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 benchmarks 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, we build a structure called a *solution graph* 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. We demonstrate our ideas in standard domains used in the recent AI planning competitions. Our results indicate an impressive potential to reduce 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 (McDermott 2000; Bacchus 2001; Long & Fox 2003; Hoffmann *et al.* 2004) 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. However, 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. We extend this and 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 in average in order to discover that action sequence.

For each training problem, we build a structure called a *solution graph* 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. We analyze a small set of solution graphs to produce a set of *partial-order* macro-operators (i.e., macros with partial ordering of operators) that are likely to be useful in the future. First, we extract a set of macros from the solution graphs of all training problems. Next, we filter and sort the set and select for future use only the most promising macros.

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 is a challenging

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

task. In terms of single-agent search, macros could lead to either reduced search effort or performance overhead, and the way we balance these is crucial for the overall performance of a search algorithm. The potential savings come from the ability to generate a useful action sequence with no search. However,, macros can increase both the branching factor and the processing cost per node. In terms of AI planning, 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 we consider all these instantiations, 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 this, we introduce a heuristic technique called *helpful macro pruning*, 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 request 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 speedup the search for new problem instances. We introduce a new 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 extend the classical definition of a macro-operator, allowing a partial ordering of its operators, and 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.

The rest of the paper is structured as follows: The next two sections provide details about how we build a solution graph, and how we extract and use macro-operators. We continue with presenting experimental results and evaluating the performance 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 easily be extended to partial-order plans. However, for the sake of 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 positive causal link is similar to a causal link in partial-order planning (Nguyen & Kambhampati 2001).*

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 the there is at least a causal link (either positive or negative) from a_1 to a_2:

$$a_1 {\rightarrow} a_2 \Leftrightarrow \exists p : a_1 \xrightarrow{+p} a_2 \vee a_1 \xrightarrow{-p} a_2.$$

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. [1] 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, and the complete list of preconditions and effects. 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 (i.e., 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 edge that shows the causal connection between the actions:

[1]The one-to-one mapping between solution steps and graph nodes allows us to use the name of one for the other when this is not ambiguous.

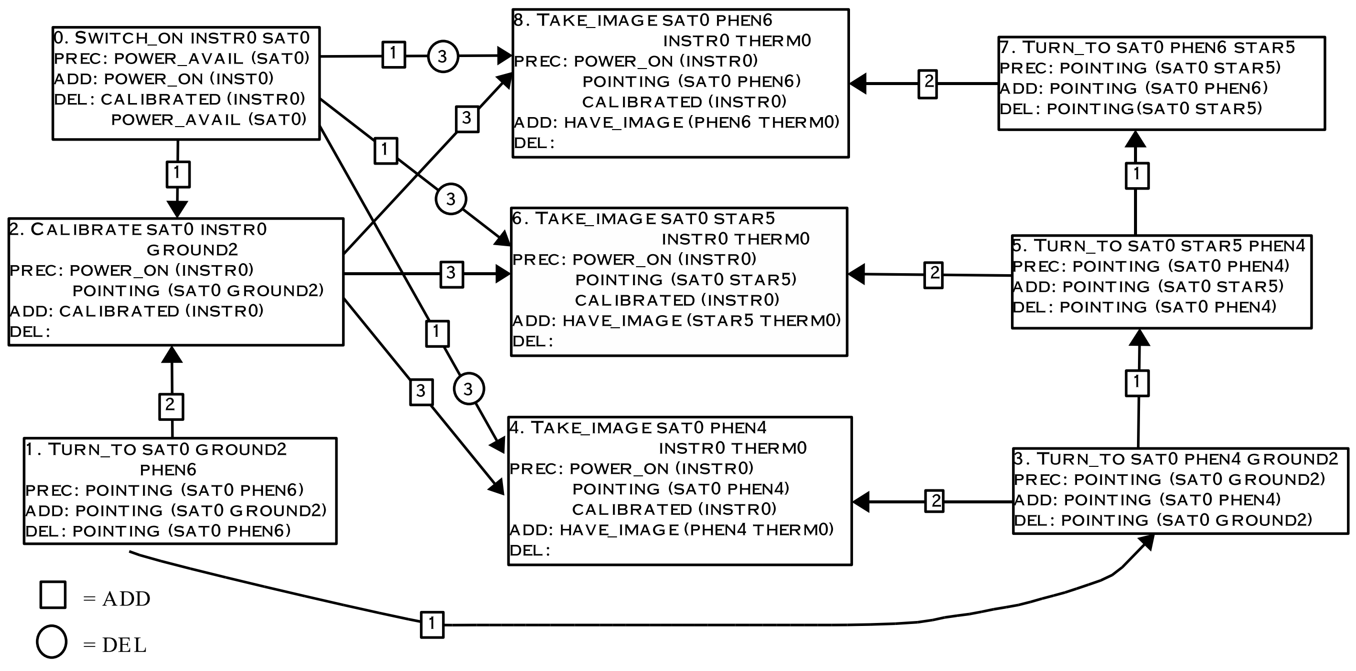


Figure 1: The solution graph for problem 1 in the Satellite benchmark.

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, we have to apply CALIBRATE, 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 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 f to the corresponding label 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. For each transition step 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*, which will be used to sort the macros in a domain. Given an instantiated macro sequence $m = a_1 ... a_k$, the node heuristic is defined as

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

and measures the effort to dynamically discover the given sequence at run-time. As we show in the next section, we use this piece of data 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 binding for 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, the global 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

We extract macros 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. We insert all the generated macros 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 associated to that element is updated. For instance, we increment the number of occurrences, and add the node heuristic of the extracted instantiation to the node heuristic of the element in the list.

```
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.

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. It is called with a length l parameter, the length of the macro to build.

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

To describe 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 conditions 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. 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 $m_l - m_1 + 1 = 7 > 6 = l + k$.

  The main goal of this condition is to greatly speed up the computation and reduce the number of generated subgraphs. Unfortunately, the method might skip occurrences of useful macros. However, this seldom happens in practice, as the actions of useful macros are usually concentrated together in a local sequence.

```
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 the macros.

- We exploit both positive and negative causal links that the sub-graph edges model. 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, we may skip a solution step a_r with $m_1 < r < m_l$ only if a_r is not connected to the current subgraph: $(\forall i \in \{1, .., l\}) : \neg(a_{m_i} \rightarrow a_r \vee a_r \rightarrow a_{m_i})$.

The 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 the action a_2 was made true by the action a_1. Therefore, the 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 sub-graph of the solution graph, and the solution graph is acyclic. A graph edge can exist from node a_1 to node a_2 in the solution graph only if a_1 comes before a_2 in the initial linear solution.

Our method extracts 24 distinct macros from the solution graph shown in Figure 1. 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.

Filtering and Ranking

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 using a criterion called the *overlap rule*. We remove a macro 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). To motivate this rule, 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 the search, applying this macro three times could be enough to discover the given sequence. Consider now using m_2 in the search. We cannot apply this macro 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 occurence of a small (i.e., of length 1 or 2) macro. We apply this exception only to small macros because 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 from the list because of the overlap, but the macro (TURN_TO TAKE_IMAGE TURN_TO TAKE_IMAGE), which is a double occurence of a small macro, is kept.

We rank the macros 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:

$$THN(m) = ANH(m) \times f(m).$$

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.

On the other hand, 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, we keep for future use only a few elements from the top of the list. 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).

The goal of dynamic filtering is to correct performance bottlenecks that a macro could produce. First, a selected macro m that is never instantiated in 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, we accumulate the following values for each macro m: $IN(m)$ is the number of search 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 is $ER(m)$ 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 exceed a threshold T, we drop m from the list.

Instantiating Macros at Run-Time

The macros obtained in the learning phase 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, we add 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 we delete no regular successors that the algorithm generates. The correctness is guaranteed by our way of applying a macro to a state. Given a state s_0 and a sequence of actions $M = a_1a_2...a_k$, 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.

Using macro-actions can lead to either reduced search effort or performance overhead, and the way we balance these is crucial for the overall performance of a planner. When a macro-operator is applied at run-time, the corresponding sequence of actions is added to the current path with little effort, since we prune the whole sub-tree that normal search would expand to discover the given action sequence. On the other hand, using macros increases the branching factor of the search space, which could lead to significantly larger searches.

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 we consider all these instantiations, the branching factor can significantly increase and the induced overhead can be larger than the potential savings achieved by the useful instantiations. 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 a heuristic method, called *helpful macro pruning*, which is based on the relaxed graphplan computation that FF (Hoffmann & Nebel 2001) performs for heuristic state evaluation. 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.

We use the relaxed plan 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 $a \in M(c_1, ..., c_n)$ so that $R(a) \in RP(s)$, where $R(a)$ is the relaxed version of a.

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 optimistic and miss actions needed in the real solution. However, 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 we have 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.

Our total node heuristic 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, we have to check if instantiations of the macro-schema are applicable to the current state, and satisfy the helpful macro pruning rule. We greatly cut the costs by keeping only a small list of macro-schemas, but there always is an overhead as compared to searching with no macros. The average overhead rate can be obtained by comparing the node speed-up rate vs. the CPU time speed-up rate in the next section.

We take no special care of factor 4 ("the cost (in terms of solution non-optimality) of using a macro"). See the next section for an analysis of how macros affect the quality of solutions in our benchmarks. Factor 5 refers to "the cost of generating and maintaining the macro set". The 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

Setup

We evaluate our method on the same benchmarks that we competed in the fourth international planning competition IPC-4 (Hoffmann *et al.* 2004): 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), Airport – ADL (50 problems), Pipesworld Notankage Nonemporal – STRIPS (50 problems), and Pipesworld Tankage Nontemporal – STRIPS (50 problems).

We ran the experiments on an AMD Athlon 2 GHz machine, within the limits of 30 minutes and 1 GB of memory for each problem. We compare the performance of three planning systems in terms of expanded nodes and CPU time. Macro-FF version 1.1 adds to the original FF version 2.3 (Hoffmann & Nebel 2001) several implementation enhancements, but no functionality for macro-operators. Macro-FF version 1.2, which was used in the competition, extends the previous version with support for macro-operators. Macro-FF version 1.3 preserves the implementation enhancements, but extends the old macro functionality to the model described in this paper. Discussing the implementation enhancements is not the goal of this paper (see (Botea *et al.* 2004) for details). Neither is evaluating their impact on the system performace. We only mention that they were very useful in domains such as Promela Optical Telegraph (increasing the number of solved problems from 3 to 13 within the constraints shown before, with no macros used) and PSR Middle Compiled (improving from 20 to 32 solved problems, under the same conditions).

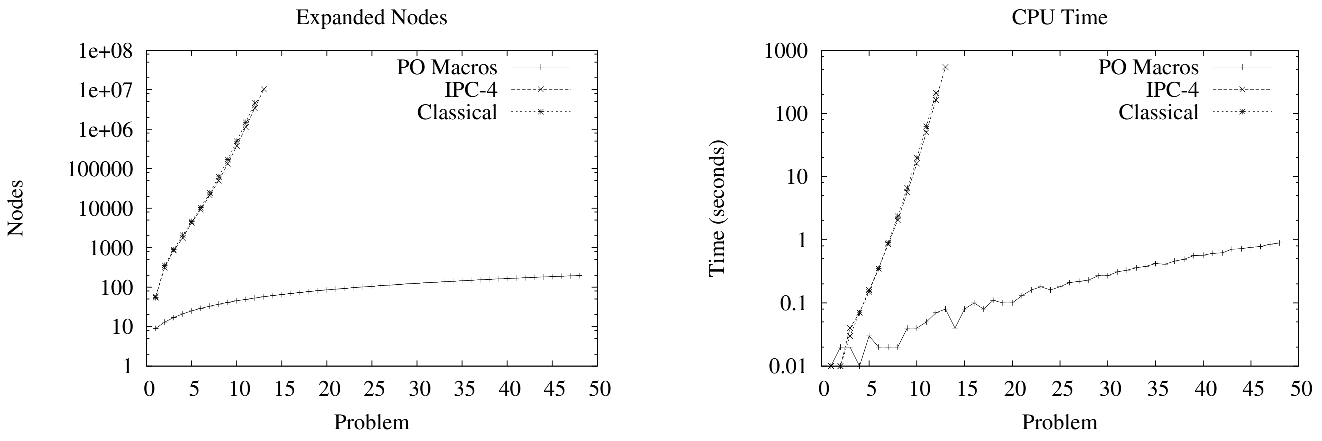


Figure 4: Experimental results in Promela Dining Philosophers.

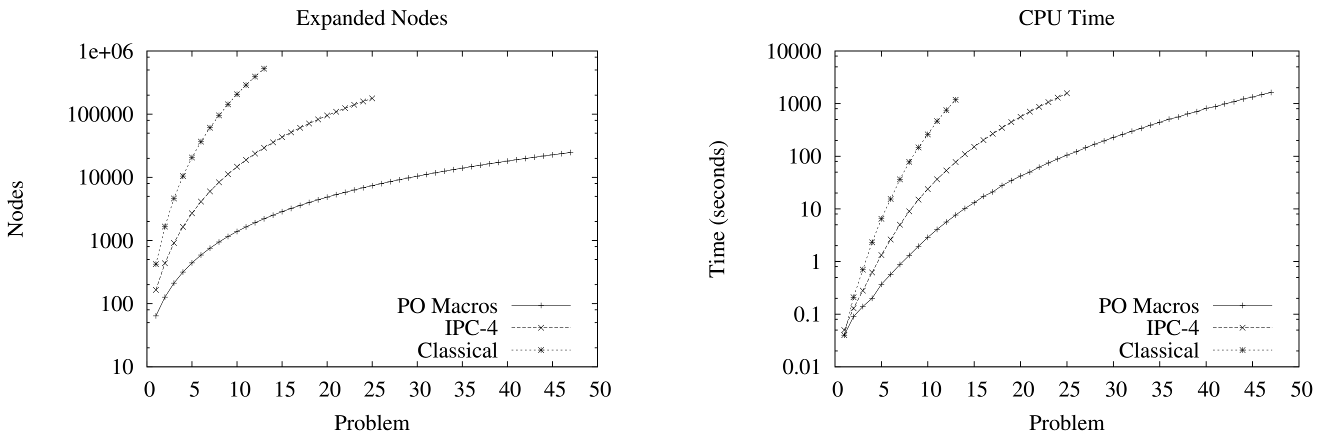


Figure 5: Experimental results in Promela Optical Telegraph.

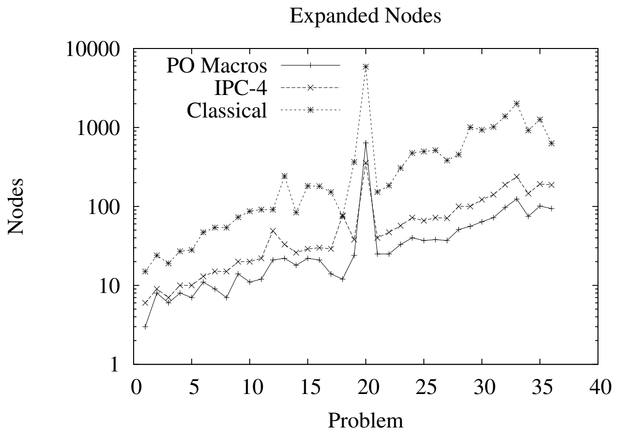
Analysis

In Figures 4, 5, and 6, the data labeled with "Classical" are obtained with Macro-FF 1.1, "IPC-4" shows the results for Macro-FF 1.2, and "PO Macros" corresponds to Macro-FF 1.3. Figures 4 and 5 show the results for Promela Dining Philosophers and Optical Telegraph. Note that, in the competition, Macro-FF 1.2 won this version of Promela Optical Telegraph. Using partial-order macros 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 system with partial-order macros outperforms by far the top performers in the competition for the same domain versions. In these domains, using macros significantly increase the cost per node, but no degradation in solution quality is observed.

Figure 6 summarizes our experiments in Satellite. In the competition results for this domain, Macro-FF 1.2 and YAHSP (Vidal 2004b) have tied for the first place (with better average performance for YAHSP over this problem set). Macro-FF 1.3 further improves our result, going up to about one order of magnitude speed up as compared to classical search. In Satellite, the heuristic evaluation of a state becomes more and more expensive as problems grow in size, with interesting effects for the system performance. First, 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 heavily dominates in cost all remaining processing per searched node. Second, the correlation between the number of expanded nodes and the CPU time varies over the problem set. For instance, problem 20 is the hardest to solve in terms of expanded nodes, but takes less time than problems that are larger in size. The solution quality slightly varies in both directions, with no significant impact for the system performance.

xxxx The largest missing part is analysis of the remaining domains: PSR, Pipesworld, Airport. I'll add this later, as I plan to explore it in the remaining days. xxxx.

An important problem that we want to address is to evaluate in which classes of problems our method works well, and



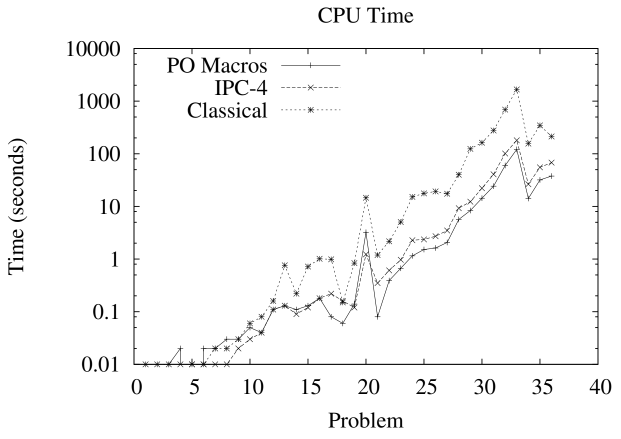


Figure 6: Experimental results in Satellite.

in which domains this approach is less effective. We identify several factors that affect our method performance in terms of search effort in a domain. The first factor is the efficiency of helpful macro pruning. This pruning rule controls the set of macro instantiations at run-time, and greatly influences the planner performance. An efficient pruning keeps only a few instantiations that are shortcuts to a goal state (one 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 to a state, and the macro-schema that is being instantiated. Since the relaxed plan is more informative in Promela and Satellite than in PSR, the performance of our approach is significantly better in the first domains.

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.

xxxx Third, the search strategy is important too. All three successful domains use Enhanced Hill Climbing, while PSR uses Best First Search. I hope I'll add some last minute comments here, depending on how the results in psr end up.xxxx

Related Work

The related work described in this section falls into two categories. We first review planning approaches that make use of the domain structure to reduce the complexity of planning, and next consider previous work on macro-operators.

An automatic method that discovers and uses the domain structure has been explored in (Knoblock 1994). In this work, a hierarchy of abstractions is built starting from the initial low-level problem description. A new abstract level is obtained dropping literals from the problem definition at the previous abstraction level. Refining an abstract plan never causes backtracking because of changes in the plan structure. Backtracking is performed only when an abstract plan has no refinement. (Bacchus & Yang 1994) define a theoretical probabilistic framework to analyze the search complexity in hierarchical models. The authors also use some concepts of that model to improve Knoblock's abstraction algorithm. More recently, implicit causal structure of a domain has been used to design a domain-independent heuristic for state evaluation (Helmert 2004). These methods either statically infer information about the structure of a domain, or dynamically discover the structure for each problem instance. In contrast, we propose an adaptive technique that learns from previous experience in a domain.

Two successful approaches that use hand-crafted information about the domain structure are hierarchical task networks and planning with temporal logic control rules. Planning with hierarchical task networks (Nau *et al.* 2003) can be seen as an extension of classical planning where the search is guided and restricted according to a hierarchical representation of a domain. Human experts design hierarchies of tasks that show how the initial task of a problem can be decomposed down to the level of regular actions. In planning with temporal logic control rules, a formula is associated with each state in the problem space. The formula of the initial state is provided with the domain description. The formula of any other state is obtained based on its successor's formula. When the formula associated to a state can be proven to be false, the subtree of that state is pruned from the search space. The best known planners of this kind are TLPlan (Bacchus & Kabanza 2000) and TALPlanner (Kvarnström & Doherty 2001). While efficient, these approaches also rely heavily on human knowledge, which sometimes can be expensive or impossible to obtain.

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. In effect, such a macro traverses a "valley" in the search space, and using this later can correct the heuristic evaluator. A macro filtering procedure uses both simple static rules and dynamic statistical data.

(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.

As in (Botea, Müller, & Schaeffer 2004b), we describe a technique to automatically generate and filter a set of macros to be used for future problems. In this paper, we extend our previous work in several directions. First, we increase the applicability from STRIPS domains to ADL domains. For more details, including a discussion of the trade-offs, see (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 the generality of the method. Third, we increase the size of macros from 2 moves to arbitrary values. Fourth, we generalize the definition of macros allowing partially ordered sequences.

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). Our helpful macro pruning method has similarities with both. 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 doesn't give up completeness of the search algorithm.

Look-ahead policies are similar to using macro-operators. The main idea of a lookahead policy is to execute 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. This technique heuristically orders the actions in the relaxed plan and iteratively applies them 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 that the loop can be resumed.

We also execute several actions from the relaxed plan in the real world, but impose a well defined structure on the action sequence. A lookahead policy has no explicit limitation on how many actions to apply, whereas we are limited to the length of a macro. A possible extension would be to apply at a step several macros in a row, and this is an interesting topic for future work.

Macro-moves were successfully used in single-agent search problems such as puzzles or path-finding in commercial computer games, usually in a domain-specific implementation. The sliding-tile puzzle was among the first testbeds for this idea (Korf 1985; Iba 1989). 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 an approach that decomposes a maze into a set of rooms connected by tunnels (Botea, Müller, & Schaeffer 2002). Search is performed at the higher level of abstract move sequences that rearrange the stones inside a room so that a stone can be transferred from one room to another. Hernádvölgyi uses macro-moves for solving Rubik's Cube puzzles (Hernádvölgyi 2001). 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. This approach greatly speeds up the search while producing near-optimal solutions.

Conclusion

Despite the great progress that AI planning has recently achieved, many benchmarks 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 search in future problems. We evaluated our method on standard benchmarks from the fourth international planning competition, showing significant improvement for domains where structure information can be inferred.

Exploring our method deeper 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 has been conducted in this direction, even though the problem is very important.

Another interesting topic is to use macros in the graphplan algorithm, rather than the 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 also plan to explore how the heuristic evaluation based on the relaxed graphplan can be improved based on macro-operators. Our previous work where a STRIPS macro is added to the original domain formulation as a regular operator suggests that macros could help to improve the evaluation of a state. In this framework, macros are considered in the relaxed graphplan computation just like any other operator, and they lead to more accurate evaluations.

References

Bacchus, F., and Kabanza, F. 2000. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence* 16:123–191.

Bacchus, F., and Yang, Q. 1994. Downward Refinement and the Efficiency of Hierarchical Problem Solving. *Artificial Intelligence* 71(1):43–100.

Bacchus, F. 2001. AIPS'00 Planning Competition. *AI Magazine* 22(3):47–56.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2004. Macro-FF. In *Booklet of the Fourth International Planning Competition*, 15–17.

Botea, A.; Müller, M.; and Schaeffer, J. 2002. Using Abstraction for Planning in Sokoban. In *Proceedings of the 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 *Proceedings of the International Conference on Automatic Planning and Scheduling ICAPS-04*, 181–190.

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.

Helmert, M. 2004. A Planning Heuristic Based on Causal Graph Analysis. In *Proceedings of the International Conference on Automatic Planning and Scheduling ICAPS-04*, 161–170.

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

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 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 *Booklet of the Fourth 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.

Knoblock, C. A. 1994. Automatically Generating Abstractions for Planning. *Artificial Intelligence* 68(2):243–302.

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

Kvarnström, J., and Doherty, P. 2001. TALplanner: Temporal Logic Based Forward Chaining Planner. *Annals of Mathematics and Artificial Intelligence (AMAI)* 30:119–169.

Long, D., and Fox, M. 2003. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research* 20:1–59. Special Issue on the 3rd International Planning Competition.

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

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 21(2):35–55.

Minton, S. 1985. Selectively Generalizing Plans for Problem-Solving. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, 596–599.

Nau, D.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, J.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research* 20:379–404.

Nguyen, X., and Kambhampati, S. 2001. Reviving Partial Order Planning. In Nebel, B., ed., *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, 459–466.

Vidal, V. 2004a. A Lookahead Strategy for Heuristic Search Planning. In *Proceedings of the International Conference on Automatic Planning and Scheduling ICAPS-04*, 150–159.

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