Marvin: Macro Actions from Reduced Versions of the Instance

Andrew Coles and **Amanda Smith**
Department of Computer and Information Sciences,
University of Strathclyde,
Livingstone Tower,
26 Richmond Street,
Glasgow,
G1 1XH
email: firstname.lastname@cis.strath.ac.uk

Abstract

Marvin is a forward-chaining heuristic-search planner. The basic search strategy used is similar to FF's enforced hill-climbing with helpful actions (Hoffmann & Nebel 2001); Marvin extends this strategy, adding extra features to the search and preprocessing steps to infer information from the domain.

Introduction to Marvin

Marvin is a forward-chaining domain-independent planner that uses a relaxed-plan heuristic to guide its search. The name Marvin stands for Macro-Actions from Reduced Versions of the INstance and gives some insight into the way in which the planner works: it attempts to create a reduced instance of the problem with which it is presented, solve this smaller instance, and then use the solution to assist with solving the original problem.

Basic Search Strategy

The basic search used is similar to FF's enforced hill-climbing with helpful actions (Hoffmann & Nebel 2001); Marvin extends this strategy, adding extra features to the search and preprocessing steps to infer information from the domain. This section details the modifications made to the search strategy.

When plateaux are encountered Marvin resorts to best-first search as opposed to breadth-first search—in practise this improves its performance but may increase the makespan of the plan.

To reduce the overheads incurred by memoising already-visited states no record is kept of visited states if search is progressing normally; however, should a plateau be encountered, the differences between states on the plateau and the state at the start of the plateau are memoised, and states whose difference has already been memoised are pruned.

To prune action choices Marvin constructs groups of symmetric objects (objects with identical properties), extracts one exemplar from each group and then prunes actions which involve any entities which are not the exemplar for their group; for example, in the gripper domain, if two balls are symmetrical in a given state it will only consider applying the pickup action to one of them.

Marvin can exploit the potential for concurrency in solution plans by considering, at each choice point, all of the actions that could be applied at the current time point (t) before considering the actions that could be applied at the next time point (for non-temporal domains this is simply $t + 1$). This approach increases the branching factor and could thus become very expensive during periods of exhaustive search; hence, during such periods the concurrency reasoning is suspended until the plateau is escaped. The steps to escape a plateau are then post-processed to reintroduce concurrency where possible.

Instance Reduction

Before attempting to solve the problem instance with which it is presented, Marvin creates a smaller instance of the problem. This approach was motivated by the observation that small instances can be solved quickly and their solutions often contain action sequences similar to those in solutions for larger problem instances. Any knowledge that can be obtained inexpensively by solving a smaller instance will be valuable in solving the larger instance that was given to the planner.

Smaller instances are created using symmetry and almost-symmetry. Two objects are symmetric if, and only if, they share the same predicates in the initial and goal states: this is the definition of symmetry used previously by STAN version 3 (Fox & Long 1999). In many domains this reduction does not discard sufficient entities to create a significantly smaller problem, hence further pruning is desirable; this is achieved through the use of almost-symmetry. In this context two objects are almost symmetric if, and only if, the predicates defining them in the initial and goal state are of the same type and they differ only in groundings of one or more arguments of a the predicates. For example, in the problem below (where all predicates involving package1 and package2 are shown):

Initial State

at package1 loc1
at package2 loc2
...

Goal State

at package1 loc3

at package2 loc4
...

the two packages are 'almost-symmetric': they only differ by one binding in the initial state (the location they are at) and one in the goal state (their destination).

Using this definition of almost-symmetry the symmetry in the solution plan for these two entities will be captured, as well as strict symmetry in the problem: if two objects share the same predicates in the initial state (even if the groundings of these predicates differ) it is likely that the same, or a similar, plan can be used to achieve the required goals for both objects.

When the extraction of groups of related objects is completed a new smaller problem instance is created by taking one exemplar from each related group and including only the predicates whose entities are wholly contained within this set of exemplars; the smaller instance is then solved, using the search algorithm described in the previous section, to generate a solution plan.

The plan generated to solve the smaller instance is processed to produce macro-actions. Partial-order lifting is used to extract independent threads of execution in the plan; after extraction independent threads are made into individual macro-actions and are added to the list of actions to be used in planning to solve the original instance. Whilst adding actions does increase the branching factor the additional actions often assist in the planning process as they encapsulate a previously-successful strategy for solving a similar problem.

It should be noted that for some domains—for example, freecell—the reduced problem is unsolvable; in such situations it is usually the case that the problem is proven unsolvable very quickly: the goals do not appear in the relaxed planning graph. For situations in which the goals are present in the relaxed planning graph it is necessary to introduce an upper bound on the plan length allowed to ensure that an unreasonable amount of time is not spent solving the smaller instance; in practise this does not prevent Marvin from generating useful macro-actions as preliminary experiments show large macro-actions are often too specialised to a certain task and are therefore not reusable.

Plateau-Escaping Macro-Actions

Solutions to planning problems often contain a given sequences of actions more than once; if finding this reused action sequence corresponds to exhaustive search a lot of unnecessary search effort is expended in repeatedly attempting to find this action sequence. Marvin attempts to improve on the plateau behaviour of previous forward-chaining planners by memoising the action sequence which successfully lead from the start of a plateau to a strictly-better state; these memoised action sequences form what are known as plateau-escaping macro-actions. To reduce the overheads of having a greater number of actions to consider at each state these plateau-escaping macro-actions are only considered when plateaux are encountered: in normal search only the original actions from the domain, and any actions derived from the solution to the reduced instance, are used.

When solving the reduced instance any plateau-escaping macro-actions devised are stored for use when later solving the original problem; this has the useful side-effect of discovering efficacious escape macros with less computational effort—it is less computationally expensive to perform the plateau-escaping search on the reduced instance of the problem. Furthermore, since the reduced instance is derived from the original problem instance, it is often the case that the heuristic breaks down when solving the reduced instance in some of the places it breaks down when solving the original problem instance.

As with the macro-actions created from the reduced version of the instance the plateau-escaping macro-actions have a partial order lifted out, the aim of which is to improve the concurrency within them, reducing the makespan. Once this processing has taken place the segment of plan which escaped the plateau is replaced with the macro-action: the macro-action may exploit concurrency which the original plan segment did not.

Transformational Operators

Transformation operators are those operators that transform a certain property of an object but leave other objects unchanged; for example, the action move in the driverlog domain:

pre:

```
at (truck loc1)
linked(loc1 loc2)
```

add:

```
at (truck loc2)
```

del:

```
at (truck loc1)
```

transforms the 'at' property of trucks. The reusability of macro-actions is adversely affected by transformation operators, as they often appear in chains of varying lengths; consequently, abstraction of the length of these chains is required if the macro-action is to be as reusable as possible.

Generating sequences of transformational operators is a shortest path problem, which can be solved by a specialist solver. Marvin currently recognises transformational operators by looking for a common fingerprint; however, in the future TIM (Long & Fox 2000) will be used to provide a method through which these operators can be identified in a more-robust manner.

When transformational operators have been identified an all-pairs shortest-path reachability analysis is done, during which the best route between two states is stored; then, static predicates for all pairwise reachable states are added to the initial state so that Marvin can plan as if the states were all linked. When an action is later selected for application the main algorithm simply asks the sub-solver for the action sequence required to achieve the desired effect.

ADL

Marvin supports ADL natively; that is, without creating distinct STRIPS actions for each of the possible ADL action

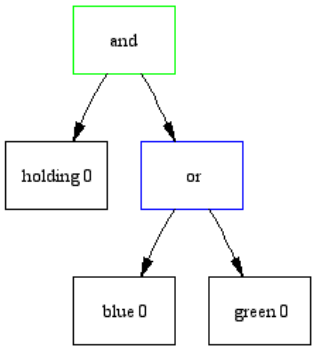

Figure 1: Example Satisfaction Tree

groundings. ADL support was written for the purpose of solving the competition ADL domains—without it, due to the nature of the STRIPS compilations provided, Marvin would not have been able to construct any reusable macro actions.

ADL preconditions are dealt with through the logical reduction of each operator's preconditions to form a 'Satisfaction Tree'. The idea is to create a tree where the leaves are predicates (or negations of predicates) and the internal nodes are either conjunction or disjunction nodes (AND or OR); then, predicates either help a given ground action become applicable (if they appear as positive predicate leaves in its satisfaction tree) or hinder its applicability (if they appear as negative predicate leaves). The tree is formed by recursively applying the following rules to each action's preconditions:

$$(\forall x f(x)) \Rightarrow (f(x_0) \wedge \ldots \wedge f(x_n))$$
$$(\exists x f(x)) \Rightarrow (f(x_0) \vee \ldots \vee f(x_n))$$
$$(a \Rightarrow b) \Rightarrow (\neg a \vee b)$$
$$(\neg(T_0 \wedge \ldots \wedge T_n)) \Rightarrow (\neg T_0 \vee \ldots \vee \neg T_n)$$
$$(\neg(T_0 \vee \ldots \vee T_n)) \Rightarrow (\neg T_0 \wedge \ldots \wedge \neg T_n)$$

The first two of these simply compile out the existential quantifiers dynamically; the third is a logical reformation of the implies operator; the final two, forms of De Morgan's duality law, are used to force any negation into the subexpressions, and eventually to the predicates.

Figure 1 shows an example satisfaction tree for an action in an imaginary domain in which objects can only have a certain action applied to them if they are being held and are either blue or green.

ADL effects are handled in a similar manner to preconditions, in that they form 'Effect Trees'; there are differences, however, due to the differing semantic structure between Preconditions and Effects: Effect Trees do not contain OR nodes; instead they introduce 'When' nodes. When nodes have two child branches - a condition branch (which is, itself, a satisfaction tree) and an effect branch (which is an effect tree). When an action is grounded any unconditional effects and effects contingent only on static predicates are associated with the ground action instance; sub-actions are then created to encapsulate any effects contingent on dynamic information.

The relaxed planning graph in Marvin is modified to account for the negative preconditions required by ADL. Before the ADL support was implemented a spike (Long & Fox 1999) for positive predicates was used; to build a relaxed planning graph forward from a given state the spike was initialised to contain the predicates in a given state and then grew as applied relaxed actions added predicates to it. To support negative preconditions a second spike was created; this spike is initialised to be empty and then any predicate present in the initial fact layer which is then, later, deleted is added to it. A negative precondition is then satisfied at a given layer in the relaxed planning graph either if it isn't present in the initial fact layer or it has since appeared in the negative fact spike.

Future Work

In the future Marvin will be extended to use the generic-type recognition knowledge provided by TIM (Long & Fox 2000). This will, amongst other things, improve its support for transformational operators by providing a flexible framework for their identification; also, it raises the possibility of using generic-type-derived heuristics to improve the discrimination between states when the relaxed plan graph heuristic reaches a plateau.

Marvin will also be extended to deal with Temporal Planning: as it already uses macro-actions and concurrency, much of the framework is already complete.

References

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *IJCAI*, 956–961.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Reserach* 14:253–302.

Long, D., and Fox, M. 1999. Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research* 10:87–115.

Long, D., and Fox, M. 2000. Automatic synthesis and use of generic types in planning. In *Artificial Intelligence Planning Systems*, 196–205.