CS 227 Programming Assignment 4: Planning with Identidem

Todd Sullivan
todd.sullivan@cs.stanford.edu

Harry Robertson
harry.robertson@gmail.com

1 Introduction

Planning algorithms are an extremely active area of AI research. They have been applied to solving a variety of complex real-world problems, from managing cargo at ports to running Mars Rovers. On the academic side, their development has been motivated by high-profile planning competitions, notably IPC.

Given the complexity of the problem of building a general planner, there is a plethora of approaches. Indeed, there are both backward and forward-chaining methods, and success has been had by more or less directly converting the problem to SAT problems or CSPs. In this paper we consider approaches which utilize the "planning graph" developed by the GraphPlan algorithm. FF is one of the most successful planners of late; it performs forward-chaining search oriented by a heuristic based on a "relaxed" version of the planning graph (in which some constraints are not considered).

In this paper we present our implementation of Identidem, an extension to FF. Identidem utilizes localized restarts to cut down on the time FF spends searching for ways to escape from local minima. It does this by performing restarts to the start point of plateaux / saddle points after judiciously chosen time-out intervals, and also global restarts if necessary.

FF, like all the other planners studied here, uses for input the planning domain description language PDDL. In this paper we present a new planning domain entitled SuperBarman, complete with a set of sample problem files, written in PDDL 2.1. We evaluate the performance of our Indentidem implementation and a selection of publicly available planners (FF, SAPA and LPG) on this planning domain.

2 Identidem Algorithm

Identidem is a local search algorithm for forward-chain planning that builds upon FF. The algorithm uses FF's relaxed planning graph heuristic as well as YAHSP's lookahead algorithm for adding an additional child to each state, although we do not implement the lookahead algorithm in this project. The key extension of FF that Identidem implements is the use of random restarts.

Identidem replaces FF's enforced hill climbing search, which uses breadth first search to escape local minima, with a probing local search. The algorithm attempts a certain number of probes with increasing depth bounds. After failing to find a better state given the current depth bound, the depth bound is doubled. This process continues for a predefined number of iterations. The number of iterations, the number of probes with a given depth bound, and the initial depth bound are all parameters. We used the same parameter values as in [1], which has 5 iterations, 60 probes at a given depth, and an initial depth bound of 10.

Upon failure to escape the local minimum after the specified number of iterations, the algorithm restarts from the initial state. Identidem also introduces an additional method for triggering a restart called fail-bounded restarts. Fail-bounded restarts are meant to restart the search when slow but steady progress

is being made (in the hopes that a location in the search space with faster progress exists). An additional parameter, called the fail bound, indicates the amount of times a probe can fail to escape local minima before forcing a restart. Every time that a probe fails to find a state with a strictly better heuristic value a fail counter is incremented. Whenever the fail counter reaches the fail bound, a restart is forced. We use the same fail bound sequence as in [1], which starts at 32 and doubles every three restarts. Therefore the fail bound sequence is 32, 32, 32, 64, 64, 64, etc.

Identidem also improves upon FF by incorporating neighborhood sampling with roulette wheel selection. When at a given state, the algorithm takes all helpful actions as identified by FF and places each into one bucket in a neighborhood bucket grid. The neighborhood bucket grid is a two-dimensional array of buckets where the row indicates the action type and the column indicates the number of parameters that the action has in common with the parent action (with the parameters being preconditions, adds, and deletes).

After placing each potential child action in a bucket, the algorithm creates a neighborhood whose size is pre-defined by the neighborhood size parameter. Our implementation uses the same neighborhood size parameter as [1], which is 3. To construct the neighborhood, one continually randomly picks a nonempty bucket and then randomly selects one of the actions in the bucket to include in the neighborhood.

Identidem also includes an additional twist in that for the second half of the probe iterations non-helpful actions are also included in the neighborhood sampling process. This allows the planner to solve problems in which incorporating non-helpful actions is essential to solving the problem. We implemented this feature as a command-line option so that we could evaluate its effect on performance.

Once a neighborhood is created, each action n in the neighborhood is given a roulette wheel segment value as follows:

$$W_n = \left(\frac{1}{h_n}\right)^{\beta}$$

where h_n is the heuristic value of action n and β is the heuristic bias parameter. We follow [1] in initially setting β to 1.5 and linearly decreasing its value to 0.5 as each probe within a given depth bound is attempted. The probability of an action in the neighborhood being chosen is determined by normalizing across all heuristic bias parameters.

3 Indentidem Optimizations

Caching and other optimizations are essential for creating usable planners. Like our previous projects, we developed our algorithms with efficiency in mind. Our optimizations include managing our own memory (including intelligent allocation), quick lookup structures for action parameters, caching of "parameters in common" values, and efficient neighborhood creation.

3.1 Memory Management

Unfortunately, since we extended FF, which is written in ANSI C, we did not continue our C++ STL Vector saga from previous projects. Staying in the spirit of FF's coding practices each function manages its own memory structures. These memory structures are allocated during the first call to the function. With the exception of the neighborhood buckets, we pre-allocate all memory in accordance with its maximum possible need during the first call to each function. Thus all stacks and other structures never need to be resized because they are always allocated the maximum amount of memory that they will need.

We do not pre-allocate all neighborhood buckets because the neighborhood bucket grid is a two-dimensional array of buckets that has a row for each action type and columns from zero to the largest number of parameters of

any given action. Each bucket within this grid needs to potentially be able to contain all actions except the action specified by the bucket's row coordinate in the grid. Indeed on many problems this massive structure requires too much memory to allocate outright during the first call to the function.

To mitigate this issue, we made two modifications. First, we turned the grid into a ragged two-dimensional array (rows have a variable number of columns). Since the column indices indicate the number of parameters the action (indicated by the row) has in common with the previous action applied, we only need to have as many columns as the action type has parameters (plus one for the zero case).

Our second modification is that we only allocate a bucket's memory the first time that we need to place an action in the bucket. During the first call to the function we allocate the grid of bucket pointers and set all of the pointers to null. Before inserting an action into a bucket, we check to see if the bucket has been allocated. If it has not been allocated, then we allocate the memory. This simple modification has a large impact on memory usage, as most buckets in the grid are actually never needed.

3.2 Action Parameters

Identidem involves calculating the number of parameters each child action has in common with the previous action applied. FF's data structures are not conducive for this task. FF splits an action's parameters (its preconditions, adds, and deletes) into three separate arrays with each array containing integers starting at zero. These arrays are not sorted in any way. Thus calculating the number of parameters in common between two actions is an $O(n^2)$ operation, where n is the number of parameters in an action.

We reduce this complexity to an O(n) operation by preprocessing the actions before entering the search loop. For each action we create an array of integers that contains all parameters. We change the values of adds and deletes by adding the number of preconditions possible to each add value and the number of preconditions possible plus the number of add values possible to each delete value. These numbers are available from FF's own preprocessing and do not require any additional work to calculate.

After constructing each array of parameters, we use quicksort to sort each array. Since the arrays are sorted, we are able to calculate the number of parameters in common between two actions in linear time. Of course, we are pushing the worst case $O(n^2)$ complexity onto the sorting procedure, but this sort is performed once for each action whereas the $O(n^2)$ parameters in common calculation would be performed significantly more times during the search.

3.3 Parameters in Common Caching

For every child action that we consider for our neighborhood we must calculate the number of parameters the child action has in common with the parent action. This task is clearly an easy location for optimization, as the search process will undoubtedly run into the same parent-child pairs multiple times. To solve this issue, we cache all parameters in common calculations. Thus after the first time an action pair occurs, we store the resulting value in an array. On subsequent occurrences of the same action pair we simply grab the pre-calculated value from the array.

To investigate the success of this caching procedure, we recorded the amount of cache hits while running our planner on all of the IPC3 problems. As our results will show, this caching is effective. Even in problems where no restarts are required, such as the Rovers problems, the planner still experiences an average of 46 cache hits. On more difficult problems, such as DriverLog where our planner restarts an average of 1,109 times, the cache hit count is more than one million on average.

3.4 Efficient Neighborhood Creation

The Identidem algorithm's neighborhood creation involves randomly sampling so many nonempty neighborhood buckets and then randomly choosing one action in the bucket and including it in the neighborhood. As discussed earlier, the neighborhood grid can be enormous for certain problem instances while most buckets are empty at any given point in time. Repeatedly randomly picking a bucket in the grid until one finds a nonempty bucket is clearly inefficient.

To solve this problem we maintain a stack of bucket pointers that contain all of the nonempty buckets. When sampling for the neighborhood, we can simply generate a random integer and use the modulus operator with the integer and the number of nonempty buckets to select a nonempty bucket. We add buckets to the nonempty bucket stack when inserting an action into the bucket causes the bucket's size to increase to one.

The nonempty bucket stack is simply an array of pointers with a size field indicating how many items are currently in the stack. This makes adding new items easy as we simply place the new item at the index indicated by the current stack size and then increment the stack size. Removing items is also delightfully simple. All we do is place the last bucket pointer in the array into the position of the pointer that we want to remove and then we decrement the stack size.

4 Identidem Experimental Method

Our experimental method is almost identical to our previous projects. We conducted all of our experiments on the Pod cluster. The Pod cluster contains Dell Precision 390s, each with a 2.4 GHz Core 2 Duo and 2 GB of RAM running Ubuntu 7.04. All results in Section 5 are from running each planner on the IPC3 planning problems using 5 different seeds for the random number generator. We used the seeds 1 through 5. We enforced a timeout of 30 minutes for each problem.

4.1 Naming Conventions

In all results the first segment of characters indicates the problem type while the second indicates the problem number within the problem type. We present results for our standard algorithm that does not use non-helpful actions as well as results for including non-helpful actions as described in Section 2. The results when using non-helpful actions always have an "N" appended to the name. For example, "Depots-07-N" indicates the results for the seventh problem of the Depots problem type with our algorithm using non-helpful actions. When presenting aggregate results, we simply remove the number portion. Thus "ZenoTravel" indicates the aggregate results for the ZenoTravel problem type with our algorithm not using non-helpful actions.

4.2 Our Performance Metrics

We track three metrics: Restart Count, Cache Hits, and Total Time. As the name suggests, Restart Count is the number of times that the algorithm restarts. Cache Hits is the number of cache hits for our parameters in common caching described in Section 3.3. Total Time is the time as reported by FF's timing functionality. All times are in seconds and include all computation time, including parsing the file, all preprocessing, and search.

5 Identidem Results

We include all our results for Identidem in the Annex. By way of introduction, our average success rates for each problem class, without non-helpful actions, are the following:

Problem Type	Attempt Count	Success Count	Success Rate
Depots	110	104	0.945
DriverLog	100	67	0.67
FreeCell	100	86	0.86
Rovers	100	100	1
Satellite	100	100	1
SuperBarman	25	23	0.92
ZenoTravel	100	100	1

Of course, these averages conceal the fact that the problem classes contain problems of greatly varying difficulty. However, they do illustrate that not all of the problem classes are created equal: the DriverLog problems in particular are very difficult for Identidem, whereas the Rovers problems are all solved. A look at the detailed results in the annex confirms this difficulty disparity (none of the Rovers problems take more than one second on average to solve, whereas a third of the DriverLog problems times out given the 30 minute timeout).

The main factor whose influence on performance we wished to evaluate is the use of non-helpful actions. Looking at the average success rates for a selection of problem classes shows mixed results:

Problem Type	Attempt Count	Success Count	Success Rate
Depots	110	104	**0.945**
DepotsN	110	92	**0.836**
DriverLog	100	67	**0.67**
DriverLogN	100	84	**0.84**
FreeCell	100	86	**0.86**
FreeCellN	100	93	**0.93**

For the DriverLog and FreeCell problem classes, use of non-helpful actions significantly improves results (increasing average success rate by around 20% for DriverLog!). However, their use actually decreases success rate on Depots. We hypothesize that this is due to inherent characteristics of the problem categories. Looking at the breakdown of results on a per-problem basis (see Annex) somewhat confirms this; indeed, for Depots problems for example, non-helpful actions does consistently worse. However, in some cases the results seem chaotic with no general trend. This is the case for DriverLog:

Problem	Restart Count	Cache Hits	Total Time
DriverLog-17	55	2292373.2	**600.1**
DriverLog-17-N	13.2	417584.4	**164.1**
DriverLog-18	19	468734	**226.3**
DriverLog-18-N	11.8	97039.6	**433.9**

On problem 17 non-helpful actions does considerably better, improving runtime by a factor of 5, but on problem 18 it is half as good as without non-helpful actions, even though we are within the same problem class. We suspect this is simply an artifact of the high variance of the Identidem algorithm, which we will examine further in the SuperBarman results analysis later in the paper. We believe that for this reason the average results over each problem class actually convey performance trends better than looking at individual problems.

Another factor whose importance is shown in our results is the role of our parameter caching mechanism. On simple problems the number of cache hits is negligible, but for the hardest problems in each set the numbers become significant. Here are the average results for FreeCell:

Problem	Restart Count	Cache Hits	Total Time
FreeCell-14-N	0	**199.6**	1.06
FreeCell-15-N	4.2	**13275**	28.60
FreeCell-16-N	3.6	**6877.4**	16.26
FreeCell-17-N	1.2	**1311.2**	6.03
FreeCell-18-N	19.2	**415158.2**	1244.61
FreeCell-19-N	18.8	**85729.8**	1353.58
FreeCell-20-N	18.6	**155572.8**	1532.97

Cache hit numbers are on the order of one million for some cases, which translates to a significant performance gain. However, the number of cache hits is not linear relative to runtime: FreeCell-18 has slightly shorter runtime than FreeCell-19, but has an order of magnitude more cache hits. These variations may however simply be due to the high variance of our algorithms.

6 SuperBarman Planning Domain

6.1 Problem Description

In our SuperBarman class of problems, we present a problem faced by everyone at some point in their life: how to run a bar efficiently. The planner must come up with a sequence of actions for a barman which satisfies all of the bar's patrons. The barman has a set of bottles of beverages on the rack behind him, and a set of (initially empty) glasses on the bar counter in front of him. He can hold as many bottles at a time as he has hands, and can pour liquid from any bottle he is holding into any of the glasses on the counter. A cocktail is a mix of three liquids, added in a specified ordering to the glass.[1] Once a cocktail is ready (I.e. it contains the mix of liquids that a certain customer wants), he can give it to the customer. More formally, the specification of the problem class is the following:

- Types:
  - liquid (which represents a specific alcohol, juice or accompaniment)
  - bottle (there is one bottle on the rack for each kind of liquid)
  - glass
  - customer (we assume without loss of generality that each customer wants one specific drink)
  - hand (our protagonist, like any self-respecting barman, is ambidextrous; unlike most barmen, he can have as many hands as we wish)
- Predicates:
  - bottleContains ?b - bottle ?l – liquid : "The bottle *b* contains the liquid *l*"
  - onRack ?b – bottle : "The bottle *b* is currently on the rack"
  - onBar ?g – glass : "The glass *g* is currently on the bar counter"
  - pouredInLast ?l - liquid ?g – glass : "*l* was the last liquid poured into glass *g*"
  - pouredInAfter ?lafter ?lbefore - liquid ?g – glass : "*lafter* was poured into the glass *g* just after the liquid *lbefore*"
  - holding ?b - bottle ?h – hand : "The barman is holding the bottle *b* in his hand *h*"
  - handFree ?h – hand : "The barman's hand *h* is not holding anything"
  - wants ?c - customer ?l1 ?l2 ?l3 ?l4 – liquid : "Customer *c* wants a cocktail made by putting into a glass first *l4*, then *l3*, then *l2*, then *l1*"
  - satisfied ?c – customer : "Customer *c* has been given his drink and is therefore satisfied"
- Actions:
  - pickup : "The barman picks up bottle *b* from the rack with his hand *h*"
  - putdown : "The barman puts the bottle *b*, which he was holding in his hand *h* back on the rack"
  - pour : "The barman pours the liquid *lnew* from the bottle *b*, which he is holding in his hand *h*, into the glass *g*, into which he had previously poured the liquid *lbefore*"
  - give : "The barman gives customer *c* the glass *g* which contains the cocktail made up of *l4*, *l3*, *l2*, *l1*, as specified by the customer"[2]

[1] In our original problem class, called simply Barman, the order of addition of the liquids to the glass did not matter. However, introducing the required ordering both makes the problem more demanding (as otherwise each bottle only needs to be picked up once), and more realistic.

[2] Note that giving a drink to a customer does not require the use of a hand, as the barman can simply visually indicate the glass on the counter to the customer.

All five instances of the problem have some common points in their objects, initial state, and goals:

- Objects:
  - There is one "fake" liquid, called emptyL, which all glasses initially "contain"
  - There are as many bottles as there are liquids (with the exception of emptyL)
  - There are as many glasses as there are customers
- Initial state:
  - Each liquid is contained by one bottle (with the exception of emptyL)
  - All bottles are on the rack
  - All glasses are on the bar
  - All glasses are empty, i.e. such that: (pouredInLast emptyL glass_i)
  - All hands are free
  - Each customer wants one specific cocktail, and the variable l4 is always emptyL.
- Goals:
  - The goals are always that every customer be satisfied

The different problems vary in the number of liquids/bottles, the number of glasses/customers, the number of hands the barman has, and the composition of the cocktails wanted by the customers. To summarize the five problems:

- SuperBarman-1 is the most basic form, with just one customer and one hand.
- SuperBarman-2 has two customers wanting the same cocktail, and so tests whether the planner can avoid picking up and putting down each bottle multiple times.
- SuperBarman-3 has three customers, who want cocktails designed such that each bottle only has to be picked up once, if the planner finds an appropriate sequencing of drinks (I.e. ice, then martini, then vodka, then orange juice).
- SuperBarman-4 has 6 liquids, 6 glasses, 2 hands, and randomly generated cocktails (the cocktail specifications were generated from the digits of Pi).
- SuperBarman-5 has 10 liquids, 10 glasses, 4 hands, and randomly generated cocktails.

6.2 Motivation

We designed the SuperBarman problem class such that it is easy to find a primitive solution, but hard to find a solution with a short plan length. Indeed, there is an obvious general solution to these problems: just satisfy the first customer, then the second customer, etc.; for a given customer, just make the cocktail by picking up the bottle for the first ingredient, pour it in the glass, put the bottle down, and continue for the two other ingredients. This obviously provides a plan which is linear in the number of customers, but this primitive method clearly leads to inefficient plans, as with a more sophisticated approach one can decrease the number of pick-ups and put-downs of bottles required to make all of the cocktails, to decrease plan length. The presence of multiple hands further complicates things.

6.3 Planner Performance

For comparison purposes, we evaluated the performance of four different planners on our SuperBarman problem set: FF, SAPA, LPG, and our implementation of Indentidem. SAPA is a Forward Chaining Heuristic Metric Temporal Planner. Like other algorithms considered here, it uses a planning graph-based approach, and uses heuristics which are designed to take into account both cost and makespan (see [2]).

LPG is a popular planner which performs a WalkSAT-inspired local search using heuristics based on a parameterized objective function (see [3]). In the following results we ran the publicly available implementation of LPG in two different modes: "speed", which minimizes runtime, and "quality", which optimizes

plan quality (I.e. minimizes the cost of the plan) at the cost of longer runtimes. We refer to these two modes as LPG-speed and LPG-quality, respectively.

For the purposes of comparison, we graphed runtime and the final plan's number of actions for each planner. As LPG generated plans with parallel actions, we also graphed total plan length, but this is not a fair point of comparison as our other planners do not do this. All results were obtained on an Athlon X2 system with 2GB of RAM running Kubuntu. We ran Identidem with Algorithm 3 and non-helpful actions.

	P1	P2	P3	P4	P5
FF	0	0	0.01	0.18	2.37
SAPA	0.05	*	3.04	*	*
LPG-quality	0.26	0.26	0.26	10.26	177
LPG-speed	0	0.01	0.17	0.19	7.33
Identidem (avg)	0	0	0.06	102	*

Runtime results (seconds)

	P1	P2	P3	P4	P5
FF	9	19	26	46	76
SAPA	9	*	20	*	*
LPG-quality	9	13	23	38	58
LPG-speed	9	15	23	65	72
Identidem (avg)	14	18.8	34.4	76.2	*

Plan size (number of actions)

Unfortunately SAPA gives very poor performance on the SuperBarman problems, failing on problems 2, 4 and 5 (the *'s indicate timeouts or failure to finish due to out-of-memory errors, which is the case for SAPA). It is unclear why it fails so badly, but the fact that it generates a valid plan for problems 1 and 3 in a reasonable amount of time seems to indicate that it is being used properly. It is notable however, that SAPA provides the shortest plan length on problem 3 (20 actions as opposed to 23 for LPG). Problem 3 was designed to test the ability of the planner to properly sequence the bottle pick-ups in order to minimize plan length, and SAPA achieves this when it works.

FF is by far the best performer in terms of runtime. However, it also generates longer plans than LPG (or SAPA on problem 3). This is surprising, as the enforced hill-climbing method employed by FF is supposed to encourage shorter solutions according to the proponents of FF (see the Results section of [4]). It would therefore seem that the version of FF considered here is minimizing run-time at the cost of plan length. The excellent runtime results are therefore somewhat deceptive: as we have already stated, each of the SuperBarman problems can be resolved very simply with basic methods; the real work lies in finding a short plan length. We would tend to conclude that the SuperBarman problem class is not well adapted to evaluating FF in this form.

The most satisfactory performance with respect to plan length is given by LPG in "quality" mode. In all but one case (problem 3), LPG-quality yields the shortest total number of actions. If one considers plan duration, with parallel actions, LPG does even better:

	P1	P2	P3	P4	P5
Plan size	9	13	23	38	58
Plan duration	9	9	20	**18**	**18**

Plan size and duration for LPG-quality

plan duration is half the number of actions for problem 4, and a third for problem 5. We deduce that LPG's multi-parameter objective function is well adapted to the SuperBarman, allowing LPG to take advantage of natural parallelizability of the actions (indeed, for each SuperBarman problem, plan duration can be divided roughly by the number of hands if actions are parallelized).

However, runtime performance of LPG-quality does blow up on the harder problems; it seems that the algorithm does not scale well on this class of problem beyond 10 liquids and 10 glasses. Furthermore, the short plan lengths come at a heavy price: LPG-quality is roughly two orders of magnitude slower than FF for all of the problems. This cost is however somewhat illusory, as the main point of SuperBarman was always to evaluate plan length rather than runtime. LPG-speed emerges as an awkward compromise between FF and LPG-quality. On problem 5, its runtime is 3 times

that of FF, for roughly the same plan length. On problem 4, its plan length is actually worse than FF's.

The performance of our own implementation of Identidem is extremely disappointing on the SuperBarman class of problems. On the easy problems (1 through 3), there is little discernible difference with regular FF for average runtime. On problem 4 however, runtime becomes extremely poor, around 100 seconds on average, and on problem 5 all but one seeds do not finish. These average figures do not tell the whole story however; examination of the results for individual seeds reveals an enormous variance problem:

	P1	P2	P3	P4	P5
r1	0	0	0.01	0.69	*
r2	0	0	0.04	217	**36**
r3	0	0	0	226	1325
r4	0	0	0.24	2.78	*
r5	0	0	0.01	64	*

Runtime results for Identidem with five different seeds (seconds)

There is terrific variance in the runtime results: runtime varies by three orders of magnitude for problem 4, and the one of the two seeds which does finish on problem 5 takes only 36 seconds (which is considerably better than LPG-quality for example). Variance is somewhat to be expected given that the choice of seed influences paths Identidem's probes take, but it is shocking that it should be this strong since one of the points of restarts is to reduce variance.

The plans generated by Identidem are poor in terms of plan length compared to the other planners, which is also disappointing. However, it is interesting to note that if one considers each time the minimum result over the five seeds we considered, then the results look quite different:

	P1	P2	P3	P4	P5
LPG-quality	0.26	0.26	0.26	10.26	177
LPG-speed	0	0.01	0.17	0.19	7.33
Identidem (min)	0	0	0	0.69	36

Runtime results (seconds)

From this point of view Identidem appears competitive with LPG, with runtime in between LPG-speed and LPG-quality. This indicates that a promising approach would be to use a random restart mechanism, restarting Identidem, restarting the algorithm with a new seed after a certain (growing) time window is filled. Alternatively one could run several threads of Identidem with different seeds on different processor cores, and take the first plan to finish.

7 References

[1] Andrew Coles, Maria Fox, and Amanda Smith (2007). A New Local- Search Algorithm for Forward-Chaining Planning, ICAPS 2007.

[2] M. B. Do and S. Kambhampati. Sapa: A Scalable Multi-objective Heuristic Metric Temporal Planner. Journal of AI Research, 20:155--194, 2003.

[3] Gerevini, A. and Serina, I. 2003. Planning as Propositional CSP: From Walksat to Local Search Techniques for Action Graphs. Constraints 8, 4 (Oct. 2003), 389-413.

[4] Hoffmann, J.. FF: The Fast-Forward Planning System. AI Magazine 22[4], 57-62. 2001. AAAI Press.