An Integrated Planning Representation using Macros, Abstractions, and Cases

Jacky Baltes and Bruce MacDonald
Computer Science Department
The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4, Canada
{baltes,bruce}@cpsc.ucalgary.ca
ph. (403) 220–5112

June 13, 2001

Abstract

Planning will be an essential part of future autonomous robots and integrated intelligent systems. After giving a brief introduction to the classical planning paradigm, this paper focuses on learning problem solving knowledge in planning systems. A general weak method for learning useful operators is the creation of macros. The paper first describes a novel approach to the selection and dynamic filtering of macros. The dynamic filtering approach is suggested for controlling the creation of operators. A new planning representation is proposed that uses a common representation for macros, abstractions, and cases. A general operator is represented by sequences of primitive or non–primitive operators. A macro is equivalent to a sequence of primitive, executable, operators with uninstantiated arguments. A case consists of primitive operators with instantiated arguments. An abstract plan is equivalent to a sequence of non–primitive operators at a lower level of abstraction. A learned indexing mechanism allows rapid access to relevant operators. The system is able to use both classical and case–based techniques. The general operators in a successful plan derivation would be assessed for the potential usefulness, and some stored.

1 Introduction

A great deal of research effort has been put into the development of planning systems. First attempts at computer problem solving started in the 1960's [?]. Goal directed behavior seems to be a cornerstone of human behavior.[1] Since programming a systems with solutions to all possible problems is overwhelming or simply impossible, an autonomous agent has to be able to generate plans if it is to function in non–trivial domains. This paper reviews classical planning, examines macro operator generation in detail, proposes a new dynamic macro filter and suggests a representation for learned operators that includes abstract plans, cases and macros, and uses a general dynamic filter to reduce the complexity. This is work in progress and we finish by indicating how we expect our learning planner to use the representation. A prototype system for learning in the common representation has been implemented.

2 The Classical Planning Paradigm

Early planning systems introduced the state space paradigm. The world is represented by a snapshot. The state of the world is described by a set of facts that are true in the given snapshot. The actions that can be executed by an agent are represented as operators. Operators map a world state into the state resulting from the action. The division of the world into discrete snapshots means that it is difficult to model continuous processes, in particular time, in the classical planning paradigm.

In order to reduce the number of necessary operators, the representation usually allows variables. So rather than operators for all colors of a disk (`move-red-disk, move-blue-disk, ...`), only one operator is necessary (`move-disk (color = $x)`).

STRIPS [?] was the first planner to address the frame problem, providing an easy way to specify what facts remain true after the application of a given operator and what facts do not. STRIPS assumes that everything not explicitly stated in the post–conditions of an operator remains true. Every operator in STRIPS maintains an add–list and a delete–list as its post–condition. For example, the operator to move the small disk from one peg to another in the towers of Hanoi problem can be described as follows:

```
move-small-disk($A,$B)
  Pre-conditions:  (is-peg $A)  (is-peg $B)  (is-on $A,Small-Disk)
  Post-conditions: Add-List:    (is-on $B,Small-Disk)
                   Delete-List: (is-on $A,Small-Disk)
```

The input of the planning system is a set of operators, an initial state, and a goal state. The output is a sequence of operators and variable instantiations that transform the

[1] This statement should be interpreted generally; to include for example the notion of reacting to the current situation, as well as planning ahead, both *to achieve a goal.*

initial state into the goal state. The state space paradigm reduces planning to a search problem in the space of possible operator sequences. The practical use of this method is greatly restricted by its time complexity, which is exponential in the length of the solution.

Classical planning work has investigated various learning methods for speeding up the planning process. This includes the use of macros, which are uninstantiated operator sequences, and more recently case–based planning, which remembers instantiated operator sequences for later modification and reuse. Our representation also naturally represents abstract plans, which are formed by successively refining increasingly detailed operator sequences. Abstract planning can speed the planning process as plans are expanded in detail only if there is a successful higher level plan.

3 Macro–Operators

A macro is a named sequence of operators, and a linear sequence is the simplest form of macro. There has also been research in the design of systems that learn disjunctive or iterative macros [**?**]. This paper focuses on linear macros since the generation of disjunctive or iterative macros requires useful linear macros. In the blocks world domain for example, a useful macro is `pickup = (goto, grasp)`.

3.1 Complexity of Macros

Korf [**?**] quantitatively analysed the effect of macros on time complexity. The complexity of planning is largely determined by the branching factor (the choice of operators) in the search space. The generation of new macros must be carefully controlled, since the increase in the branching factor may outweigh the reduction in computation associated with a shorter solution, as seen below.

Let us assume that b is the branching factor of the search space — the average number of primitive operators applicable in a state — and that n is the length of the solution in terms of primitive operators. The complexity of brute force search is then $O(b^n)$. Now assume a planner has stored all possible two–operator macros. We will assume b remains the same. At any stage of the planning, there will on average be b^2 two–operator macros applicable (the b applicable primitive operators, each with b applicable operators to follow in the second step of the macro). An optimal plan that turns out to have an even number of steps will be $n/2$ long (all macros), while an odd length optimal plan will have one primitive operator. Assuming that the planner is unaware of the coverage of its operator set, it must consider all primitives and macros applicable at each stage. Thus, for optimal even length plans, the time complexity is $(b + b^2)^{n/2}$. The ratio of macro–planning and brute force complexity is given by

$$\frac{(b + b^2)^{n/2}}{b^n} = (1 + \frac{1}{b})^{n/2}$$

This ratio is greater than one, and it grows exponentially with the solution length, which shows that the complexity of the macro–planner is worse than that of the brute force planner. All primitive operators must be considered, so the completeness of the planning system is guaranteed. Minton showed experimentally that the performance of MACROPS, a version of STRIPS that stores all possible macros from a solution, is worse than the performance of the original STRIPS planner [**?**].

3.2 Dynamic Filters

It follows from the previous section that only a small number of macros should be generated, ideally ones that will be useful in future problem solving tasks. The basic problem of macro learning is that the system has to predict the usefulness of a macro based on its previous experience. The following paragraphs describe the effect of adding one macro to the operator set and derive a *usefulness* measure for such an addition. This measure can be used to dynamically delete macros.

By adding a macro m, the branching factor is increased. However, the new macro will not be applicable in all situations. Let b be the branching factor without the macro in question. Let c be the fraction of states where m is applicable. Furthermore, not all applications of m will lead to a solution, so let u_m be the *usefulness* of m, which is the overall chance of applying m to achieve a solution; the ratio of the total number of times m leads to a solution, to the total number of times any operator is applicable. If l_m is the number of primitive operators in m, then the time complexity for the new operator set is of the order:

$$(b+c)^{n/l_m u_m}$$

The branching factor is increased by the applicability of m, and the effective solution length is reduced by the chances of using m, and in proportion to m's length. If u_m is 1, this means m is used at each step of the solution, and the plan is divided in length by the number of operators in m. If planning is to be faster when a macro is added, then the following inequality must be satisfied:

$$b^n \geq (b+c)^{n/l_m u_m} \tag{1}$$

$$\log b \geq \frac{1}{l_m u_m} \log(b+c) \tag{2}$$

$$u_m \geq \frac{1}{l_m} \cdot \frac{\log(b+c)}{\log b} \tag{3}$$

The macro length predominates the generality of its pre–conditions, c, in 3. So it will be more important to allow long macros than more specific macro pre–conditions. However, the pre–conditions cannot be ignored; impractically large values are required for b to make the second fraction in 3 approach unity. Note that u_m and l_m are not

independent; as the length grows the chances of the macro being used in a plan decrease. Furthermore, it is also assumed that the space searched does not change with m. Under this assumption, b and c are independent.

Equation 3 cannot be used directly for selecting new macros because u_m, b, and c cannot be effectively computed *a priori*. However, these values can be approximated statistically. After a number of trials, equation 3 can be used as a dynamic filter to remove unnecessary macros.

Iba [**?**] proposes dynamic filters in his MACLEARN system. However, the implementation seems ad hoc; the user determines when to call the dynamic filter routine, which deletes all macros that have not been used at least once in a previous problem solving episode.

3.3 Macro selection

As mentioned above, simply generating all macros of a given length increases the time complexity of planning. Even given a dynamic filtering scheme as described above, performance will be considerably worse, until all macros that are not useful are deleted from the operator set. Macros should be more selectively generated and only promising candidates added to the operator set. This section compares macro selection in three different approaches: Korf's Macro Problem Solver (MPS) [**?**, **?**], Iba's MACLEARN [**?**], and Yamada's and Tsuji's PiL2 [**?**].

MPS MPS was one of the first systems to show the usefulness of macros as a weak (domain–knowledge free) learning method. MPS successfully solved the 3×3 Rubik's cube puzzle, which is an interesting problem because no general purpose heuristic, such as means ends analysis or steepest descent, is better than brute force search. MPS creates a macro–table for all subgoals in the given, ordered goal conjunction and all possible values of the state variables that achieve those subgoals. The entries in the macro–table are macros that achieve the subgoals, given that all previous subgoals are satisfied. This requires *serial operator decomposability*: macros must be independent of succeeding subgoals in the solution order because there can only be one entry for each slot in the table. For example, when trying to solve the towers of Hanoi problem, the smallest disk must be moved to the goal peg first. Moving the largest disk first would mean that, depending on where the smaller disks are, different macros must follow. *Serial operator decomposability* guarantees that the number of macros necessary to solve the same problem from any initial state is equal to the product of the number of subgoals and the number of values for each subgoal. Otherwise the number of macros grows exponentially in the number of subgoals. MPS can reach the same goal state from any initial state in linear time.

However, the time complexity of the brute force search for macros is exponential. Also, the algorithm may not terminate even when a solution exists or is already

stored in the macro–table. For example, MPS may search forever to fill a slot in the macro–table corresponding to the last cube being in the right position but oriented incorrectly (a state that is both impossible and unsolvable). Macros contain only elementary operators and are therefore not generalized to new goal statements.

MACLEARN MACLEARN [?] uses a peak–to–peak heuristic to delimit subsequences of a plan formed during search, and proposes them as potential macros. This avoids the exponential complexity and possible non–termination of searching for macros. A peak in a heuristic function is a node in the search path whose heuristic value is higher than that of its two neighbors. Whenever a peak in the solution path is detected, the sequence from the previous peak (or root if there is no previous peak) to the current peak is extracted and proposed as a new macro. The reasoning behind the peak–to–peak heuristic is that this new macro allows the search to traverse valleys in the search space more rapidly. The proposed sequence is generalized and passed through a static filter, which uses redundancy, macro length, and other domain knowledge to rule out poor macros. As mentioned previously, MACLEARN also uses a user–invoked dynamic filter to discard unused macros. Macros are generalized, and can be built from other macros. However, MACLEARN uses a best first search rather than a goal directed strategy such as means ends analysis. This means that the system has difficulty traversing plateaus in the search space, where all neighboring nodes have the same heuristic value. In this case, means ends analysis can provide focus for the search algorithm.

PiL2 Yamada and Tsuji propose a different heuristic for the selection of macros in their PiL2 system [?]. As in the MACLEARN system, PiL2 extracts macros from a worked example. The extracted macros are generalized using explanation–based techniques. PiL2 uses the *perfect causality* heuristic to propose new macros. This heuristic selects macros such that (a) the pre–conditions of the last operator of the sequence were not fulfilled in the initial problem state, and (b) after application of the previous steps in the macro, the pre–conditions of the last operator are satisfied and the macro can be applied to the initial state. The basic method of PiL2 is to reduce the pre–conditions of operators. It tries to find macros that allow it to apply more operators to the initial problem state. It is unclear whether this tight connection between the initial state and the learned macro reduces the generality of macros.

4 Abstraction Hierarchies

Abstraction hierarchies allow a planner to ignore low–level details at first and generate a plan whose details can be filled in later. For example, if the goal is to get from one city to another, the planning system can generate a plan to fly, and later fill in the details of how to get to the airport.

Abstraction hierarchies are usually represented by assigning criticality values to all predicates that appear in pre–conditions of operators. For example, ABSTRIPS [**?**, **?**] and ABTWEAK [**?**] use this method. If the planning system is generating a plan at abstract level i, all predicates whose criticality value is less than i are ignored. The plan generated at level i is used to constrain the search at the next lower level of abstraction. The assignment of criticality values does not effect the completeness of the planner. The planning system is guaranteed to find a solution if it exists. Nevertheless, the assignment of criticality values greatly influences the time complexity. Knoblock [**?**] describes an algorithm for computing the criticality values of predicates and proves that the time complexity is reduced to be linear in the length of the solution. This proof is based on the assumption that backtracking occurs only within an abstraction level. This means that a plan at abstraction level i can always be extended into a solution at the next level.

5 Case–based Planning

Case–based systems [**?**, **?**] are motivated by the expense of classical planners, storing the results of planning for future reuse. Plans are indexed by goals, and when a new, but similar goal retrieves a plan, which is modified to solve the new problem. The basic assumption is that the complexity of retrieving and altering a similar plan is less than the complexity of creating a new plan from scratch. Case–based planners repair plans that fail, and store them. The important issues are:

Representation: What facts about a plan are stored and how are they represented?
Indexing: How does the system recall plans for goals similar to the current one?
Adaptation: How are plans modified and repaired to fit the new situation?

Built–in indexing schemes can contain hidden heuristics. A more general approach would be to use induction methods to learn an appropriate indexing scheme.

CHEF's [**?**, **?**] domain is chinese cooking. Its first step when trying to generate a new recipe is to search through its case–memory using a similarity metric. The similarity metric is provided by the user and contains encoded knowledge such as "vegetable substitutions are easier than substituting beef for shrimp." In the next stage, CHEF applies one of its plan critics to fulfill the requirements of the new goal. The resulting plan is then simulated and if successful added to the case memory. If the plan fails, it signals that the expectations of CHEF are wrong and must be repaired. CHEF "realizes" the need to learn new information.

The failed plan is handed to a repair module. If the repaired plan succeeds, CHEF creates a reminder of the problem and its explanation. The explanation is a generalization of the initial state and the features that caused the problem. If a similar goal is selected in the future, CHEF remembers the failure and will automatically anticipate it, trying to find a plan that succeeded in avoiding the same problem in

the past. However, CHEF has no notion of an abstract plan. All cases in memory are fully instantiated. This means that a large number of almost identical plans are stored. As mentioned previously, CHEF depends on a large amount of domain knowledge for plan indexing and plan repair. This makes it hard to adapt it to new domains.

6 Macros, Abstractions, and Cases

This section suggests a common representation for macros, abstractions, and cases in a planning system. It will show the similarity and differences between these methods, and suggest that a common representation will allow a problem solver to use all three strategies simultaneously.

The proposed representation will enable a planning system to maintain important advantages of the systems discussed above:

- The planner will learn only when there is strong motivation, in order to increase performance in the future. This point has been shown by the MACLEARN system (flatten the search space) and by the CHEF system (repair failed plans and anticipate problems).
- Proposed macros are filtered statically as well as dynamically. A new heuristic described in equation 3 is used.
- The planner learns from a worked example (similar to MACLEARN, PiL2, CHEF) rather than using a brute force search to find new operators (which would be similar to MPS).
- The planner should be able to use a heuristic function or other knowledge that is available.

Korf mentioned the similarity between abstractions and macros [**?**]. Both methods try to reduce the search by generating a skeleton search space of the original problem space. Instead of searching in the original space, a solution is found in the skeleton space and this solution is then refined into a solution in the original problem space. One difference, however, is that there can be more than one abstraction level whereas macros normally generate only one skeleton space.

Cases can be viewed as long, specific macros. The main distinction between macros and cases is the way in which they are used in a planning system. Cases are fetched from memory and some plan critics are applied to change the case to the new situation. Macros are usually not altered, i.e. the sequence of elementary operators is not adapted to the new situation.

The common feature among all three items is that the most important information stored is a set of pre–conditions and a set of post–conditions, as for elementary operators.

KEY: $[\mathrm{Pre}, \mathrm{Post}]$ indicates a non–primitive operator, $<\mathrm{Pre}, \mathrm{Post}>$ a primitive one, and $<[\mathrm{Pre}, \mathrm{Post}]>$ either. $<[\mathrm{Pre}, \mathrm{Post}]>^k$ is an operator with predicates at criticality level k or above.

General Operator:

$$[\mathrm{Pre}, \mathrm{Post}] \longrightarrow \begin{array}{l} <[\mathrm{Pre}_{11}, \mathrm{Post}_{11}]>, <[\mathrm{Pre}_{12}, \mathrm{Post}_{12}]> \ldots <[\mathrm{Pre}_{1n_1}, \mathrm{Post}_{1n_1}]> \\ <[\mathrm{Pre}_{21}, \mathrm{Post}_{21}]>, <[\mathrm{Pre}_{22}, \mathrm{Post}_{22}]> \ldots <[\mathrm{Pre}_{2n_2}, \mathrm{Post}_{2n_2}]> \\ \vdots \\ <[\mathrm{Pre}_{m1}, \mathrm{Post}_{m1}]>, <[\mathrm{Pre}_{m2}, \mathrm{Post}_{m2}]> \ldots <[\mathrm{Pre}_{mn_m}, \mathrm{Post}_{mn_m}]> \end{array}$$

Each $\mathrm{Pre}_{i1} \Rightarrow \mathrm{Pre}$ and each $\mathrm{Post}_{in_i} \Rightarrow \mathrm{Post}$.

Macro (uninstantiated arguments) or Case (instantiated arguments):

$$[\mathrm{Pre}, \mathrm{Post}] \longrightarrow <\mathrm{Pre}, \mathrm{Post}_1><\mathrm{Pre}_2, \mathrm{Post}_2> \ldots <\mathrm{Pre}_n, \mathrm{Post}>$$

Abstract operator:

$$[\mathrm{Pre}, \mathrm{Post}]^k \longrightarrow \begin{array}{l} <[\mathrm{Pre}_{11}, \mathrm{Post}_{11}]>^{k-1}, <[\mathrm{Pre}_{12}, \mathrm{Post}_{12}]>^{k-1} \ldots <[\mathrm{Pre}_{1n_1}, \mathrm{Post}_{1n_1}]>^{k-1} \\ <[\mathrm{Pre}_{21}, \mathrm{Post}_{21}]>^{k-1}, <[\mathrm{Pre}_{22}, \mathrm{Post}_{22}]>^{k-1} \ldots <[\mathrm{Pre}_{2n_2}, \mathrm{Post}_{2n_2}]>^{k-1} \\ \vdots \\ <[\mathrm{Pre}_{m1}, \mathrm{Post}_{m1}]>^{k-1}, <[\mathrm{Pre}_{m2}, \mathrm{Post}_{m2}]>^{k-1} \ldots <[\mathrm{Pre}_{mn_m}, \mathrm{Post}_{mn_m}]>^{k-1} \end{array}$$

Automatic subgoaling: $[\mathrm{Pre}, \mathrm{Post}] \longrightarrow [\mathrm{Pre}, \mathrm{Post}_1][\mathrm{Pre}_2, \mathrm{Post}_2] \ldots [\mathrm{Pre}_n, \mathrm{Post}]$

Anticipation of failure: $[\mathrm{Pre}, \mathrm{Post}] \longrightarrow [\mathrm{Pre}, \mathrm{Post}_1]$

Reactive rules: $[\mathrm{Pre}, \mathrm{Post}] \longrightarrow <\mathrm{Pre}, \mathrm{Post}_1>[\mathrm{Pre}_2, \mathrm{Post}]$

RWM–type operator subgoal: $[\mathrm{Pre}, \mathrm{Post}] \longrightarrow [\mathrm{Pre}, \mathrm{Post}_1]<\mathrm{Pre}_2, \mathrm{Post}>$

Figure 1: The representation of planning operators.

Abstraction hierarchies are equivalent to elementary operators that are missing some pre–condition predicates. This means that although the specific execution depends on all pre–conditions, the post–conditions can be achieved independently of the actual value of the deleted predicates in the pre–conditions. One abstract operator can be specialized in a number of different ways. The representation can capture this by associating a set of operators with pre and post–conditions. This structure represents that the post–conditions can be achieved given that the pre–conditions are true, but that the instantiation of the plan may depend on predicates not mentioned in the pre–conditions. A method similar to PiL2's perfect causality heuristic is used to generate new operators that depend on fewer pre–conditions. More than one level of abstraction can be represented by showing that elements of the refinement of an abstract operator can consist of abstract operators.

6.1 Common representation

In our common representation, shown in Figure 1, an operator is recursively represented as (a) a pre– and post–condition pair, and (b) a set of refinements, each of which is an operator sequence. A primitive operator has no refinements, and can be executed.

A variety of well–known problem solving knowledge is supported. An operator is like a macro if (a) there is only one refinement, (b) each operator in the refinement is a primitive, and (c) pre– and post–condition predicate arguments are instantiated in neither the operator nor the refinement (i.e. the macro has formal parameters). A case is an operator with a refinement that is a fully instantiated (long) sequences of primitives (i.e. an instantiated macro). An abstract operator at criticality level k has refinement/s whose operators are abstract ones at level $k-1$.

An operator is a subgoal sequence if all refinements contain no primitive operators (e.g. means ends analysis). Anticipation of failure can be represented by an operator whose single refinement is a single operator pre–, post–condition pair in which there is an additional post–condition (such as "avoid soggy broccoli"). This ensures that the planner knows about the problem, and the refinement of the failure anticipation operator will be expanded using the successful plan, which is also stored as an operator. Since our representation does not enforce a common level of abstraction for operators in the refinement, a case or macro can also be generalized by making some operators non–primitives. This allows us to store adaptations of a case such as the replacement of some steps. Reactive rules may be represented as an operator whose single, two–operator refinement is a fully instantiated, primitive first operator, followed by a non–primitive pre–, post–condition pair. If this two–operator refinement is reversed, then the resulting operator is suitable for backward chaining from the goal (similar to RWM [**?**]).

While this generality provides a common operator representation, it also presents the immediate problem of controlling the creation of operators, so that planning is not impossibly expensive. We intend to control this using the dynamic operator deletion mechanism introduced above. In addition, the learning methods that add operators to the case memory must do so only when there is strong justification, and must choose "important" parts of new plans for storage, deciding the level of abstraction, number of refinements, and so on. This is the subject of current work. The common representation enables us to treat the various kinds of planning system in a single consistent framework, to better aid analysis and comparison.

The planner may choose to "forget" the refinements of some operators, when their usefulness decreases. But the pre–, post–condition pair is retained, and the details can be replanned if necessary.

6.2 Controlling a learning planner

This section indicates how one might use the operator representation given in this paper. The planner should combine case–based as well as classical planning techniques, to take advantage of both previous experience, and the ability to solve new problems. What is needed is a control strategy that recalls and uses previous experiences to solve similar new problems, but gracefully moves into classical planning if no similar cases can be found. The recursive structure of the representation lends itself well to a recursive control strategy. Learning is designed to support and improve the planning process, by storing new operators. The planner should restrict the branching factor of the search space by focusing on a small number of operators instead of all applicable ones. An implementation of our planner already incorporates some of the suggestions below.

6.2.1 Planning

The input to the planner are initial state, goal state, and the operator set. Additional input is a resource limit and a skeleton plan agenda, which may support resource limited and multiple task planning. The planning begins by matching and recalling stored operators that have pre– and post–conditions similar to the current state and goal. The refinement/s of these operator/s will give various types of plans to be considered for solving the current problem.

Recalling similar operators A stored, similar case may have additional pre– or post–conditions, or be missing some. Operators should be recalled when their pre– and/or post–conditions are similar to the current goal and initial state. Possible indexing schemes for recall can be based on the number of predicates in pre– and post–conditions, the predicates themselves, or combinations of predicates.

Recalling is independent of the learned operator hierarchy; the fetched operator is not necessarily at the top level of a refinement tree. For example, if there is an abstract operator to move a medium disk independently of the small disk, and one refinement of this is to move the medium disk when both are on the first peg, and if the current state is that both disks are on that peg, that refinement is retrieved, rather than a more abstract one.

Adapting an existing plan Adaptation of a plan to new initial states and goal states is done by analysing the differences among the initial state and the pre–conditions and among the goal state and the post–conditions of the similar plan. There are a number of general purpose adaptations to a plan that substitute one operator, listed below. If these don't provide a complete plan, then we treat the adapted plan as a subplan and use means ends analysis to complete it.

Replace Steps: An operator should be removed and steps inserted to achieve either

pre– or post–conditions.

- Remove Side effect: If a plan fails because one operator has a specific side effect try to replace this operator with one that works.
- Protect effect: A following operator destroys an effect of the solution. Try to replace this operator with one that does not change the side effect.

Substitution: Replace a variable instantiation with a different object (the operator sequence is unchanged).

To find where to replace an operator, pre–conditions of the case that are not given in the current situation are pushed forward up to the first operator depending on those pre–conditions. Then the planner finds all elements of the post–conditions that are dependent on this operator. If the post–conditions are also part of the current goal, the system generates a new planning problem from the state just before the operator with the non–matching pre–condition to the first operator that uses any of the post–conditions established. For example, if the goal is to have a barbecue and one of the pre–conditions is to have a match, this pre–condition is pushed forward to the operator `make-fire`. Since fire is a prerequisite for a barbecue, this post–condition is pushed backward to the operator `put steaks on fire` which has `has-fire` as a pre–condition. The system then tries to "improvise" and generate a plan using the state just before the operator `make-fire` to operator `put steaks on fire`. Given that we have a lighter in the current state, this plan can be easily generated. The `light match` operator is replaced by the `use lighter` operator. "Remove side effects" and "Protect effect" are specializations of the Replace operator strategy.

If the non–matching pre–condition does not establish a current goal predicate, the system tries to apply all operators of the plan, substituting variables where necessary (e.g. beans for broccoli). For example, given a plan to make a beef and bean dish from the ingredients, and if the system returns a plan for a beef and broccoli dish, the pre–condition have broccoli does not establish any predicate in the current goal (beef and beans dish). In this case, the system simulates the plan and uses beans instead of broccoli.

After the case has been fixed to achieve all its post–conditions with the new initial conditions, the system tries to achieve missing goal conditions one by one, using means ends analysis. The first non–satisfied term of the goal conjunction is selected and a new planning problem is generated from the goal state of the case to the goal state of the original problem.

The planner computes the subgoals that are necessary for the achievement of any of the adaptations or classical planning rules. It then retrieves similar cases for each of the generated subgoals and tries to work on them in order of similarity. This can

also be used to repair failed plans, if the failed plan is stored in memory with a new post–condition added so that the failure is avoided.

6.2.2 Learning

Learning is extracting useful operators from the current plan derivation, and storing these in a way that makes them accessible by indexing in similar situations. Every time a partial plan is generated, the planner should run a learning algorithm to find out whether the generated plan allows it to optimize the operator set. The dynamic filter introduced above should be used to limit the branching factor of planning. These two heuristics should be used to formulate detailed strategies for the learning phase:

Heuristic 1: Try to create new abstract operators that contain fewer pre–conditions than existing operators, and the same number of post–conditions. Store it as both a case and an abstraction. This has been implemented and shows promising results on the towers of Hanoi problem

Heuristic 2: Try to introduce operators that have fewer post–conditions than existing ones. This generates operators for the substitution heuristics such as "remove side effect" and "protect side effect" above.

Furthermore any adaptation to an existing plan is stored. This will create rules to anticipate failure and to substitute. For example, suppose the first plan fails. Explain the failure, to give an additional post–condition that must be added to the goal. Create an abstract operator that has the initial state and original goal. Link it to the operator that has the initial state and new goal. Now, solve for the new goal.

Substitution of operators means the creation of an abstract operator within the plan that has the old and the new operator as instantiations.

7 Conclusions

The major contribution of this paper is the design of a representation for a planning system, that combines macros, abstraction hierarchies, and case–based planning. The advantage of this approach is that techniques from classical planning as well as case–based planning can be combined in the problem solving process. The paper also describes an analytical dynamic filtering scheme used to rule out inefficient operators. The dynamic filter is based on a formula relating the empirical *usefulness* and the length and branching factor of the operator. The common representation means that the dynamic filter can be applied to abstract operators and cases as well.

The paper also compares three different approaches to the selection of new macros. From this comparison guidelines are suggested for the selection of new operators. The main motivation for these heuristics is to find widely applicable operators with very specific effects.

References