parameterization.

[21] By assumption [22], every transition equation has at least one proposition involving control variables in its antecedent.

will be executed, and hence the expected, next spacecraft-state. But actions don't always have their desired effects, so Williams and Nayak included additional transition equations that describe possible failure modes as well. For example, in contrast to the expected result of setting $\texttt{cmd}_{\texttt{in}} = \texttt{open}$ shown above, a failure transition might enumerate the possibility that `valve-mode` could become `stuck-closed`. Both normal and failure transition rules are annotated with probabilities. Thus, instead of predicting a unique next spacecraft-state, one may predict a ranked list of possible next states, ordered by likelihood.

Given this framework, the processes of execution monitoring and goal interpretation can be cast in terms of combinatorial optimization subject to a set of propositional logic constraints. As input execution monitoring takes a set of observations of the current values of a subset of the state and dependent variables, and these observations set the values of the corresponding propositions. Thus, execution monitoring is seen to be the problem of finding the most likely next state which is logically consistent with the current observations. An incremental SAT solver forms the core of this optimization computation.

Goal interpretation is similar. The objective is to find a spacecraft-state which entails the high-level goals, and which is most-cheaply reached from the state that execution monitoring deems most likely. Estimating the cost of reaching a goal state is relatively easy (*e.g.*, one compares the number of switched pyro valves and the differential power usage) so logical entailment, computed by the incremental SAT solver is, again, central.

The incremental replanner takes as input the (most likely) initial state computed by execution monitoring, the (least cost) goal state computed by goal interpretation. As output the incremental replanner produces an action that is guaranteed to be the first step of a successful[23], cycle-free plan from the initial state to the goal. The beauty of Williams and Nayak's algorithm is its guarantee of a speedy response, which at first glance appears to contradict results showing STRIPS planning is PSPACE-complete [12].

Underlying Williams and Nayak's method is the insight that spacecraft configuration planning is far easier than general STRIPS planning, because spacecraft engineers specifically designed their creations to be controllable. Williams and Nayak formalize this intuition with a set of crisp constraints that are satisfied by the spacecraft domain. The most important of these restrictions is the presence of a *serialization ordering* for any (satisfiable) set of goals. As previous theoretical work has shown [67, 4], serialized subgoals can be solved extremely quickly, because no backtracking is necessary between subgoals. To give an intuitive blocksworld example, the set of goals

1. Have block `C` on the `table`.
2. Have block `B` on block `C`.
3. Have block `A` on block `B`.

[23] In the absence of failure transitions.

is serializable, and solving them in the order 1, 2, 3 is the correct serialization. It doesn't matter how goal 1 is achieved, goals 2 and 3 can be solved without making goal 1 false. Once goals 1 and 2 are achieved, they never need be violated in order to solve goal 3. In summary, researchers have long known that serializable goals were an ideal special case, but Williams and Nayak's contribution is twofold. First, they recognized that their spacecraft configuration task was serializable (many real-world domains are not), and second they developed a fast algorithm for computing the correct order. This last step is crucial, because if one attempts to solve a serializable problem in the wrong order, than an exponential amount of time can be wasted by backtracking search. For example, if one solved goal 3 above (getting block `A` on block `B`) before solving 1 and 2, the work on goal 3 might be wasted.

Williams and Nayak's goal ordering algorithm is based on the notion of a *causal graph*[24] whose vertices are state variables; a directed edge is present from v_1 to v_2 if a proposition mentioning v_1 is in the antecedent of a transition equation whose consequent mentions v_2.[25] Williams and Nayak observe that the spacecraft-model causal graphs are acyclic and thus a topological sort of the graph yields a serialization ordering. If the goals are solved in an "upstream" order (*i.e.*, goals involving v_2 are solved before those of v_1), then no backtracking is required between goals. Essentially all search is eliminated, and the incremental replanner generates control actions in real-time.

5 Discussion

While, the focus of this paper has been on the dramatic explosion in SAT-planning and Graphplan-based algorithms, we close by briefly mentioning some other recent trends.

5.1 Planning as Search

Refinement search forms an elegant framework for comparing different planning algorithms and representations [51, 48]. Recent results extend the theory to handle partially HTN domains [46].

McDermott showed that an emphasis on (automatically) computing an informative heuristic can make an otherwise simple planner extremely effective [81]. TLPLAN uses (user-provided) domain specific control information to offset a simple, forward-chaining search strategy — with impressive results [2]. Hector

[24] It is interesting to compare this work with similar research on subgoal ordering discussed earlier in the section on *Solution Extraction as Constraint Satisfaction*. Problem-space graphs [25] and operator graphs [97, 98] share many resemblances to causal graphs. Knoblock's ALPINE abstraction system [58] can be viewed as finding a serialization ordering, and it can eliminate most search when given a problem with acyclic structure such as the towers of Hanoi [60].

[25] The causal graph is constructed offline from a compiled version of the domain theory which eliminates all reference to dependent variables.

Geffner demonstrated impressive performance on planning competition problems using heuristic search through the space of world-states.

5.2 Causal Link Planning

Causal link planners, *e.g.* SNLP [78] and UCPOP [88], have received less attention in recent years because they are outperformed by Graphplan and SATPLAN in most domains. However, some of the intuitions underlying these planners have been adopted by the propositional approaches. For example, one of the biggest advantages of causal-link planners, resulting from their backward-chaining regression search, was their insensitivity to irrelevant information in the initial state. Regression focussing (described previously) provides some of these advantages to propositional planners.

One situation where causal link planners still seem to excel are software domains in which the domain of discourse is unknown to the agent [24]. When an agent is faced with incomplete information, it can not construct the Herbrand base and hence is unable to use propositional planning methods. Causal-link planners such as XII [37] and PUCCINI [36], on the other hand, work competently.

5.3 Handling Uncertainty

Starting with work on the CNLP [89], SENSp [26], Buridan [68, 69] and C-Buridan [21] systems, the AI planning community has more seriously considered extensions to action languages that allow the specification of uncertain effects and incomplete information. Of course, much related work has been performed by the UAI community, but usually with different assumptions. For example, work on Markov Decision Processes (MDPs) typically assumes that an agent has complete, immediate, free observability of the world state, even if its own actions are not completely deterministic. Work on Partially-Observable MDPs relaxes this assumption, but much remains to be done in this area since POMDP solvers are typically much less efficient than MDPs. MDP and POMDP researchers typically state the agent's objective in terms of maximizing a utility function over a fixed, finite horizon. Planning researchers, on the other hand, usually seek to achieve a fixed goal configuration, either with complete confidence or with probability greater than some threshold, but no time horizon is considered. In the past it was thought that planning-based approaches (by their goal-directed natures) were less sensitive to high-dimension state descriptions, *i.e.* the presence of many attributes in the initial state. However, recent work on MDP abstraction and aggregation [11, 19] calls this intuition into question. In order for the field to advance, more work needs to be done comparing these approaches and testing their relative strengths and limitations. Initial results in this area are a start [10, 72], but empirical comparisons are badly needed.

Several researchers have extended Graphplan to handle uncertainty. Conformant graphplan (CGP) [99] handles uncertainty in the initial state and in action effects, but does not allow sensing; the resulting "conformant" plan works in the presence of uncertainty by choosing robust actions that cover all eventualities. Sensory graphplan (SGP) [109] extends CGP to allow branching ("contingent") plans based on run-time information gathered by noiseless sensory actions which may have preconditions. Neither CGP nor SGP incorporate numerical probabilistic reasoning; both build separate planning graph structures for each possible world specified by the problem's uncertainty, and so scaling is a concern. PGraphplan [8] adopts the MDP framework (*i.e.*, numerical probability, complete observability) and builds an optimal n-step, contingent plan using a single planning-graph-like structure to accelerate forward-chaining search (see also [11]).

Other researchers have investigated the compilation approach to planning under uncertainty, but instead of compiling to SAT, they target a probabilistic variant called E-MAJSAT:

> Given a Boolean formula with *choice variables* (variables whose truth status can be arbitrarily set) and *chance variables* (variables whose truth status is determined by a set of independent probabilities), find the setting of the choice variables that maximizes the probability of a satisfying assignment with respect to the chance variables. [72]

[73] describes a planning compiler based on this idea, and presents a E-MAJSAT solver akin to DPLL. Caching expensive probability calculations leads to impressive efficiency gains [74].

5.4 Conclusions

In the past few years, the state of the art in AI planning systems has advanced with extraordinary speed. Graphplan and SAT-based planning systems can quickly solve problems that are orders of magnitude harder than those tackled by the best previous planners. Recent developments extend these systems to handle expressive action languages, metric resources, and uncertainty. Type-theoretic domain analysis promises to provide additional speedup, and there are likely more ideas in from the constraint satisfaction and compiler areas which could be usefully applied. The use of a modern planning system to control a real NASA spacecraft demonstrates that AI planning has matured enough as a field to increase the number of fielded applications. A common thread running through all of this research is the use of propositional representations, which support extremely fast inference.

Acknowledgements

We thank Corin Anderson, Mike Ernst, Mark Friedman, Alfonso Gerevini, Rao Kambhampati, Henry Kautz, Todd Millstein, Bart Selman, David Smith, Brian Williams, and Steve Wolfman for enlightening discussions on planning and for comments on this paper. This research was funded by Office of Naval Research Grant N00014-98-1-0147, by National Science Foundation Grant IRI-9303461, and by ARPA / Rome Labs grant F30602-95-1-0024.

References

[1] C. Anderson, D.E. Smith, and D. Weld. Conditional effects in graphplan. In *Proc. 4th Intl. Conf. AI Planning Systems*, June 1998.

[2] F. Bacchus and Y. W. Teh. Making forward chaining relevant. In *Proc. 4th Intl. Conf. AI Planning Systems*, pages 54–61, June 1998.

[3] F. Bacchus and P. van Run. Dynamic variable ordering in csps. In *Proceedings of the 1995 conference on Principles and Practice of Constraint Programming*, pages 258–275, September 1995.

[4] A. Barrett and D. Weld. Partial order planning: Evaluating possible efficiency gains. *J. Artificial Intelligence*, 67(1):71–112, 1994.

[5] R. Bayardo and R. Schrag. Using csp look-back techniques to solve real-world SAT instances. In *Proc. 14th Nat. Conf. AI*, Providence, R.I., July 1997.

[6] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proc. 14th Int. Joint Conf. AI*, pages 1636–1642, 1995.

[7] A. Blum and M. Furst. Fast planning through planning graph analysis. *J. Artificial Intelligence*, 90(1–2):281–300, 1997.

[8] A. L. Blum and J. C. Langford. Probabilistic planning in the graphplan framework. In *AIPS98 Workshop on Planning as Combinatorial Search*, pages 8–12, June 1998.

[9] D. Bobrow, editor. Special issue on qualitative reasoning about physical systems. *J. Artificial Intelligence*, 24, December 1984.

[10] C. Boutilier, T. Dean, and S. Hanks. Planning under uncertainty: Structural assumptions and computational leverage. In *Proceedings of the Second European Workshop on Planning*, 1995.

[11] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *Proc. 14th Int. Joint Conf. AI*, pages 1104–1111, August 1995.

[12] T. Bylander. Complexity results for planning. In *Proceedings of IJCAI-91*, pages 274–279, 1991.

[13] D. Chapman. Planning for conjunctive goals. *J. Artificial Intelligence*, 32(3):333–377, 1987.

[14] J. Cheng and K. B. Irani. Ordering problem subgoals. In *Proc. 11th Int. Joint Conf. AI*, pages 931–936, August 1989.

[15] S. Cook and D. Mitchell. Finding hard instances of the satisfiability problem: A survey. *Proceedings of the DIMACS Workshop on Satisfiability Problems*, To Appear, 1997.

[16] J. Crawford and L. Auton. Experimental results on the cross-over point in satisfiability problems. In *Proc. 11th Nat. Conf. AI*, pages 21–27, 1993.

[17] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *C. ACM*, 5:394–397, 1962.

[18] J. de Kleer. An assumption-based truth maintenance system. *J. Artificial Intelligence*, 28:127–162, 1986.

[19] R. Dearden and C. Boutilier. Abstraction and approximate decision-theoretic planning. *J. Artificial Intelligence*, 89(1-2):219–283, 1997.

[20] J Doyle. A truth maintenance system. *J. Artificial Intelligence*, 12:231–272, 1979.

[21] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proc. 2nd Intl. Conf. AI Planning Systems*, June 1994.

[22] M. Ernst, T. Millstein, and D. Weld. Automatic sat-compilation of planning problems. In *Proc. 15th Int. Joint Conf. AI*, 1997.

[23] K. Erol, J. Hendler, and D. Nau. HTN planning: Complexity and expressivity. In *Proc. 12th Nat. Conf. AI*, pages 1123–1128, July 1994.

[24] O. Etzioni and D. Weld. A softbot-based interface to the Internet. *C. ACM*, 37(7):72–6, 1994.

[25] Oren Etzioni. Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62(2):255–302, 1993.

[26] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 115–125, 1992.

[27] B. Falkenhainer and K. Forbus. Setting up large scale qualitative models. In *Proc. 7th Nat. Conf. AI*, pages 301–306, August 1988. Reprinted in [108].

[28] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *J. Artificial Intelligence*, 2(3/4), 1971.

[29] M. Fox and D. Long. The automatic inference of state invariants in TIM. Technical Report 11/98, University of Durham, UK, 1998.

[30] B. Gazen and C. Knoblock. Combining the expressivity of UCPOP with the efficiency of Graphplan. In *Proc. 4th European Conference on Planning*, Sept 1997.

[31] M. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[32] I. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for sat. In *Proc. 11th Nat. Conf. AI*, pages 28–33. MIT Press(AAAI), July 1993.

[33] A. Gerevini and L. Schubert. Accelerating partial-order planners: Some techniques for effective search control and pruning. *J. Artificial Intelligence Research*, 5:95–137, 1996.

[34] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Proc. 15th Nat. Conf. AI*, Madison, WI, July 1998.

[35] E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In *Proc. 15th Nat. Conf. AI*, pages 948–953, Madison, WI, July 1998.

[36] K. Golden. Leap before you look: Information gathering in the PUCCINI planner. In *Proc. 4th Intl. Conf. AI Planning Systems*, June 1998.

[37] Keith Golden, Oren Etzioni, and Dan Weld. Omnipotence without omniscience: Sensor management in planning. In *Proc. 12th Nat. Conf. AI*, pages 1048–1054, 1994.

[38] C.P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. 15th Nat. Conf. AI*, pages 431–437, Madison, WI, July 1998.

[39] C. Green. Application of theorem proving to problem solving. In *Proc. 1st Int. Joint Conf. AI*, pages 219–239, 1969.

[40] A. Haas. The case for domain-specific frame axioms. In *The Frame Problem in Artificial Intellegence, Proceedings of the 1987 Workshop*. Morgan Kaufmann, 1987.

[41] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *J. Artificial Intelligence*, 14:263–313, 1980.

[42] K. B. Irani and J. Cheng. Subgoal ordering and goal augmentation for heuristic problem solving. In *Proc. 10th Int. Joint Conf. AI*, pages 1018–1024, August 1987.

[43] D. Joslin and M. Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proc. 12th Nat. Conf. AI*, July 1994.

[44] D. Joslin and M. Pollack. Is "early commitment" in plan generation ever a good idea? In *Proc. 13th Nat. Conf. AI*, pages 1188–93, 1996.

[45] R. Kambhampati, E. Lambrecht, and E. Parker. Understanding and extending graphplan. In *Proc. 4th European Conference on Planning*, Sept 1997.

[46] R. Kambhampati, A. Mali, and B. Srivastava. Hybrid planning for partially hierarchical domains. In *Proc. 15th Nat. Conf. AI*, pages 882–888, 1998.

[47] S. Kambhampati. Challenges in bridging plan synthesis paradigms. In *Proc. 15th Int. Joint Conf. AI*, pages 44–49, 1997.

[48] S. Kambhampati. Refinement planning as a unifying framework for plan synthesis. *AI Magazine*, 18(2):67–97, 1997.

[49] S. Kambhampati. Ebl and ddb for grapplan. Department of Computer Science and Engineering TR-99-008, Arizona State University, August 1998.

[50] S. Kambhampati. On the relations between intelligent backtracking and failure-driven explanation based learning in constraint satisfaction and planning. Department of Computer Science and Engineering TR-97-018, Arizona State University, 1998. To appear in *Artificial Intelligence*.

[51] S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. *J. Artificial Intelligence*, 76:167–238, 1995.

[52] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proc. 5th Int. Conf. Principles of Knowledge Representation and Reasoning*, 1996.

[53] H. Kautz and B. Selman. Planning as satisfiability. In *Proc. 10th Eur. Conf. AI*, pages 359–363, Vienna, Austria, 1992. Wiley.

[54] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. 13th Nat. Conf. AI*, pages 1194–1201, 1996.

[55] H. Kautz and B. Selman. Blackbox: A new approach to the application of theorem proving to problem solving. In *AIPS98 Workshop on Planning as Combinatorial Search*, pages 58–60, June 1998.

[56] H. Kautz and B. Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proc. 4th Intl. Conf. AI Planning Systems*, June 1998.

[57] Kelleher and Cohen. Automatically synthesizing domain constraints from operator descriptions. In *Proc. 10th Eur. Conf. AI*, Vienna, Austria, 1992. Wiley.

[58] C. Knoblock. Learning abstraction hierarchies for problem solving. In *Proc. 8th Nat. Conf. AI*, pages 923–928, August 1990.

[59] C. Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, Carnegie Mellon University, 1991. Available as technical report CMU-CS-91-120.

[60] C. Knoblock. An analysis of ABSTRIPS. In *Proc. 1st Intl. Conf. AI Planning Systems*, June 1992.

[61] C. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *Proc. 14th Int. Joint Conf. AI*, pages 1686–1693, 1995.

[62] J. Koehler. Planning under resource constraints. In *Proc. 15th Eur. Conf. AI*, 1998.

[63] J. Koehler. Solving complex planning tasks through extraction of subproblems. In *Proc. 4th Intl. Conf. AI Planning Systems*, Pittsburgh, PA, June 1998.

[64] J. Koehler, B. Nebel, J. Hoffmann, and Y Dimopoulos. Extending planning graphs to an adl subset. In *Proc. 4th European Conference on Planning*, pages 273–285, Sept 1997.

[65] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. TR 88, Institute for Computer Science, University of Freiburg, 1997. See `http://www.informatik.uni-freiburg.de/~ koehler/ipp.html`.

[66] G. Kondrack and van Beek. P. A theoretical evaluation of selected backtracking algorithms. *J. Artificial Intelligence*, 89:365–387, 1997.

[67] R. Korf. Planning as search: A quantitative approach. *J. Artificial Intelligence*, 33(1):65–88, September 1987.

[68] N. Kushmerick, S. Hanks, and D. Weld. An Algorithm for Probabilistic Least-Commitment Planning. In *Proc. 12th Nat. Conf. AI*, 1994.

[69] N. Kushmerick, S. Hanks, and D. Weld. An Algorithm for Probabilistic Planning. *J. Artificial Intelligence*, 76:239–286, 1995.

[70] A. L. Lansky. Localized planning with action-based constraints. *J. Artificial Intelligence*, 98(1-2):49–136, 1998.

[71] C. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. 15th Int. Joint Conf. AI*, August 1997.

[72] M. Littman. Probabilistic propositional planning: Representations and complexity. In *Proc. 14th Nat. Conf. AI*, pages 748–754, 1997.

[73] S. M. Majercik and M. L. Littman. MAXPLAN: a new approach to probabilistic planning. In *Proc. 4th Intl. Conf. AI Planning Systems*, pages 86–93, June 1998.

[74] S. M. Majercik and M. L. Littman. Using caching to solve larger probabilistic planning problems. In *Proc. 15th Nat. Conf. AI*, pages 954–960, July 1998.

[75] A. D. Mali and S. Kambhampati. Encoding HTN planning in propositional logic. In *Proc. 4th Intl. Conf. AI Planning Systems*, June 1998.

[76] D. McAllester. An outlook on truth maintenance. Ai memo 551, MIT AI LAB, 1980.

[77] D. McAllester. Truth maintenance. In *Proc. 8th Nat. Conf. AI*, pages 1109–1116, 1990.

[78] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. 9th Nat. Conf. AI*, pages 634–639, July 1991.

[79] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proc. 14th Nat. Conf. AI*, pages 321–326, Providence, Rhode Island, July 1997.

[80] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[81] D. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proc. 3rd Intl. Conf. AI Planning Systems*, pages 142–149, May 1996.

[82] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem-solving perspective. *J. Artificial Intelligence*, 40(1–3):63–118, 1989.

[83] P. Nayak and B. Williams. Fast context switching in real-time propositional reasoning. In *Proc. 14th Nat. Conf. AI*, Providence, R.I., July 1997.

[84] B. Nebel, Y. Dimopoulos, and J. Koehler. Ignoring irrelevant facts and operators in plan generation. In *Proc. 4th European Conference on Planning*, Sept 1997.

[85] F. Okushi. Parallel cooperative propositional theorem proving. (Submitted for publication; preliminary version presented at the Fifth International Symposium on Artificial Intelligence and Mathematics: `http://rutcor.rutgers.edu/~amai`), 1998.

[86] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[87] B. Pell, D. Bernard, S. Chien, E. Gat, N. Muscettola, P. Nayak, M. Wagner, and B. Williams. An autonomous spacecraft agent prototype. In *Proc. First Intl. Conf. Autonomous Agents*, pages 253–261, 1997.

[88] J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. Principles of Knowledge Representation and Reasoning*, pages 103–114, October 1992. See also `http://www.cs.washington.edu/research/projects/ai/www/ucpop.html`.

[89] M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proc. 1st Intl. Conf. AI Planning Systems*, pages 189–197, June 1992.

[90] M. E. Pollack, D. Joslin, and M. Paolucci. Flaw selection strategies for partial-order planning. *J. Artificial Intelligence Research*, 6:223–262, 1997.

[91] J. T. Rintanen. A planning algorithm not based on directional search. In *Proc. 6th Int. Conf. Principles of Knowledge Representation and Reasoning*, June 1998.

[92] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. 12th Nat. Conf. AI*, pages 337–343, July 1994.

[93] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:521–532, 1996.

[94] B. Selman, H. Levesque, and D.. Mitchell. A new method for solving hard satisfiability problems. In *Proc. 10th Nat. Conf. AI*, pages 440–446, July 1994.

[95] Bart Selman, Henry Kautz, and David McAllester. Computational challenges in propositional reasoning and search. In *Proc. 15th Int. Joint Conf. AI*, 1997.

[96] D. Smith. Controlling backward inference. *J. Artificial Intelligence*, 39:145–208, 1989.

[97] D. Smith and M. Peot. Postponing threats in partial-order planning. In *Proc. 11th Nat. Conf. AI*, pages 500–506, June 1993.

[98] D. Smith and M. Peot. Suspending recurison in causal link planning. In *Proc. 3rd Intl. Conf. AI Planning Systems*, 1996.

[99] D. Smith and D. Weld. Conformant Graphplan. In *Proc. 15th Nat. Conf. AI*, July 1998.

[100] D. Smith and D. Weld. Temporal graphplan. Technical report, Univ. of Washington, Dept. of Computer Science and Engineering, 1998.

[101] R. Srinivasan and A. Howe. Comparison of methods for improving search efficiency in a partial-order planner. In *Proc. 14th Int. Joint Conf. AI*, pages 1620–1626, 1995.

[102] M. Stefik. Planning with constraints (MOLGEN: Part 1). *J. Artificial Intelligence*, 14(2):111–139, 1981.

[103] J. Tenenberg. Abstraction in planning. Ph.d. thesis, University of Rochester, Department of Computer Science, May 1988.

[104] A. Van Gelder and F. Okushi. A propositional theorem prover to solve planning and other problems. (Submitted for publication; preliminary version presented at the Fifth International Symposium on Artificial Intelligence and Mathematics: `http://rutcor.rutgers.edu/~amai`), 1998.

[105] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996.

[106] Manuela Veloso. Flexible strategy learning: Analogical replay of problem solving episodes. In *Proc. 12th Nat. Conf. AI*, pages 595–600, July 1994.

[107] D. Weld. An introduction to least-commitment planning. *AI Magazine*, pages 27–61, Winter 1994. Available at `ftp://ftp.cs.washington.-edu/pub/ai/`.

[108] D. Weld and J. de Kleer, editors. *Readings in Qualitative Reasoning about Physical Systems.* Morgan Kaufmann, San Mateo, CA, August 1989.

[109] D. S. Weld, C. R. Anderson, and D. E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *Proc. 15th Nat. Conf. AI*, pages 897–904, July 1998.

[110] B. C. Williams and P. P. Nayak. A model-based approach to reactive self-configuring systems. In *Proc. 13th Nat. Conf. AI*, Portland, OR, August 1996.

[111] B. C. Williams and P. P. Nayak. A reactive planner for a model-based execution. In *Proc. 15th Int. Joint Conf. AI*, Nagoya, Japan, August 1997.

[112] Q. Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6(1):12–24, February 1990.

[113] Q. Yang and A. Chan. Delaying variable binding committments in planning. In *Proc. 2nd Intl. Conf. AI Planning Systems*, pages 182–187, June 1994.

[114] Q. Yang and J. Tenenberg. ABTWEAK: Abstracting a nonlinear, least-commitment planner. In *Proc. 8th Nat. Conf. AI*, pages 204–209, August 1990.