Learning Partial-Order Macros from Solutions

Adi Botea and **Martin Müller** and **Jonathan Schaeffer**
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{adib,mmueller,jonathan}@cs.ualberta.ca

Abstract

Despite recent progress in AI planning, many problems remain challenging for current planners. In many domains, the performance of a planner can greatly be improved by discovering and exploiting information about the domain structure that is not explicitly encoded in the initial PDDL formulation. In this paper we present an automated method that learns relevant information from previous experience in a domain and uses it to solve new problem instances. Our approach produces a small set of useful macro-operators as a result of a training process. For each training problem, a structure called a *solution graph* is built based on the problem solution. Macro-operators with partial ordering of moves are extracted from the solution graph. A filtering and ranking procedure selects the most useful macro-operators, which will be used in future searches. We introduce a heuristic technique that uses only the most promising instantiations of a selected macro for node expansion. Our results indicate an impressive reduction of the search effort in complex domains where structure information can be inferred.

Introduction

AI planning has recently made great advances. The evolution of the international planning competition over its four editions accurately reflects this. Successive editions introduced more and more complex and realistic benchmarks, or harder problem instances for the same domain. Still, the top performers could successfully solve many of the problems each time. Many hard domains, including benchmarks used in the latest competition, still remain a great challenge for the current capabilities of automated planning systems.

In many domains, the performance of a planner can be improved by inferring and exploiting information about the domain structure that is not explicitly encoded in the initial PDDL formulation. In this paper we present an automated method that learns such implicit domain knowledge and uses it to simplify planning for new problem instances. This "hidden" information that a domain encodes is, arguably, proportional to how complex the domain is, and how realistically this models the world. Consider driving a truck between two locations. This operation has many details in the real world. To name just a few, the truck should be fueled and have a driver assigned. In a detailed planning formulation, we would define several operators such as FUEL, ASSIGN-DRIVER, or DRIVE. This representation already contains implicit information about the domain structure. It is quite obvious for a human that driving a truck between two remote locations would be a macro-action where we first fuel the truck and assign a driver (with no ordering constraints between these two actions) and next we apply the drive operator. In a simpler formulation, we can remove the operators FUEL and ASSIGN-DRIVER and consider that, in our model, a truck needs neither fuel nor a driver. Now driving a truck is modeled as a single action, and the structured detail described above is removed from the model.

Our method first learns a set of useful macro-operators, and then uses them in new searches. The learning is based on several training problems from a domain. In traditional planning, the main use of the solving process is to obtain the solution of a given problem instance. In addition, we use both the search process and the solution plan to extract new information about the domain structure. We assume that searching for a plan can provide valuable domain information that could be hard to obtain with only static analysis and no search. For example, it is hard to evaluate beforehand how often a given sequence of actions would occur in a solution, or how many nodes the search algorithm would expand on average in order to discover that action sequence.

For each training problem, a structure called a *solution graph* is built starting from the linear solution sequence that the planner produces. The solution graph contains one node for each step in the solution. Edges model the causal effects that a solution step has on subsequent steps of the linear plan. Analysis of a small set of solution graphs produces a set of *partial-order* macro-operators (i.e., macros with partial ordering of operators) that are likely to be useful in the future. A set of macros is extracted from the solution graphs of all training problems. Then the set is filtered and sorted such that only the most promising macros will be used in the future.

After completing the training, the selected macro-operators are used to speed up future searches in the same domain. Using macro-actions at run-time can potentially introduce the utility problem, which appears when the savings are dominated by the extra costs of macros. The potential savings come from the ability to generate a useful action sequence with no search. On the other hand, macros can increase both the branching factor and the processing cost per node. Many instantiations of a selected macro-operator schema could be applicable to a state, but only a few would actually be shortcuts towards a goal state. If all these instantiations are considered, the induced overhead can be larger than the savings achieved by the useful instantiations. Therefore, the challenge is to utilize for state expansion only a small number of "good" macro instantiations.

To address the utility problem, which is crucial of the overall performance of a search algorithm, we define heuristic techniques such as *helpful macro pruning*. Helpful macro pruning is based on the relaxed plan $RP(s)$ used in FF (Hoffmann & Nebel 2001). In FF, the relaxed plan is used to heuristically evaluate problem states and prune low-level actions in the search space (helpful action pruning). In addition, we use the relaxed plan to prune the set of macro-instantiations that will be used for node expansion. Since actions of the relaxed plan are often useful in the real world, we require that a macro-instantiation m will be used to expand a state s only if m has a minimal number of actions common with $RP(s)$.

The contributions of this paper include an automated technique that learns a small set of useful macro-operators in a domain, and uses them to speed up the search for new problem instances. We introduce a structure called a solution graph that encodes information about the structure of a given problem instance and its domain, and we use this structure to build macro-operators. Using macro-operators in AI planning and search algorithms is by no means a new idea. In our work, we combine the use of macro-operators with modern and powerful techniques such as the relaxed graphplan computation implemented in FF and other top-level planners. We provide experimental results, performance analysis and a detailed discussion of our approach.

We extend our previous work (Botea, Müller, & Schaeffer 2004b) in several directions. First, we increase the applicability from STRIPS domains to ADL domains (Botea *et al.* 2004). Second, the old method generates macros based on component abstraction, which is limited to domains with static predicates in their definition. The current method generates macros from the solution graph, increasing its generality. Third, we increase the size of macros from 2 moves to arbitrary values. Fourth, we generalize the definition of macros allowing partially ordered sequences.

The rest of the paper is structured as follows: The next two sections provide details about building a solution graph, and how to extract and use macro-operators. Next is experimental results and an evaluation of our system. We then briefly review related work and discuss the similarities and differences with our work. The last section shows our conclusions and future work ideas.

Solution Graph

In this section, we describe how to build the solution graph for a problem, starting from the solution plan and exploiting the effects that an action has on the following plan sequence. We first set our discussion framework with some preliminary comments and definitions. Then we present a high-level description of the method, show how this works on an example, and provide the algorithm in pseudo-code.

In the general case, the solution of a planning problem is a partially ordered sequence of steps. When actions have conditional effects, a step in the plan should be a pair (state, action) rather than only an action. This allows us to precisely determine what effects a given action has in the local context. Our implementation handles domains with conditional effects in their actions and can be extended to partial-order plans. However, for simplicity, we assume in the following discussion that the initial solution is a totally-ordered sequence of actions. When an action occurs several times in a solution, each occurrence is a distinct solution step.

To introduce the solution graph, we need to define the causal links in the solution.

Definition 1 *A structure (a_1, p, a_2) is a positive causal link in the solution if: (1) a_1 and a_2 are steps in the solution with a_1 applied earlier than a_2, (2) p is a precondition of a_2 and a positive effect of a_1, and (3) a_1 is the most recent action before a_2 that adds p. We write a positive causal link as $a_1 \xrightarrow{+p} a_2$.*

A structure (a_1, p, a_2) is a negative causal link in the solution if: (4) a_1 and a_2 are steps in the solution with a_1 applied earlier than a_2, (5) p is a precondition of a_2 and a delete effect of a_1, and (6) a_1 is the most recent action before a_2 that deletes p. We write a negative causal link as $a_1 \xrightarrow{-p} a_2$.

We write $a_1 \rightarrow a_2$ if there is at least a causal link (either positive or negative) from a_1 to a_2.

A positive causal link is similar to a causal link in partial-order planning (Nguyen & Kambhampati 2001).

The solution graph is a graph structure that explicitly stores relevant information about the problem extracted from the linear solution. For each step in the linear solution, we define a node in the solution graph. The graph edges model causal links between the solution actions. We define an edge between two nodes a_1 and a_2 if $a_1 \rightarrow a_2$. An edge has two labels: The ADD label is the (possibly empty) list of all facts p so that $a_1 \xrightarrow{+p} a_2$. The DEL label is obtained similarly from the negative causal links.

Figure 1 shows the solution graph for problem 1 in the Satellite benchmark. The graph has 9 nodes, one for each step in the linear solution. Each node contains a numerical label showing the step in the linear solution, the action name and arguments, the preconditions and the effects. We safely ignore static preconditions: no causal link can be generated by a static fact, since such a fact is never part of an action's effects. Graph edges have their ADD labels shown as square boxes, and DEL labels as circles. Consider the edge from node 0 to node 8. Step 0 adds the first precondition of step 8, and deletes the third. Therefore, the ADD label of this edge is 1 (the index of the first precondition), and the DEL label is 3.

A brief analysis of this graph reveals interesting insights about the problem and the domain structure. The action sequence TURN_TO TAKE_IMAGE occurs three times (between steps 3–4, 5–6, and 7–8), which takes 6 out of a total of 9 actions. For each occurrence of this sequence, there is a graph

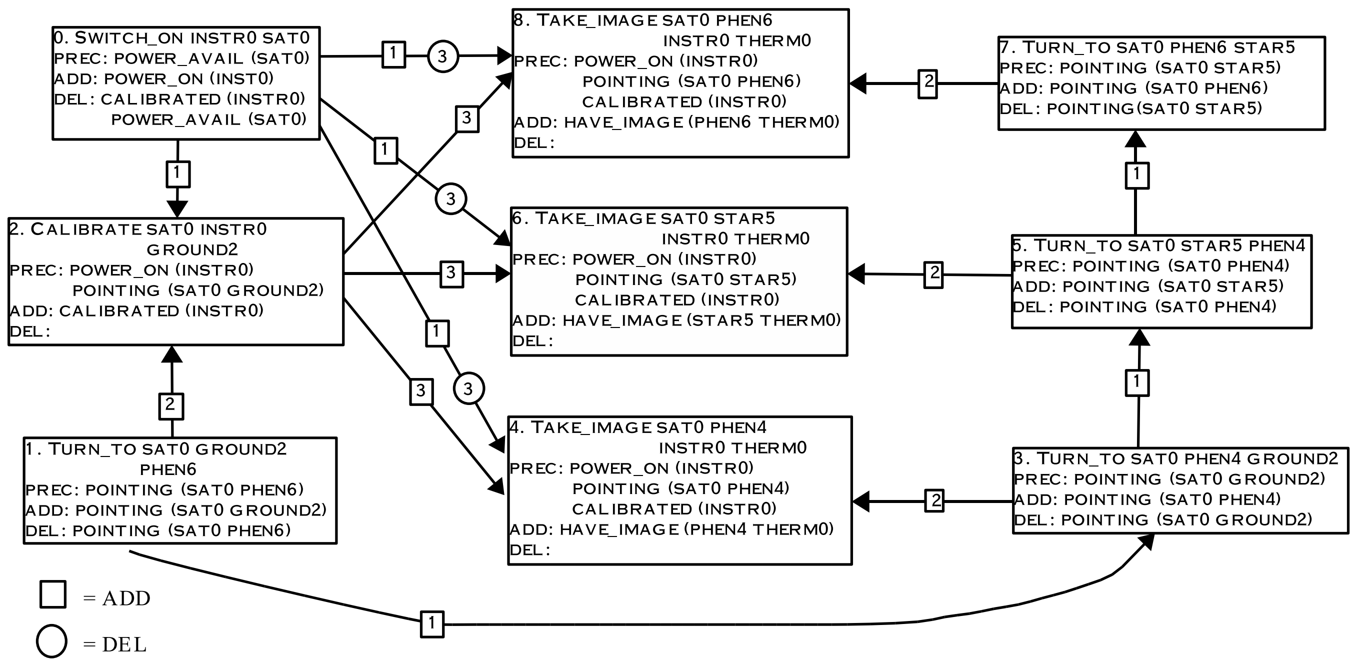

Figure 1: The solution graph for problem 1 in the Satellite benchmark.

edge that shows the causal connection between the actions: applying operator TURN_TO satisfies a precondition of operator TAKE_IMAGE.

Second, the sequence SWITCH_ON TURN_TO CALIBRATE (steps 0–2) is important for repeatedly applying macro TURN_TO TAKE_IMAGE. This sequence makes true two preconditions for each occurrence of operator TAKE_IMAGE. The graph also shows that, after SWITCH_ON has been applied, CALIBRATE should follow, since the latter restores the fact (CALIBRATED INSTR0) which is deleted by the first. Finally, there is no ordering constraint between SWITCH_ON and TURN_TO, so we have a partial ordering of the actions of this sequence.

In this paper we propose automated methods to perform this type of analysis and learn useful information about a domain. The next sections will focus on this idea.

The pseudo-code of building the solution graph is given in Figure 2. The complexity is linear with L, the length of the solution at hand. The methods are in general self explanatory, and follow the high level description provided before. The method findAddActionId(p, id, s) returns the most recent action before the current step id that adds precondition p. The method addEdgeInfo(n_1, n_2, t, f, g) creates a new edge between nodes n_1 and n_2 (if one didn't exist) and concatenates the fact f to the label of type $t \in \{ADD, DEL\}$. The data piece nodes(a) that is used in method buildNodes provides information extracted from the search tree generated while looking for a solution. A search tree has states as nodes and actions as transitions. For each action a in the tree, nodes(a) is the number of nodes expanded in the search right before exploring action a. We further introduce the *node heuristic* (NH) associated to an instantiated macro sequence $m = a_1 ... a_k$ as follows:

$$NH(m) = \text{nodes}(a_k) - \text{nodes}(a_1).$$

NH measures the effort to dynamically discover the given sequence at run-time. As we show in the next section, the node heuristic is used to rank macro-operators in a list.

Macro-Operators

A macro-operator is a structure $m = (O, \prec, \sigma)$ with O a set of domain operators, $\prec$ a partial ordering of the elements in O, and σ a mapping that defines the macro's variables from the operators' variables. A domain operator can occur several times in O.

In this section we focus on how our approach learns and uses macro-operators. Our method is a three-step technique: generation, filtering, run-time instantiation. First, a global set of macros is generated from the solution graphs of several training problems. Second, this set is filtered down to a small set of selected macros, completing the learning phase. Finally, the selected macros are used to speed up planning in new problems.

Generating Macro-Operators

Macros are extracted from the solution graphs of one or more training problems. Our method enumerates and selects subgraphs from the solution graph(s) and builds one macro for each selected subgraph. Two distinct subgraphs can produce the same macro. All generated macros are inserted into a global list that will later be filtered and sorted. The list contains no duplicate elements. When an extracted macro is already part of the global list, relevant information

```
void buildSolutionGraph(Solution $s$, Graph & $g$)
{
    buildNodes($s$, $g$);
    buildEdges($s$, $g$);
}
void buildNodes(Solution $s$, Graph & $g$)
{
    for (int $id = 0$; $id <$ length($s$); ++$id$) {
        Action $a$ = getSolutionStep($id$, $s$);
        addNode($id$, $a$, nodes($a$), $g$);
    }
}
void buildEdges(Solution $s$, Graph & $g$)
{
    for (int $id = 0$; $id <$ length($s$); ++$id$) {
        Action $a$ = getSolutionStep($id$, $s$);
        for (each precondition $p \in$ precs($a$)) {
            $id_{add}$ = findAddActionId($p$, $id$, $s$);
            if ($id_{add}$ != NO_ACTION_ID)
                addEdgeInfo($id_{add}$, $id$, ADD, $p$, $g$);
            $id_{del}$ = findDeleteActionId($s$, $id$, $p$);
            if ($id_{del}$ != NO_ACTION_ID)
                addEdgeInfo($id_{del}$, $id$, DEL, $p$, $g$);
        }
    }
}
```

Figure 2: Pseudo-code for building the solution graph.

associated to that element is updated. For instance, the algorithm increments the number of occurrences, and adds the node heuristic of the extracted instantiation to that of the element in the list.

We present the enumeration and selection process, and then show how a macro is built starting from a given subgraph. Figure 3 presents our method for extracting macros from the solution graph. Parameters MIN_LENGTH and MAX_LENGTH bound the length l of a macro. We set this range from 2 to 10. The minimal size is trivial: each macro should have at least two actions. The upper bound is set to speed up macro generation. In a more general setup, the upper bound could be the plan length of the problem at hand, provided that the whole solution might be an useful macro. Note that this is usually not the case in practice. Our work focuses on identifying a few local patterns that are generally useful, rather than caching many whole solutions of solved problems.

Method selectSubgraphs(l, g, $subgraphList$) finds *valid* subgraphs of size l of the original solution graph. It is implemented as a backtracking procedure that produces all the valid node combinations and early prunes incorrect partial solutions.

To describe the validation rules, we consider a subgraph sg with l arbitrary distinct nodes $a_{m_1}, a_{m_2}, ..., a_{m_l}$. Node a_{m_i} is the m_i-th step in the linear solution. Assume that the nodes are ordered according to their position in the linear solution: $(\forall i < j) : m_i < m_j$. The rules that we impose for sg to be valid are the following:

- The nodes of a valid subgraph are obtained from a sequence of consecutive steps in the linear solution by skipping at most k steps, where k is a parameter. Skipping actions allows irrelevant actions to be ignored for the macro at hand. The upper bound k captures the heuristic that good macros are likely to have their steps "close" together in a solution. Formally, the following formula should stand for a valid macro: $m_l - m_1 + 1 <= l + k$. In our example, consider the subgraph with nodes $\{0, 1, 2, 6\}$. For this subgraph, $l = 4$, $m_l = 6$, and $m_1 = 0$. If $k = 2$, then the subgraph breaks this rule, since $m_l - m_1 + 1 = 7 > 6 = l + k$.

  A small value for k speeds up the computation and reduces the number of generated subgraphs. Unfortunately, occurrences of useful macros might be skipped. However, this seldom happens in practice, as the actions of useful macros usually concentrate together in a local sequence. In our experiments, we set $k = 2$, which was empirically shown as a good trade-off value.

- Positive and negative causal links that the sub-graph edges model are exploited. Consider our example in Figure 1. Nodes 2 and 3 do not form a valid subgraph, since there is no direct link between them, and therefore this subgraph is not connected. However, nodes 3 and 4 are connected through a causal link, so our method will validate this sub-graph. In the general case, we require that a valid subgraph is connected, since we assume that two separated connected components correspond to two independent macros.

- When selecting a subgraph, a solution step a_r ($m_1 < r < m_l$) can be skipped only if a_r is not connected to the current subgraph: $(\forall i \in \{1, .., l\}) : \neg(a_{m_i} \rightarrow a_r \vee a_r \rightarrow a_{m_i})$.

```
void generateAllMacros(Graph $g$, MacroList & $macros$)
{
    for (int $l$ = MIN_LENGTH; $l \leq$ MAX_LENGTH; ++$l$ )
        generateMacros($g$, $l$, $macros$);
}
void generateMacros(Graph $g$, int $l$, MacroList & $macros$)
{
    selectSubgraphs($l$, $g$, $subgraphList$);
    for (each subgraph $s \in subgraphList$) {
        buildMacro($s$, $m$);
        int $pos$ = findMacroInList($m$, $macros$);
        if ($pos$ != NO_POSITION)
            updateInfo($pos$, $m$, $macros$);
        else
            addMacroToList($m$, $macros$);
    }
}
```

Figure 3: Pseudo-code for generating macros.

Method buildMacro(s, m) builds a partially ordered macro m based on the subgraph s which is given as an argument. For each node of the subgraph, the corresponding action is added to the macro. Note that, at this step, actions are instantiated (i.e., they have constant arguments rather than generic variables). After all actions have been added, we replace the constants by generic variables, obtaining a variable identity map σ. The partial order between the operators of the macro is computed using the positive causal links of the subgraph. If a positive causal link exists between two nodes a_1 and a_2, then a precondition of action a_2 was made true by action a_1. Therefore, action a_1 should come before a_2 in the macro sequence. Note that the ordering never has cycles. The ordering constraints are determined using a subgraph of the solution graph, and the solution graph is acyclic. A graph edge can exist from a_1 to a_2 in the solution graph only if a_1 comes before a_2 in the initial linear solution.

From the solution graph in Figure 1, 24 distinct macros are extracted. The largest contains all nodes in the solution graph. One macro occurs 3 times (TURN_TO TAKE_IMAGE), another twice (TURN_TO TAKE_IMAGE TURN_TO), and all remaining macros occur once.

The complexity of generating macros of length l from a solution graph with L actions is $\binom{l+k}{l} \times L \times I$. The first factor is the cost to enumerate macros of length l within a window of size $l + k$ (i.e., a subgraph with $l + k$ nodes that are consecutive in the initial sequential solution – see the first validation rule). We slide the window along the solution plan, obtaining the second factor (we assume $l + k < L$). I is the cost to find/insert a macro into the global set of macros which, in a tree implementation, is logarithmic with the set size.

Filtering and Ranking

The goal of filtering is to address the utility problem, which appears when the overhead of using macros is larger than the savings. After all training problems have been processed, the global list of macros is *statically* filtered and sorted, so that only the most promising macros will be used to solve new problems. When the selected macros are used in future searches, they are further filtered in a *dynamic* process that evaluates their run-time performance.

The static filtering is performed with the so called *overlap rule*. A macro is removed from the list when two occurrences of this macro overlap in a given solution (i.e., the end of one occurrence is the beginning of the other). Consider the following sequence in a solution:

$$...a_1a_2...a_la_1a_2...a_la_1a_2...a_l...$$

Assume both $m_1 = a_1a_2...a_l$ and $m_2 = a_1a_2...a_la_1$ are initially part of the list of macros. When m_1 is used in the search, applying this macro three times could be enough to discover the given sequence. Consider now using m_2 in the search. This macro cannot be applied twice in a row, since the first occurrence ends beyond the beginning of the next occurrence. In effect, the sequence $a_2...a_l$ in the middle has to be discovered with low-level search. Note that a macro that contains two instances of a smaller macro (e.g., $m_3 = m_1m_1 = a_1a_2...a_la_1a_2...a_l$) is still able to generate the whole sequence with no low-level search. For this reason, we do not reject a macro that is a double occurrence of a small (i.e., of length 1 or 2) sequence. The reason why we apply this exception only to small macros is the following. Another important property of the overlap rule is the capacity to automatically limit the length of a macro. In our case, $a_1a_2...a_l$ is kept in the final list, while larger macros such as $a_1a_2...a_la_1$ or $a_1a_2...a_la_1a_2$ are rejected. In Satellite, the macro (TURN_TO TAKE_IMAGE TURN_TO) mentioned before is removed because of the overlap, but the macro (TURN_TO TAKE_IMAGE TURN_TO TAKE_IMAGE), a double occurrence of a short sequence, is kept.

Macros are ranked according to the *total node heuristic* $TNH(m)$ associated to each macro m, with ties broken based on the occurrence frequency. For a generic macro m in the list, $TNH(m)$ is the sum of the node heuristic values (NH) for all instantiations of that macro in the solutions of the training problems. The *average* node heuristic ANH is the total node heuristic divided by the number of occurrences F, and estimates the average search effort needed to discover an instantiation of this macro at run-time:

$$TNH(m) = ANH(m) \times F(m).$$

The total node heuristic is a robust ranking method, which combines into one single rule several factors that can evaluate the performance of a macro. First, since TNH is proportional with F, it favors macros that frequently occurred in the training set, and therefore are more likely to be applicable in the future. Second, TNH directly depends on ANH, which evaluates the search effort that one application of the macro could save.

TNH depends on the search strategy. For instance, changing the move ordering can potentially change the ranking in the macro list. How much the search strategy affects the ranking, and how a set of macros selected based on one search algorithm would perform in a different search algorithm are still open questions for us.

After ranking and filtering the list, only a few elements from the top of the list are kept for future searches. In our Satellite example, the selected macros are (SWITCH_ON TURN_TO CALIBRATE TURN_TO TAKE_IMAGE) and (TURN_TO TAKE_IMAGE TURN_TO TAKE_IMAGE).

Selected macros are further filteried dynamically, based on their run-time performance. Pathologic behaviour of macros at run-time can have several reasons. First, a selected macro m that is never instantiated in a search does not affect the number of expanded nodes, but increases the cost per node. Second, a macro that is instantiated much more often than desired can lead the search into subtrees that contain no goal states. To address these, the following values are accumulated for each macro m: $IN(m)$ is the number of searched nodes where at least one instantiation of m is used for node expansion. $IS(m)$ is the number of times when an instantiation of m is part of a solution. The efficiency rate, $ER(m)$, is $IS(m)$ divided by $IN(m)$. A first implementation of our dynamic filtering procedure evaluates each macro after solving a number of problems NP given as a parameter. If $IN(m) = 0$ or $ER(m)$ does not reach a threshold T, m is removed from the list.

T's value was set based on the empirical observation that there is a gap between the efficiency rate of successful macros and the efficiency rate of macros that should be filtered out. The efficiency rate of successful macros has been observed to range roughly from more than 0.05 to almost 1.00. For pathological macros, ER is less than 0.01. We set $T = 0.03$, which estimates the gap center.

Instantiating Macros at Run-Time

The learned macros are used to speed up the search in new problem instances. A classical search algorithm expands a node by considering low-level actions that can be applied to the current state. In addition, our method adds to the successor list states that can be reached by applying a sequence of actions that compose a macro. This enhancement affects neither the soundness nor the correctness of the original algorithm. When the original search algorithm is complete, we preserve this, since no regular successors that the algorithm generates are deleted. The correctness is guaranteed by our way of applying a macro to a state. Given a state s_0 and a sequence of actions $m = a_1 a_2 ... a_k$, we say that m is applicable to s_0 if a_i can be applied to s_{i-1}, $i \in 0, ..., k$, where $s_i = \gamma(s_{i-1}, a_i)$ and $\gamma(s, a)$ is the state obtained by applying a to s.

When a given state is expanded at run-time, many instantiations of a macro could be applicable, but only a few would actually be shortcuts towards a goal state. If all these instantiations are considered, the branching factor can significantly increase and the induced overhead can be larger than the potential savings achieved by the useful instantiations (the utility problem). Therefore, the challenge is to select for state expansion only a small number of good macro instantiations.

To determine what a "good" instantiation of a macro is, we propose two heuristic methods, called *helpful macro pruning* and *goal macro pruning*. The way these methods are combined depends upon the search strategy employed. We developed our method on top of FF (Hoffmann & Nebel 2001), which uses two search algorithms. Planning starts with Enforced Hill Climbing (EHC), a fast but incomplete algorithm. When EHC fails to find a solution, the search restarts with a complete Best First Search (BFS). In EHC, using only the helpful macro pruning turned out to be effective. In BFS, both helpful macro pruning and goal macro pruning are used.

The goal macro pruning keeps a macro instantiation only if the number of satisfied goal conditions is greater in the destination state as compared to the starting state. Helpful macro pruning is based on the relaxed graphplan computation that FF performs for all evaluated states, and hence is available at no additional cost. For each evaluated state s, FF solves a relaxed problem, where the initial state is the currently evaluated state, the goal conditions are the same as in the real problem, and the actions are relaxed by ignoring their delete effects. This computation produces a relaxed plan $RP(s)$, and FF returns its length as the heuristic evaluation of the current state.

The relaxed plan is used to decide what macro-instantiations to select in a given state. Since actions from the relaxed plan are often useful in the real world, we request that a selected macro and the relaxed plan match partially or totally (i.e., they have common actions). To formally define the matching, consider a macro $m(v_1, ..., v_n)$, where $v_1, ..., v_n$ are variables, and an instantiation $m(c_1, ..., c_n)$, with $c_1, ..., c_2$ constant symbols. We define Match$(m(c_1, ..., c_n), RP(s))$ as the number of actions present in both $m(c_1, ..., c_n)$ and $RP(s)$.

If we require a total matching (i.e., each action of the macro is mapped to an action in the relaxed plan) then we will often end up with no selected instantiations, since the relaxed plan can be too optimistic and miss actions needed in the real solution. On the other hand, a loose matching can significantly increase the number of selected instantiations, with negative effects on the overall performance of the search algorithm. Our solution is to select only the instantiations with the best matching seen so far for the given macro in the given domain. We select a macro instantiation only if

$$\text{Match}(m(c_1, ..., c_n), RP(s)) \geq \text{MaxMatch}(m(v_1, ..., v_n)),$$

where MaxMatch$(m(v_1, ..., v_n))$ is the largest value of Match$(m(c'_1, ..., c'_n), RP(s'))$ that has been encountered so far in this domain, for any instantiation of this macro and for any state. Our experiments show that MaxMatch$(m(v_1, ..., v_n))$ quickly converges to a stable value. In our example, MaxMatch(SWITCH_ON TURN_TO CALIBRATE TURN_TO TAKE_IMAGE) converges to 4, and MaxMatch(TURN_TO TAKE_IMAGE TURN_TO TAKE_IMAGE) converges to 3.

Discussion

Desired properties of macros and trade-offs involved in combining them into a filtering method are discussed in (McCluskey & Porteous 1997). The authors identify five factors that can be used to predict the performance of a macro set. In the next paragraphs we briefly discuss how our system deals with each factor.

TNH includes the first two factors ("the likelihood of some macro being usable at any step in solving any given planning problem", and "the amount of processing (search) a macro cuts down"). Factor 3 ("the cost of searching for an applicable macro during planning") mainly refers to the additional cost per node in the search algorithm. At each node, and for each macro-schema, a check should be performed if instantiations of the macro-schema are applicable to the current state, and satisfy the macro pruning tests. We greatly cut the costs by keeping only a small list of macro-schemas, but there often is an overhead as compared to searching with no macros. We take no special care of factor 4 ("the cost (in terms of solution non-optimality) of using a macro"). The next section gives an analysis of factors 3 and 4.

Factor 5 refers to "the cost of generating and maintaining the macro set". Our costs to generate macros include, for each training problem, solving the problem instance, building the solution graph, extracting macros from the solution graph, and inserting the macros into the global list. Solving the problem instance dominates the theoretical complexity of processing one training problem. The only maintenance operations that our method performs are to dynamically filter the list of macros and to update MaxMatch for each macro-schema, which need no significant cost.

Experimental Results

We use as testbeds domains that we competed in as part of the fourth international planning competition IPC-4: Promela Dining Philosophers – ADL (containing a total of 48 problems), Promela Optical Telegraph – ADL (48 problems), Satellite – STRIPS (36 problems), PSR Middle Compiled – ADL (50 problems), Pipesworld Notankage Nontemporal – STRIPS (50 problems), Pipesworld Tankage Nontemporal – STRIPS (50 problems), and Airport – ADL (50 problems). See (Hoffmann *et al.* 2004) for details about these domains. We present detailed performance analysis for the first four domains, and briefly comment on the remaining three. The experiments ran on an AMD Athlon 2 GHz machine, with limits of 30 minutes and 1 GB of memory for each problem.

Figures 4 – 7 show three data curves each. The curves are not cummulative: each data point shows a value corresponding to one problem in the given domain. The horizontal axis preserves the problem ordering as in the competition domains. The data labeled with "Classical" are obtained with FF version 2.3 (Hoffmann & Nebel 2001) plus several implementation enhancements (Botea *et al.* 2004), but no macro-operators. "PO Macros" corresponds to a planner that implements the ideas described in this paper on top of "Classical". "IPC-4" shows results with the planner that we used in IPC-4. This planner also uses macro-operators extracted from solutions, but it was restricted to macros of length 2, eliminating many of the challenges that have to be addressed for longer macros. The work we describe in this paper is an extension of that preliminary model, and hence it is relevant to see an empirical performance comparison.

Figures 4 and 5 show the results for Promela Dining Philosophers and Optical Telegraph. Note that, in the competition, our planner won this version of Promela Optical Telegraph. The new extended model leads to massive improvement. For instance, in Dining Philosophers, each problem is solved within less than 1 second, while expanding less than 200 nodes. In addition, in both domains, our new system outperforms by far the top performers in the competition for the same domain versions. We found no difference in terms of solution quality between "Classical" and "PO Macros".

Let the *cost rate* be the cost per node in "PO Macros" divided by the cost per node in "Classical". We collected statistics about the cost rate from problems solved by *both* planners. In Optical Telegraph, the cost rate varies between 1.40 and 1.47, with an average of 1.43. Since problems in Dining Philosophers are solved very easily (e.g., 33 nodes in 0.01 seconds) in the "PO Macros" setup, it is hard to obtain accurate statistics about the cost rate. The reason is that the reported CPU time always has a small amount of noise partly caused by truncation to two decimal places. When the total time is small too, the noise significantly affects the statistics accuracy.

Figure 6 summarizes our experiments in Satellite. In the competition results for this domain, our planner Macro-FF and YAHSP (Vidal 2004b) tied for the first place (with better average performance for YAHSP over this problem set). Our new model further improves our result, going up to about one order of magnitude speedup as compared to classical search. In Satellite, the heuristic evaluation of a state becomes more expensive as problems grow in size, with interesting effects for the system performance. The rate of the extra cost per node that macros induce is greater for small problems, and gradually decreases for larger problems since, in large problems, the heuristic evaluation dominates in cost all remaining processing per searched node. The cost rate varies from 0.83 to 2.04 and averages 1.14. Interestingly, the cost rate is sometimes less than 1. This is because other costs such as maintaining the open and closed queues grow with the number of nodes, and the growth rate can be greater than linear. The solution quality slightly varies in both directions, with no significant impact for the system performance.

Figure 7 shows our experiments in PSR Middle Compiled. Partial-order macros solve 33 problems, as compared to 32 problems in "Classical" and 31 in "IPC-4". In terms of expanded nodes, partial-order macros often achieve significant savings, but never result in more expanded nodes. For this problem set, the number of nodes expanded in "Classical" is an upper bound for the number of nodes in "PO Macros". This is mainly due to the goal macro pruning rule, which turned out to be very selective in PSR. There are problems where the number of expanded nodes is exactly the same in both setups, suggesting that no macro was instantiated at run-time. The cost rate averages 1.39, varying between 1.01 and 1.87. There is no significant decay in the plan quality.

Compared to the previous three testbeds, the performance improvement in PSR is rather limited. We believe that part of the explanation is that our definition of macro equivalence is too relaxed and misses useful structural information in PSR. When checking if two action sequences are equivalent, our current algorithm considers the set of operators, their partial ordering, and the variable binding. The algorithm ignores whether conditional effects are activated correspondingly in the two compared sequence instantiations. However, in PSR, conditional effects encode a significant part of the local structure of a solution. There are operators with zero parameters but rich lists of conditional effects (e.g., operator AXIOM). Further exploration of this insight is left as future work.

In Pipesworld, the generated macros have a very small efficiency rate ER, and the dynamic filtering drops all of them, reducing the search to the classical algorithm. We ran no experiments in Airport. In the ADL version of this domain, the classical algorithm quickly solves the first 20 problems, leaving little room for further improvement. The pre-processing phase of the remaining problems is so hard that only one more instance can be solved within the given constraints. We could not use the STRIPS version to test our method either. In STRIPS Airport, each problem instance has its own domain definition, whereas our learning method requires several training problems for one domain.

An important problem that we want to address is to evaluate in which domains our method works well, and in which classes of problems this approach is less effective. We identify several factors that affect our method performance in

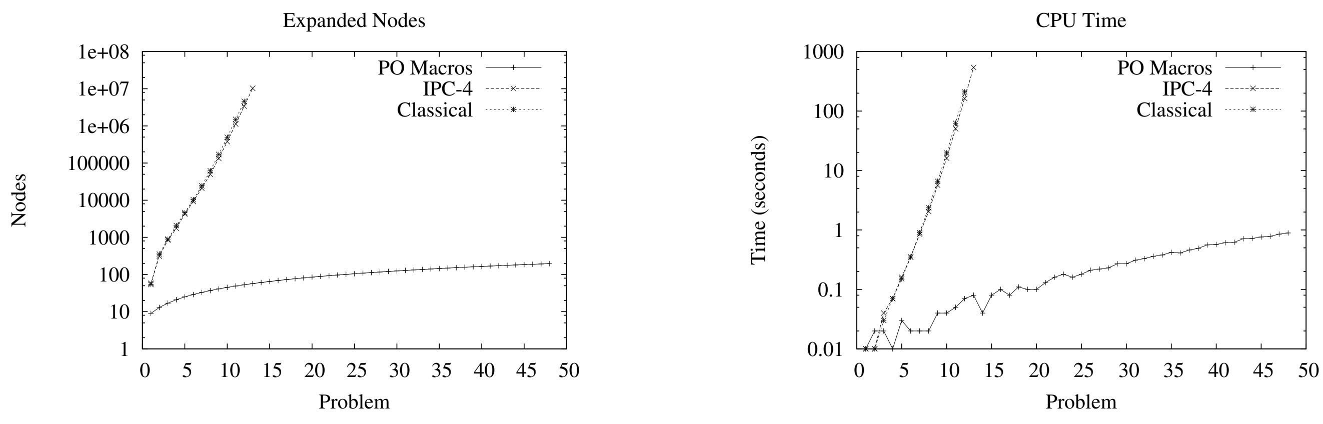

Figure 4: Experimental results in Promela Dining Philosophers.

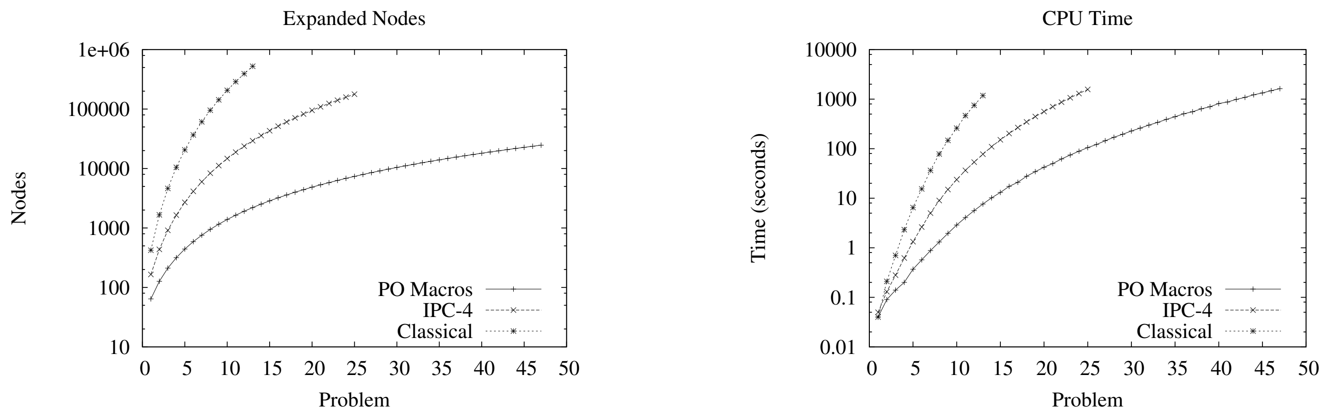

Figure 5: Experimental results in Promela Optical Telegraph.

terms of search effort in a domain. The first factor is the efficiency of the macro pruning rules, which control the set of macro instantiations at run-time and influences the planner performance. An efficient pruning keeps only a few instantiations that are shortcuts to a goal state (a single such instantiation will do). The performance drops when more instantiations are selected, and many of them lead to subtrees that contain no goal states. The efficiency of helpful macro pruning directly depends on the quality of both the relaxed plan associated with a state, and the macro-schema that is being instantiated. Since the relaxed plan is more informative in Promela and Satellite than in PSR or Pipesworld, the performance of our approach is significantly better in the former applications.

Second, our experience suggests that our method performs better in "structured" domains rather than in "flat" benchmarks. Intuitively, we say that a domain is more structured when more local details of the domain in the real world are preserved in the PDDL formulation. In such domains, local move sequences occur over and over again, and our method can catch these as potential macros. In contrast, in a "flat" domain, such a local sequence is often replaced with one single action by the designer of the PDDL formulation.

Third, the search strategy seems to be important too. Enforced Hill Climbing is successful in Promela Dining Philosophers, Promela Optical Telegraph, and Satellite, where we have achieved great performance improvement. Best First Search has to be launched in PSR and many instances of Pipesworld when hill climbing is perceived to have failed. More analysis of this issue is left as future work.

Related Work

Early work on macro-operators in AI planning includes (Fikes & Nilsson 1971). As in our approach, macros are extracted after a problem was solved and the solution became available. (Minton 1985) advances this work by introducing techniques that filter the set of learned macro-operators. In his approach, two types of macro-operators are preferred: S-macros, which occur with high frequency in problem solutions, and the T-macros, which can be useful but have low-priority in the original search algorithm. In (Iba 1989) macro-operators are generated at run-time using the so-called peak-to-peak heuristic. A macro is a move sequence between two peaks of the heuristic state evaluation.

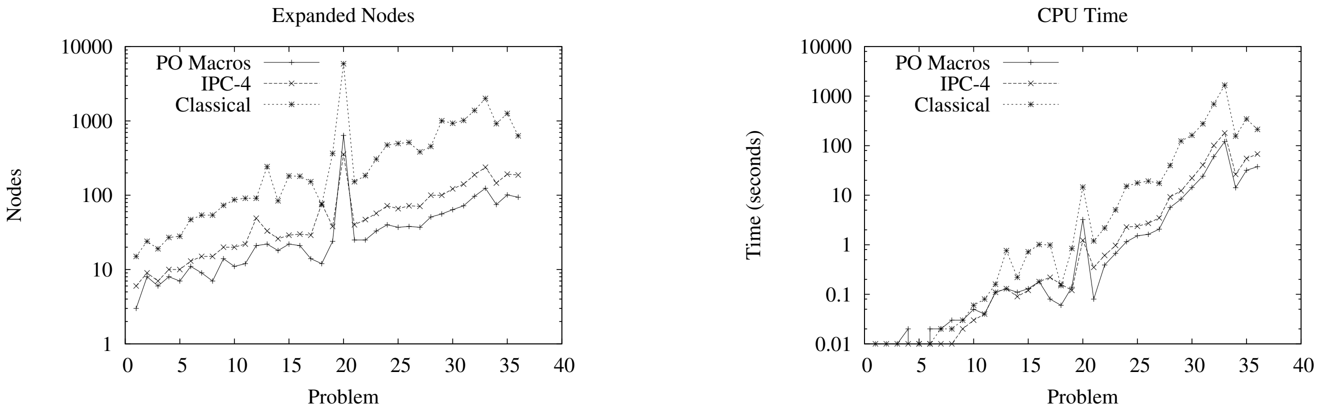

Figure 6: Experimental results in Satellite.

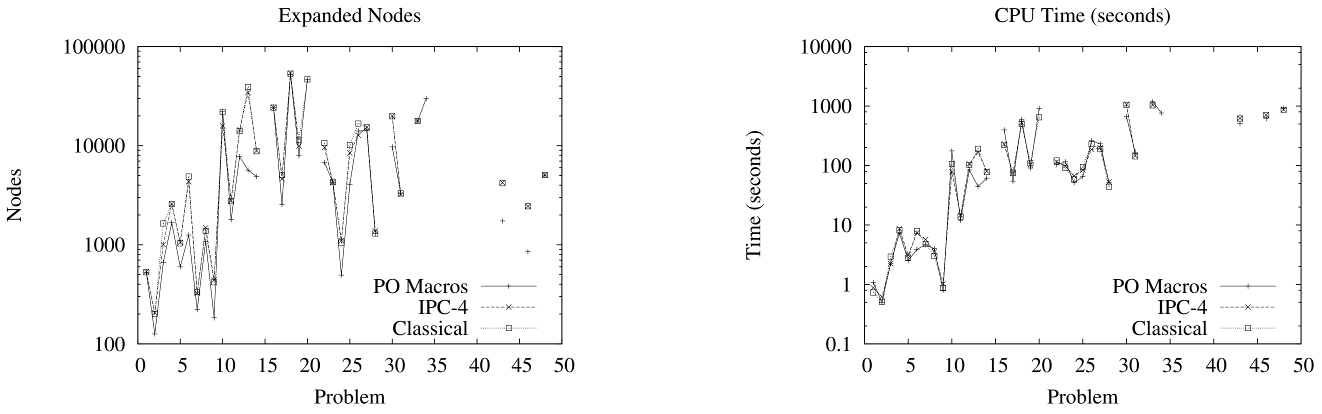

Figure 7: Experimental results in PSR Middle Compiled.

Such a macro traverses a "valley" in the search space, and using this later can correct the heuristic evaluation. A macro filtering procedure uses both simple static rules and dynamic statistical data. As in our work, (Mooney 1988) introduces partial ordering of operators in a macro based on their causal interactions. This work considers whole plans as macros, whereas we focus on local macro-sequences.

Work on improving planning based on solutions of similar problems includes (Veloso & Carbonell 1993; Kambhampati 1993). Solutions of solved problems annotated with additional relevant information are stored for later use. This additional information contains explanations of successful or failed search decisions in (Veloso & Carbonell 1993), and the causal structure of solution plans in (Kambhampati 1993). These approaches further define several similarity metrics between planning problems. When a new problem is fed to the planner, the annotated solutions of similar problems are used to guide the current planning process.

(McCluskey & Porteous 1997) focus on constructing planning domains starting from a natural language description. The approach combines human expertise and automatic tools, and addresses both correctness and efficiency of the obtained formulation. Using macro-operators is a major technique that the authors propose for efficiency improvement. In this work, a state in a domain is composed of local states of several variables called dynamic objects. Macros model transitions between the local states of a variable.

Planner Marvin (Coles & Smith 2004) generates macros both online (as plateau-escaping sequences) and offline (from a reduced version of the problem to be solved). No macros are cached from one problem instance to another.

Other methods that exploit at search time the relaxed graphplan associated to a problem state (Hoffmann & Nebel 2001) include helpful action pruning (Hoffmann & Nebel 2001) and look-ahead policies (Vidal 2004a). Helpful action pruning considers for node expansion only actions that occur in the relaxed plan and can be applied to the current state. Helpful macro pruning applies the same pruning idea for the macro-actions applicable to a state, with the noticeable difference that helpful macro pruning does not give up completeness of the search algorithm.

A lookahead policy executes parts of the relaxed plan in the real world, as this often provides a path towards a goal state with no search and few states evaluated. The actions in the relaxed plan are heuristically sorted and iteratively applied as long as this is possible. When the lookahead procedure cannot be continued with actions from the relaxed plan, a plan-repair method selects a new action to be applied, so the procedure can resume.

Application-specific implementations of macro-actions include work on the sliding-tile puzzle (Korf 1985; Iba 1989). Using macro-moves to solving Rubik's Cube puzzles is proposed in (Hernádvölgyi 2001). Two of the most effective concepts used in the Sokoban solver *Rolling Stone*, tunnel and goal macros, are applications of this idea (Junghanns & Schaeffer 2001). More recent work in Sokoban includes decomposition of a maze into a set of rooms connected by tunnels, with macro-moves defined for internal processing of rooms and tunnels (Botea, Müller, & Schaeffer 2002). In (Botea, Müller, & Schaeffer 2004a), a navigation map is automatically decomposed into a set of clusters, possibly on several abstraction levels. For each cluster, an internal optimal path is pre-computed between any two entrances of that cluster. Path-finding is performed at an abstract level, where a macro-move crosses a cluster from one entrance to another in one step.

Conclusion

Despite the great progress that AI planning has recently achieved, many domains remain challenging for current planners. In this paper we presented a technique that automatically learns a small set of macro-operators from previous experience in a domain, and uses them to speed up the search in future problems. We evaluated our method on standard benchmarks, showing significant improvement for domains where structure information can be inferred.

Exploring our method more deeply and improving the performance in more classes of problems are major directions for future work. We also plan to extend our approach in several directions. Our learning method can be generalized from macro-operators to more complex structures such as hierarchical task networks. Little research focusing on learning such structures has been conducted, even though the problem is of great importance.

Another interesting topic is to use macros in the graphplan algorithm, rather than our current framework of planning as heuristic search. The motivation is that a solution graph can be seen as a subset of the graphplan associated to the initial state of a problem. Since we learn common patterns that occur in solution graphs, it seems natural to try to use these patterns in a framework that is similar to solution graphs.

We will explore how a heuristic evaluation based on the relaxed graphplan can be improved with macro-operators. Our previous work, where a STRIPS macro is added to the original domain formulation as a regular operator, shows the potential of this idea. In this framework, macros are considered in the relaxed graphplan computation just like any other operator, and they may lead to more accurate evaluations.

References

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2004. Macro-FF. In *4th International Planning Competition*, 15–17.

Botea, A.; Müller, M.; and Schaeffer, J. 2002. Using Abstraction for Planning in Sokoban. In *3rd International Conference on Computers and Games (CG'2002)*.

Botea, A.; Müller, M.; and Schaeffer, J. 2004a. Near Optimal Hierarchical Path-Finding. *Journal of Game Development* 1(1):7–28.

Botea, A.; Müller, M.; and Schaeffer, J. 2004b. Using Component Abstraction for Automatic Generation of Macro-Actions. In *ICAPS-04*, 181–190.

Coles, A., and Smith, A. 2004. Marvin: Macro Actions from Reduced Versions of the Instance. In *4th International Planning Competition*, 24–26.

Fikes, R. E., and Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 5(2):189–208.

Hernádvölgyi, I. 2001. Searching for Macro-operators with Automatically Generated Heuristics. In *14th Canadian Conference on Artificial Intelligence*, 194–203.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR* 14:253–302.

Hoffmann, J.; Edelkamp, S.; Englert, R.; Liporace, F.; Thiébaux, S.; and Trüg, S. 2004. Towards Realistic Benchmarks for Planning: the Domains Used in the Classical Part of IPC-4. In *4th International Planning Competition*, 7–14.

Iba, G. A. 1989. A Heuristic Approach to the Discovery of Macro-Operators. *Machine Learning* 3(4):285–317.

Junghanns, A., and Schaeffer, J. 2001. Sokoban: Enhancing Single-Agent Search Using Domain Knowledge. *Artificial Intelligence* 129(1–2):219–251.

Kambhampati, S. 1993. *Machine Learning Methods for Planning*. Morgan Kaufmann. chapter Supporting Flexible Plan Reuse, 397–434.

Korf, R. E. 1985. Macro-operators: A weak method for learning. *Artificial Intelligence* 26(1):35–77.

McCluskey, T. L., and Porteous, J. M. 1997. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. *Artificial Intelligence* 95:1–65.

Minton, S. 1985. Selectively Generalizing Plans for Problem-Solving. In *IJCAI-85*, 596–599.

Mooney, R. 1988. Generalizing the Order of Operators in Macro-Operators. In *Fifth International Converence on Machine Learning ICML-88*, 270–283.

Nguyen, X., and Kambhampati, S. 2001. Reviving Partial Order Planning. In Nebel, B., ed., *IJCAI-01*, 459–466.

Veloso, M., and Carbonell, J. 1993. *Machine Learning Methods for Planning*. Morgan Kaufmann. chapter Toward Scaling Up Machine Learning: A Case Stydy with Derivational Analogy, 233–272.

Vidal, V. 2004a. A Lookahead Strategy for Heuristic Search Planning. In *ICAPS-04*, 150–159.

Vidal, V. 2004b. The YAHSP Planning System: Forward Heuristic Search with Lookahead Plans Analysis. In *4th International Planning Competition*, 56–58.