An adaptive technique for improving the efficiency of planners

J.A.N.A. Trudel

Randy Goebel

Logic Programming and Artificial Intelligence Group
Computer Science Department
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

ABSTRACT

We describe a planning program called "Madame," which is based on D.H.D. Warren's WARPLAN modified to use Kowalski's representation for dynamic worlds. The modified WARPLAN uses a network data structure, called the "spider," to store goal states generated during the planners use. The spider is retained over multiple invocations of Madame, and is used to provide possible subplans for subsequent planning requests.

Experimental results are presented which show that the spider gives improvement in terms of the time required to produce plans for a collection of randomly generated queries. We also study the usefulness of the spider as it increases in size.

We discuss the spider, the implementation that uses it, and various heuristic strategies for retaining and using subplans stored in the spider.

June 30, 1986

An adaptive technique for improving the efficiency of planners

J.A.N.A. Trudel

Randy Goebel

Logic Programming and Artificial Intelligence Group
Computer Science Department
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

Introduction

WARPLAN (Warren 1974) is a goal-directed planner implemented in Prolog. It uses a STRIPS-like (Nilsson 1980) representation of state change operations. For example, the operation of removing a file in a single-user UNIX environment might be represented as:

```
add( not-exist( File ), rm( File ) );
del( exist( File ), rm( File ) );
can( rm( File ), exist( File ) & write-permission( File ) );
```

This example uses the original syntax of WARPLAN; the binary relations "add," "del," and "can" correspond to the add, delete, and precondition lists of STRIPS. If given an initial state description and a collection of operations like the above, WARPLAN will accept a partial description of a hypothetical state and attempt to generate a sequence of operations that will transform the initial state into the hypothetical one. For example, given an initial state

```
exist(file1), exist(file2), ..., write-permission(file1), ...
```

that describes the properties of a set of files, one might like to generate a plan to achieve a state in which "file1" no longer existed. The above rule can be used to deduce that such a state follows by applying the operation "rm(file1)" in the initial state.

This approach to automatic planning has provided a basis for several elaborations (e.g., see Nilsson 1980), and has also demonstrated the necessity for more highly structured problem-solving programs (e.g., Sacerdoti 1979).

Selecting a planning strategy for ISH

Madame is intended for use as the planning component of the University of Waterloo Intelligent Shell (ISH). ISH seeks to provide a replacement command interpreter for UNIX. It has been designed to augment the normal notion of command shell with an ability to answer questions about how to accomplish a desired

state (cf. Wilensky et al. 1984). Such an ability requires a planner that can determine what sequence of applicable actions will accomplish the desired state.

The primary motivation for ISH is to investigate the description of realistic dynamic worlds using logic. For example, although ISH has a "natural" language front-end (Malito 1984), it is not nearly as sophisticated as UC. However, it's axiomatization of UNIX file commands is quite sophisticated.

For several reasons not reported here (see Pakalns 1984), the above STRIPS-like representation is too simple for the UNIX domain. Kowalski's representation for dynamic worlds (Kowalski 1979) is more general, and was adopted for ISH. This representation uses a meta predicate "holds" that is a binary relation on assertions and states. For example, the assertion

holds(exist(file1), state1)

represents the fact that "file1" exists in "state1." The description of the above remove operation becomes

```
possible( result( rm( File ), State ) ) <-
        possible( State )
        holds( exist( File ), State )
        holds( write-permission( File ), State );
```

which can be read as "If one has write permission for a file and that file exists, then the 'rm' action can be applied." One advantage of such a representation include a succinct expression of frame axioms (via assertions related to the "possible" relation) which are implicit in the STRIPS-like strategy (see Kowalski 1979, ch. 6).

The WARPLAN program modified to use Kowalski's representation provides a strong planning foundation but its naive use (e.g., to generate a plan by always beginning at the initial state) is prone to gross inefficiency. To combat this inefficiency, the spider data structure was implemented as a kind of "lemma repository." An improvement in the efficiency of plan generation can be had, at least in theory, by retaining the plans (named states) derived for previous requests and using them as building blocks in subsequent requests.

As the logical foundation shows, plans and state names are synonomous. One can view a plan as the result of a constructive proof of a query

holds(List-of-goals, Plan-to-achieve-state)

where "Plan-to-achieve-state" is an existential variable. The proof constructs a state in which the goals hold. Abstractly, assertions about the saved states are used as lemmas for complex derivations; in planning terms, plans are retained and an attempt is made to use retained plans as subplans of future plan requests.

One way of viewing the use of these saved plans is to adopt the naive view and consider initiating the plan search from an alternate "start state." In other words, one might try and begin the search for the appropriate sequence of actions from another state (other than the start state) that was assumed to be on the path of states to the desired state. If the planner uses a backward search, it could

terminate when one of the alternative start states is reached. Madame has been designed to work with multiple start states. It is a backward (or goal-directed) planner that takes a specification of what properties are desired and, based on these, selects a start state.

Acquisition and Retention of Multiple Start States

There are two major issues with having multiple start states: acquisition and retention. Acquisition begs the question of how to obtain these extra start states. Retention involves storing the extra start states in such a way that they are easily accessible. For computational efficiency, ease of addition and deletion of start states is also desirable.

When a plan is generated, it uniquely defines a goal state. This goal state may then be used as a state from which further planning can be pursued. That is, the plan used to reach one state might also serve as the initial portion of a larger plan. For example, in fig. 1, plan P1 transforms the start state into the goal state G1.

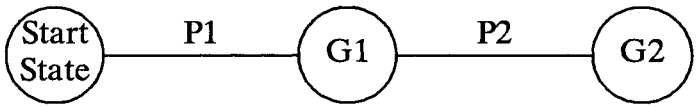


Figure 1 Using G1 as the Start State

If a plan between the start state and a new goal state G2 is required, it may be possible to use G1 as the start state. If G1 is used, P2 is a plan which transforms G1 into G2. A plan between the start state and G2 can now be generated by appending P2 to P1. In other words, G1 is a goal state that was subsequently used as a start state.

In general, the goal states from previously generated plans are retained as shown in the above example. The retention problem has been further divided into three sub-problems: storage, addition, and deletion of start states. Start states are stored in the spider data structure; implementation details are provided elsewhere (Trudel 1984). A spider is a directed tree with the original start state as its root. Arcs represent previously generated plans; nodes represent goal states associated with these plans. Each node in the spider including the root is called a *pseudo start state (p-s-s)*. For example, fig. 2a represents a spider with only one initial start state. Assuming that a goal request G1 produces a plan P1, the resulting spider is given in fig. 2b. An edge has been added that represents the actions in plan P1, and a p-s-s representing G1 is added. A subsequent goal request G2, and corresponding plan P2 might result in the spider shown in fig. 1. Now a plan between the original start state and G2 can be built, by appending the actions in P2

Figure 2a Initial Spider

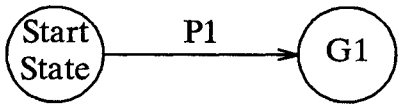


Figure 2b G1 is Added

to the actions in P1. This spider can now be saved for future use. Upon subsequent uses of Madame, the spider will contain three p-s-s's instead of only one.

Any p-s-s can be used as the start state. If a plan exists between one of the p-s-s's and a goal state, then a plan between the root and that goal state can be constructed. For example, assume a plan has been found which transforms p-s-s 4 into the Goal State; this is shown in fig. 3a. This Goal State with the newly generated plan is then added to the spider. A plan which transforms the Start State into the Goal State can now be easily constructed. It is built by appending the actions which are represented by the edges in the unique path between the Start State and the Goal State. This path is represented by the dashed lines in fig. 3b.

When Madame is given a goal state description to achieve, the spider is checked to see if the goal state corresponds to one of the p-s-s's. If so, no planning is required. The required plan is built by concatenating the sub-plans represented by the edges on the unique path between the start state and the p-s-s corresponding to the goal state. For example, assume we have a spider represented by fig. 4a and the goal state is equal to p-s-s 3. If a check was not made to see if the goal state is the same as one of the p-s-s's, then p-s-s 1 might be used for planning. An edge linking p-s-s 1 to p-s-s 3 is added to the spider; the resulting spider (which is not a tree) is shown in fig. 4b. Therefore, the initial check to see if the goal state equals one of the p-s-s's, guarantees that when a new p-s-s is added to the spider, it will have exactly one parent.

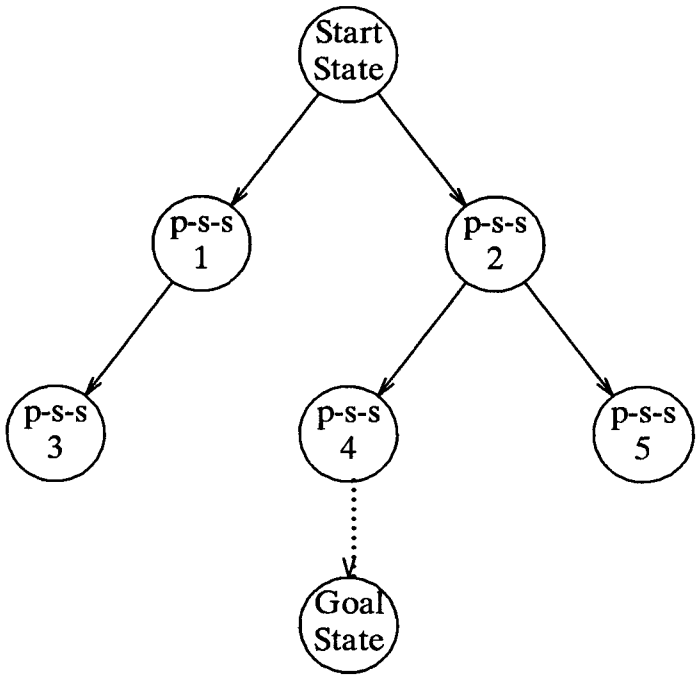


Figure 3a Plan between p-s-s 4 and the Goal State

Using the spider

Fig. 5 shows Madame's major components. The first component will find the p-s-s which is "closest" to the goal state described by the query. The method used to find the closest p-s-s is described below. The query and the closest p-s-s are then passed to the goal-directed planner. If an appropriate plan is found, the spider is updated with the new plan. After the spider has been updated, the plan is forwarded to the user. The user now has a plan which transforms the start state into the required goal state. All the plans returned to the user are relative to the start state.

The user now has two options: (1) he can pose another query to Madame, or (2) he can terminate Madame. If the user chooses termination, Madame will save the spider. Madame will automatically restore the spider upon the next invocation.

Selecting the closest p-s-s

One of the p-s-s's in the spider must be chosen as the start state for the planning component. Ideally, the p-s-s which is closest to the goal state should be chosen. Assume that the goal state is described by a list of goals LG. Madame measures closeness in terms of the number of goals in LG which do not hold in the

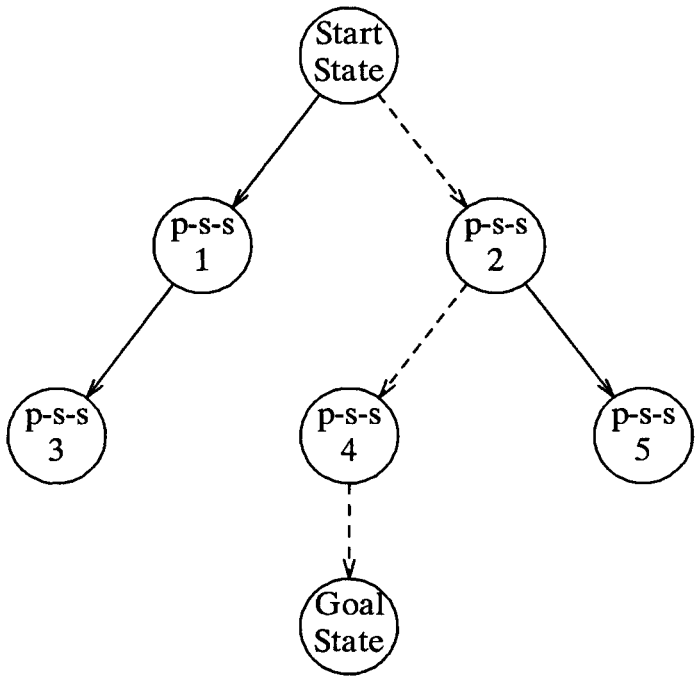


Figure 3b Plan between Start State and Goal State

p-s-s. The lower the number, the higher the "closeness." Note that the test of whether goals hold in a p-s-s uses only matching, not general deduction. This ensures that the selection is computable. In otherwords, the selection of the closest p-s-s depends on a STRIPS-like specification of worlds in which only those relations explicitly changed by actions are recorded.

To find the closest p-s-s, Madame must examine each p-s-s in the spider. A depth-first traversal of the spider is done. Any type of tree traversal algorithm is sufficient, as long as it is guaranteed to visit every node in the spider.

As each p-s-s in the spider is visited, its distance from the goal state is calculated. If a p-s-s is very close to the goal state (i.e. a short distance away), a large number of goals in LG will hold in the p-s-s. With the closest p-s-s, the planner will only have to work on the goals which do not hold in the p-s-s.

If the distance between the p-s-s and the goal state is zero (i.e. all the goals hold in the p-s-s), then the p-s-s is equal to the goal state. Traversal of the spider is terminated because it is impossible to find a closer p-s-s. If the p-s-s does not equal the goal state, the spider traversal is continued.

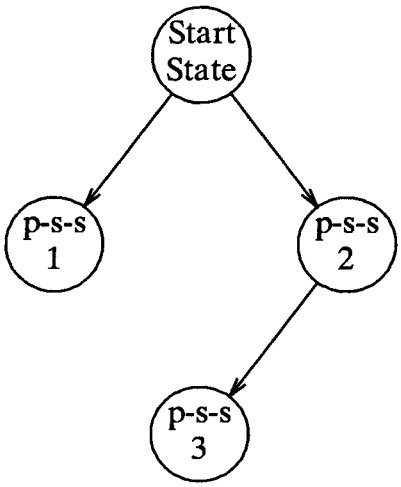


Figure 4a Initial Spider

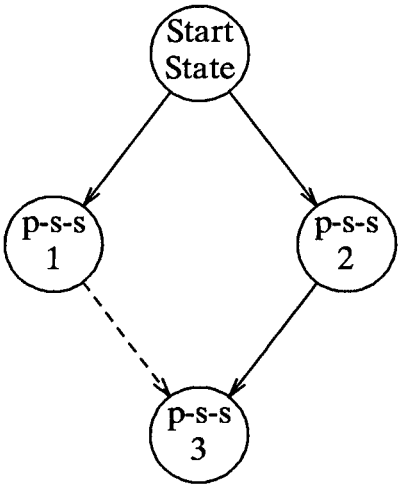


Figure 4b Illegal Spider

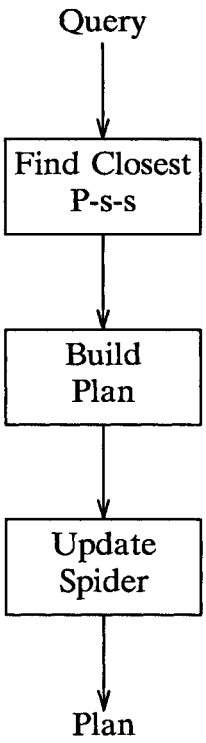


Figure 5 The Components of Madame

If a tie occurs between two p-s-s's, Madame will pick the one which is closest to the root (the start state). This will be the p-s-s with the least number of actions in the plan between itself and the root. This choice of p-s-s will hopefully minimize the number of steps in the eventual plan between the root and the goal state.

It is possible that, after traversing the spider, none of the goals in LG held in any of the p-s-s's. This results in a global tie, with all the p-s-s's equidistant from the goal state. The same rule that was used for breaking a tie between two p-s-s's is used; that is, the closest p-s-s to the root is chosen. This is a trivial problem, as the closest p-s-s is the root. In this case, the spider does not aid the planner.

Problems with finding the closest p-s-s

The above method does not always succeed in finding the closest p-s-s. For example, in the two blocks world states depicted in figs. 6a,b, the goal on(c a), places both p-s-s's at a distance of 1 from the goal state. The tie is broken by choosing the root because the distance between the root and itself is zero. Madame will choose the root as the closest p-s-s, but this is the wrong choice because p-s-s 1 is actually the closest: the number of blocks which must be moved to transform the root into the goal state is three, while only one move is needed for p-s-s 1.

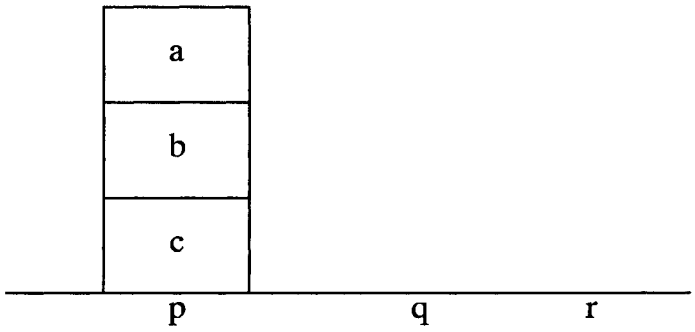


Figure 6a Root - Start State

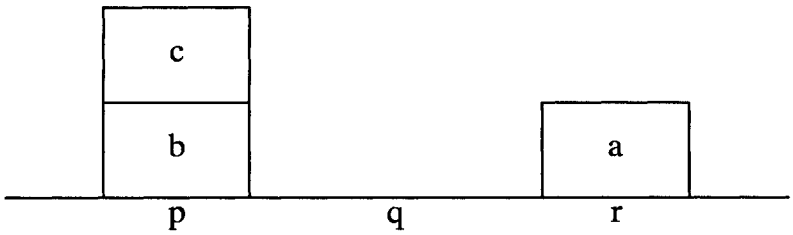


Figure 6b P-s-s 1

Though the method used by Madame does not always find the correct closest p-s-s, it was chosen for its simplicity and effectiveness. In most cases, our experiments show that the correct p-s-s will be chosen.

Updating the Spider

After the planning component has extended the plan between the start state and the closest p-s-s to the goal state, the goal state is added to the spider. There are two possible parents for the newly added p-s-s: (1) the p-s-s closest to the goal state, or (2) the start state. As the planner uses a regression strategy (cf. Nilsson 1980), it will sometimes insert actions in the plan which transforms the start state into the closest p-s-s. If actions were inserted, then the start state will be the parent. Otherwise, the closest p-s-s will be the parent of the newly added p-s-s.

All newly generated plans are added to the spider. Currently, there is no implemented policy for retiring or rejecting p-s-s's. One possible criteria would be to only accept a new p-s-s if the plan represented by the edge between the new p-s-s and its parent contains *n* or more actions. This would represent accepting new p-s-s's for which a lot of work had been expended to generate a plan.

Description of experiment

For the experiment, identical planning queries were passed to Madame and to Madame's planning component. The difference being that the former had the benefit of the spider. The latter, Madame's planning component, only had access to the original start state.

Although Madame's intended problem domain was UNIX, the blocks world was chosen for experimentation. At this time, only a small portion of the UNIX domain has been encoded.† The experimental problem domain used here is a five blocks world with three table positions. The start state is shown in fig. 7. During the experiment, total time was measured.

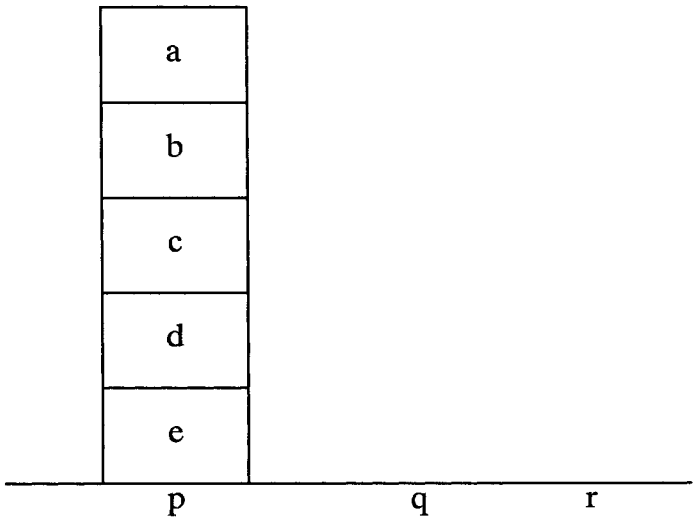


Figure 7 Start State

† Test results from the UNIX domain can be found elsewhere (Trudel 1984).

Input for the experiment consisted of sixty-one lists of goals, each list represents a goal state. For example, the first list of goals was:

on(c,q) & on(e,b) & clear(e)

The number of goals in each list and the goals themselves were generated randomly. It is possible that the queries could have been generated so that one planner was favoured over the other, e.g., queries could be generated where the closest p-s-s in the spider for each query would be the root (the start state). At the other extreme, the same query could have been repeated sixty-one times. The planner using the spider would produce a plan once while the other planner would re-generate the same plan sixty-one times. The planner using the spider would be the clear winner in this case.

Experimental results

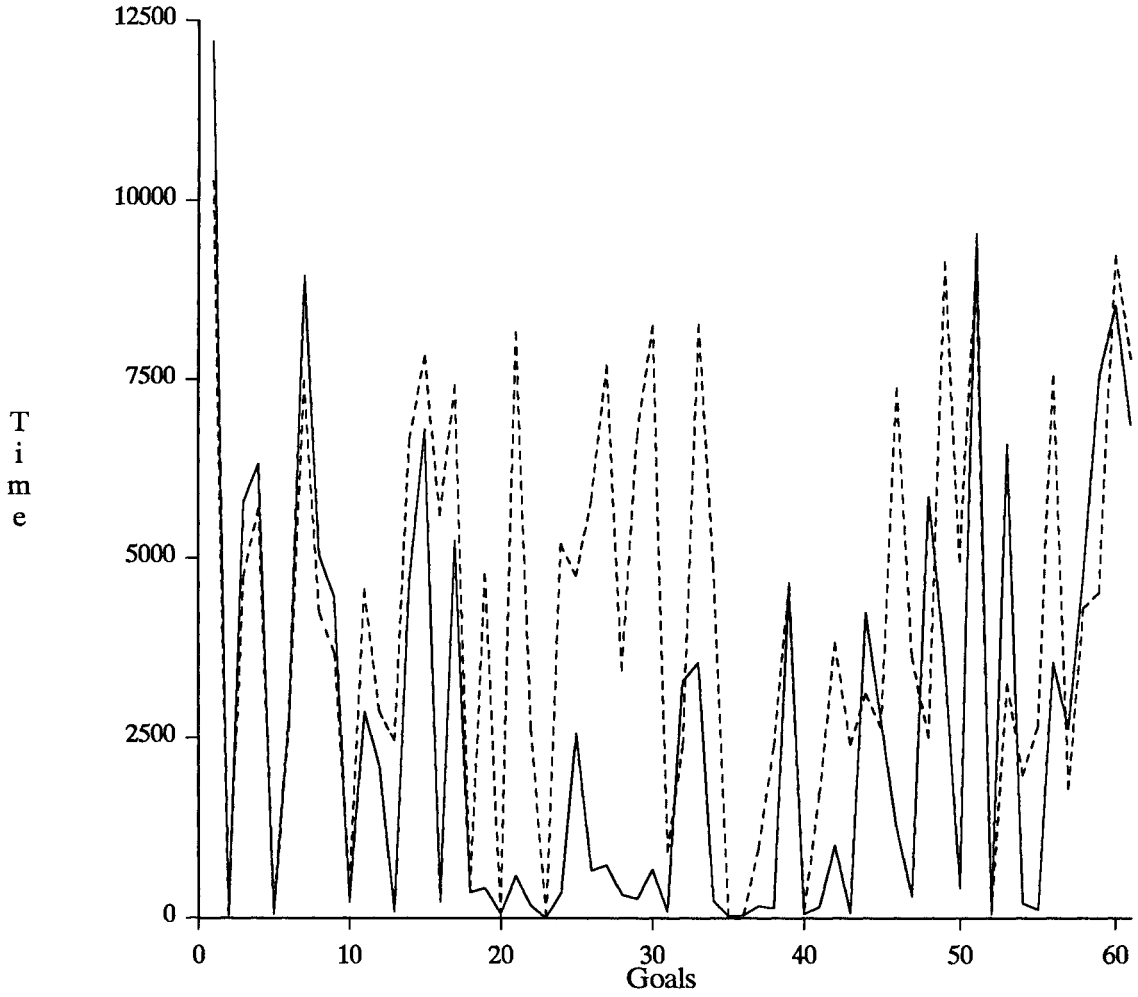
The time taken to generate each goal state is shown in fig. 8a. The spikyness of the graph is due to the fact that easy and difficult goal states are intermixed. To remove the spikes, cumulative time versus number of goals is plotted in fig. 8b. Fig. 8c is a plot of the difference between the two curves in fig. 8a. For each goal state, the solid line is subtracted from the dashed line.‡ Madame is more efficient whenever the curve is above the zero axis.

Experimental conclusions

Figs. 8b and 8c show that once the spider has grown to a certain size, Madame becomes more efficient. At the beginning of the test, the overhead of keeping the spider makes the planner less efficient. Eventually, the spider will attain a size where the probability of finding a p-s-s which is closer to the goal state than the start state is relatively high. The crossover point is marked with an "A" in both figures.

Madame as yet has no strategy for deleting p-s-s nodes. As the spider becomes large, the time required to maintain and use the spider will diminish its beneficial properties. This situation first occurs at point B in fig. 8c. But surprisingly, the curve oscillates beyond point B. In this region, a point above the x-axis means that a "good" p-s-s was found, and a point below the x-axis means that time was wasted searching and updating the spider. We speculate that the curve would continue to oscillate if the experiment was continued beyond sixty-one goal states. It was observed that there were 28 p-s-s's in the spider after sixty-one queries. Since the spider has grown slowly, it is reasonable to hypothesize that as more queries are solved, the spider will remain small in proportion to the number of queries solved. Also, as the spider grows, the probability that it will contain a p-s-s near the goal state increases.

‡ Graphs similar to fig. 8b can be found elsewhere (Trudel 1984). The experiments were performed in the UNIX and blocks domain.



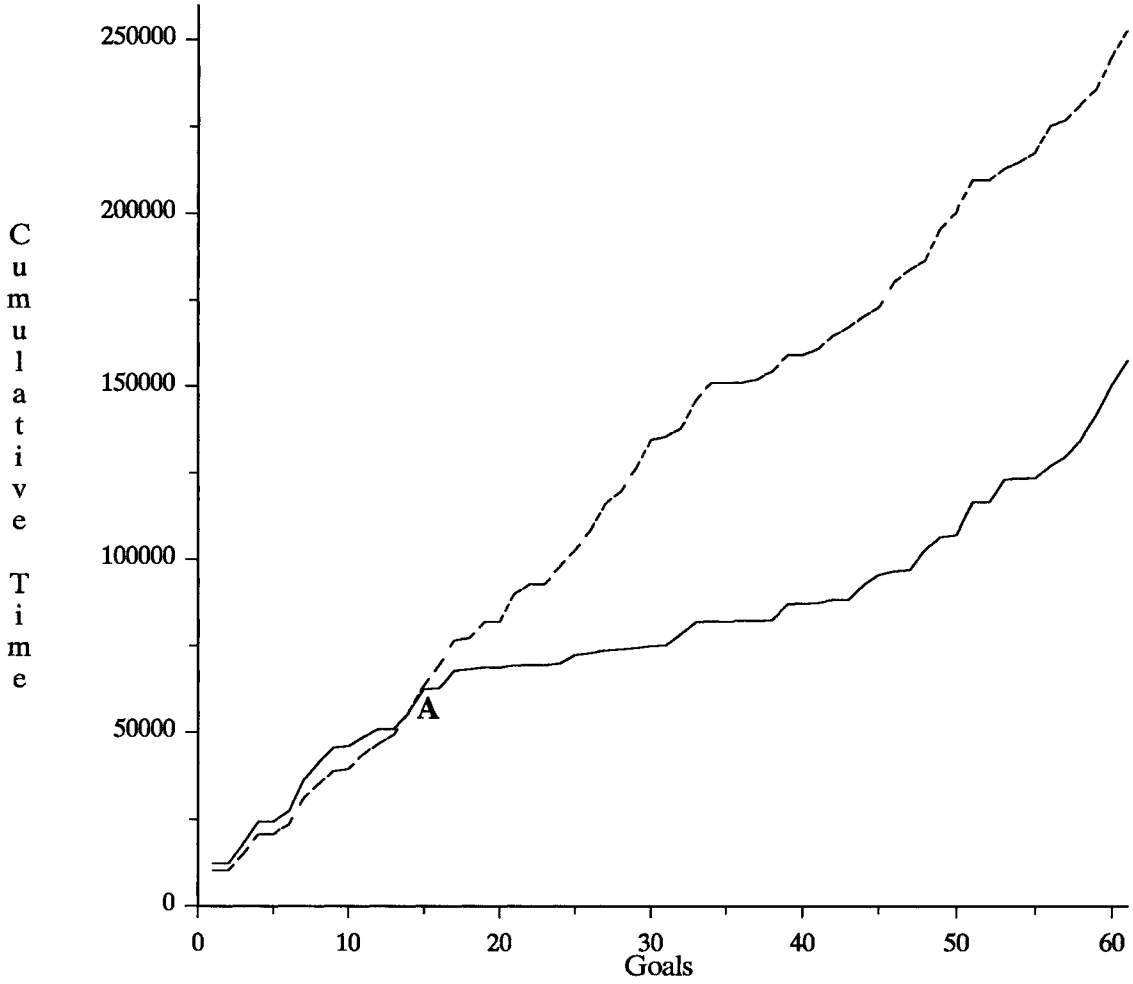
Solid Line: Madame

Dashed Line: Madame's planning component

Figure 8a Experimental Results

Comparison with previous work

One of the earliest attempts at saving plans for subsequent use, was STRIPS' MACROPS or macro-operators (Fikes et al. 1972). A MACROP is a generalized version of a plan. The generalization is obtained by replacing problem specific constants with variables. The MACROP is stored in a data structure called a triangle table. The triangle table allows STRIPS to access any subsequence of the MACROP efficiently. Every time STRIPS must extend the plan it is working on,



Solid Line: Madame

Dashed Line: Madame's planning component

Figure 8b Cumulative Time

it will consider unique subsequences of the MACROPS in addition to the actions. This approach quickly leads to an explosion of alternatives at each planning step.

Another problem with MACROPS is that "...even 'useless' operator sequences, such as STACK(x,y) followed by UNSTACK(x,y), can become macro-operators" (Minton 1985). This problem never arises with Madame because every node in the spider describes a unique state.

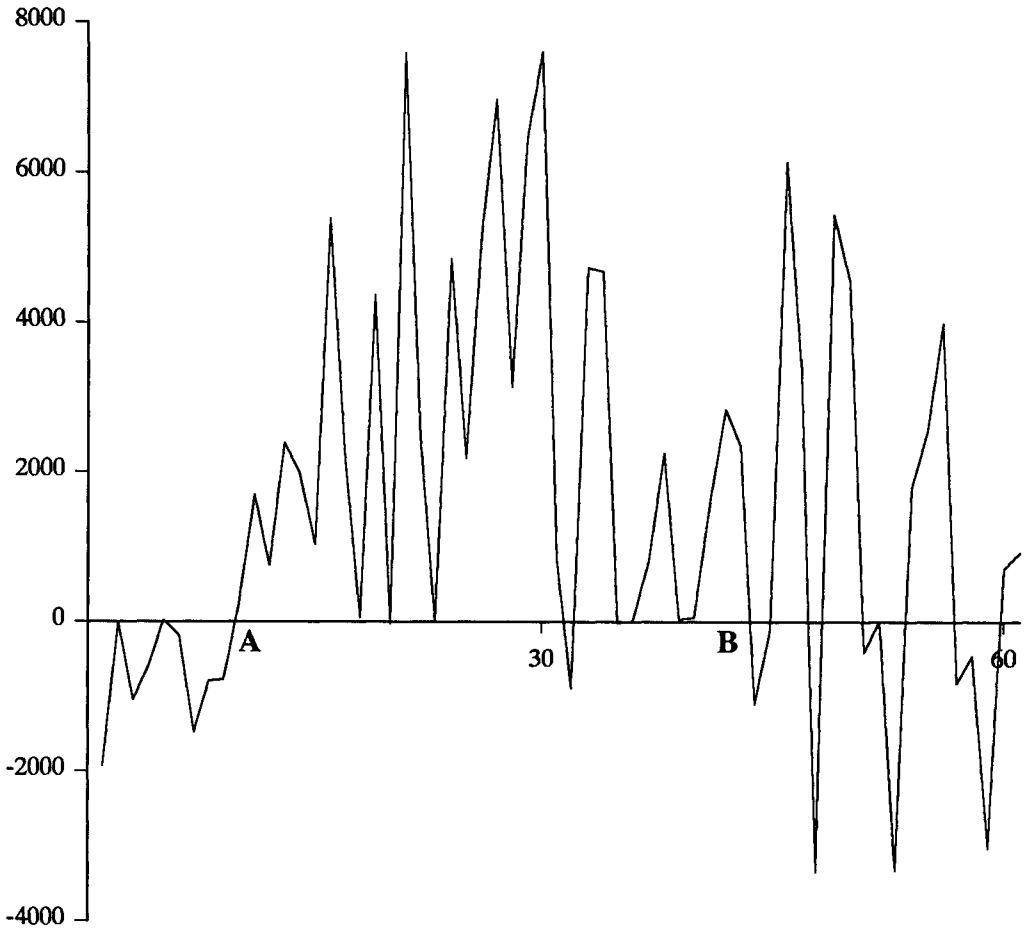


Figure 8c Difference Curve

To avoid being swamped by the number of MACROPS, Minton (1985) added heuristics to prune the number saved. He called his planner MORRIS. "Currently MORRIS saves two types of macro-operators, s-macros and t-macros. S-macros, or 'scripts', are frequently used operator sequences. T-macros, or 'tricks', are operator sequences for solving difficult problems." (Minton 1985). A limit on the number of s-macros is maintained; when this limit is exceeded, the least frequently used macros are deleted. This strategy could also be applied to the spider to control its growth. The maximum number of p-s-s's can be determined with the aid of fig. 8c. Since we want the curve to remain above the x-axis, a point is chosen between A and B. The size of the spider at this point, s-limit, will be the maximum number

of p-s-s's allowed. If the experiment were now performed with a different set of goal states and only s-limit p-s-s's were allowed in the spider, the behavior after point B would probably never occur. Adding heuristics to control the number of saved plans is more important for STRIPS due to the large number of MACROPS generated. Madame does not have this problem because it only stores the goal states and not the intermediate states needed to achieve the goal state.

Another system that saves sequences of actions is SOAR/CHUNKING (Laird et al. 1985). A "chunk" represents a sequence of actions. CHUNKING was added to SOAR, which is a planner that "...can create goals to reason about any aspect of its problem-solving behavior" (Laird et al. 85). For example, when SOAR must choose one of possibly many applicable actions at some stage, it will formulate the selection as a planning subproblem.

The CHUNKS are not restricted to being subsequences of the eventual plan. The CHUNKS are created dynamically whenever a subgoal is successfully completed. As with MACROPS, appropriate constants are replaced with variables.

Although no data is presented in (Laird et al. 85), SOAR/CHUNKING will probably face the same combinatorial explosion as STRIPS with MACROPS. We therefore further speculate that both systems would exhibit behavior similar to fig. 8c. But, beyond point B, the curve would be negative and have a negative slope.

Concluding remarks

Madame sometimes produces a plan which is not optimal. For example, assume the spider contains two p-s-s's (the root and p-s-s 1) and the edge linking the p-s-s's represents a plan containing one hundred actions. Further assume that the list of goals LG passed to Madame hold in p-s-s 1. Madame will therefore return the plan linking the root, the start state, with p-s-s 1. But, it is possible that a more efficient, perhaps optimal, plan could have been constructed by applying fewer actions to the start state. This optimal plan would not have been considered by Madame; plans that are found in the spider will be used before any other planning is done.

The spider obtains new p-s-s's from plans which have been generated. Since Madame uses the spider, it generates new plans from old plans. This means that as the user asks for plans, they are recorded in the spider. As every user can have his own spider, the spider crudely models a user's history of interaction with the planner. Note also that the generated plans are in the user's language, i.e., portions of the plan will already be familiar to him.

As the user asks for plans, the spider will grow larger because new p-s-s's will be added. This means that the spider will "know" more about the problem domain and be more helpful in planning. Therefore, the spider learns about the problem domain along with the user. The experimental results show that the growth of the spider is not a major concern.

Acknowledgements

We would like to thank Cindy "Madame" Trudel for her helpful comments, and David Poole for his sage advice. This research was supported by National Sciences and Engineering Research Council of Canada grant A0894.

References

R. Fikes, P. Hart, and N.J. Nilsson (1972), *Learning and Executing Generalized Robot Plans*. Artificial Intelligence 3, 4, 251-288.

R.A. Kowalski (1979), *Logic for problem solving*. Elsevier North Holland, New York.

J.E. Laird, P.S. Rosenbloom, and A. Newell (1985), *Towards Chunking as a General Learning Mechanism*. Technical report CMU-CS-85-110, Carnegie-Mellon University, Pittsburgh, USA.

J.L. Malito (1983), *Project Report for CS786*. Department of Computer Science, University of Waterloo, Waterloo, Canada.

J.L. Malito (1984), *UWISH: The UNIX Wizard Component of the Intelligent Shell*. M. Math essay, Department of Computer Science, University of Waterloo, Waterloo, Canada.

S. Minton (1985), *Selectively Generalizing Plans for Problem-Solving*. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, USA, 596-599.

N.J. Nilsson (1980), *Principles of Artificial Intelligence*. Tioga Publishing Co., Palo Alto.

J.L. Pakalns (1983), *AHA, A UNIX Consultant!* M. Math essay, Department of Computer Science, University of Waterloo, Waterloo, Canada.

P.S. Rosembloom, and A. Newell (1982), *Learning by Chunking: Summary of a Task and a Model*. Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, USA, 255-257.

P.S. Rosenbloom (1983), *The Chunking of Goal Hierarchies: A Model of Practice and Stimulus-Response Compatibility*. Technical report CMU-CS-83-14, Carnegie-Mellon University, Pittsburgh, USA.

E. Sacerdoti (1979), *Problem Solving Tactics*. Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan, 1077-1085.

J.A.N.A. Trudel (1984), *Madame: a planner for ISH*. M. Math thesis, Department of Computer Science, University of Waterloo, Waterloo, Canada.

D.H.D. Warren (1974), *Warplan: A System for Generating Plans*. Memo No. 76, Department of Computational Logic, University of Edinburgh, Edinburgh.

R. Wilensky, Y. Arens, and D. Chin (1984), *Talking to UNIX in English: an overview of UC*. ACM Communications **27**(6), 575-593.