

Macro-FF

Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer

Department of Computing Science, University of Alberta

Edmonton, Alberta, Canada T6G 2E8

{adib,emarkus,mmueller,jonathan}@cs.ualberta.ca

Abstract

This document describes Macro-FF, an adaptive planning system developed on top of FF version 2.3. The original FF is a fully automatic planner that uses a heuristic search approach. In addition, Macro-FF can automatically learn and use macro-actions with the goal of reducing the number of expanded nodes in the search. Macro-FF also includes implementation enhancements for reducing space and CPU time requirements that could become performance bottlenecks in some problems.

Introduction

Macro-FF is an extension of the automatic planner FF version 2.3 (Hoffmann & Nebel 2001). We developed a first version of Macro-FF as a tool for exploring how macro-actions can reduce the complexity of automated planning (Botea, Müller, & Schaeffer 2004). Further extensions have been implemented to prepare Macro-FF for participating in the fourth international planning competition (IPC4). Macro-FF is designed for classical planning and can use both STRIPS and ADL domain formulations. The plans that Macro-FF produces are not guaranteed to be optimal. The system has no capabilities for temporal and metric planning, and implements no support for derived predicates and timed initial literals.

This extended abstract summarizes the architecture of Macro-FF. The structure of our presentation is the following: First, we provide a brief description of FF, focusing on the parts that are relevant for our work. Next, we describe the main contributions that we have added to the original FF. The extensions that we present mainly go into two directions:

- Speeding up search with macro-operators. A macro-operator is an ordered sequence of operators together with a variable mapping showing how the variable sets of operators overlap. The intuition for using macro-actions is that several actions can often work in a sequence to accomplish a local task (e.g., first take the key out of the pocket, next unlock the door). Identifying and exploiting such sequences have a significant potential to reduce the overall planning effort. Macro-FF can automatically

learn and use macro-actions with the goal of reducing the number of expanded nodes in the search.

- Implementation enhancements for reducing memory and CPU time requirements. The number of expanded nodes and the solution quality are not affected by changes in this category. However, when the memory or CPU time necessary to solve a problem are larger than the available resources, this kind of improvements can make the difference between failure and success in solving a problem.

Overview of FF

FF is a state-of-the-art fully automatic planner that uses a heuristic search approach. The basic version of FF, which we started from, is designed for classical planning. Specialized versions of FF have capabilities for planning with numerical state variables (Metric-FF) and planning with incomplete information (Conformant-FF).

FF uses a preprocessing phase that includes the generation of all facts (i.e., instantiated predicates) and actions (i.e., instantiated operators) that could possibly be used in the current problem instance. These elements, which are extensively used during the search, become available at little runtime cost.

FF automatically computes a heuristic state evaluator that guides the search process. Given a state, the distance to a goal state is approximated by the length of a relaxed plan that achieves the goal conditions starting from the current state. This plan is computed in a relaxed GRAPHPLAN framework, where the delete effects of actions are ignored.

The planner implements two search algorithms. Enforced hill climbing (EHC) is a fast but incomplete algorithm that greedily searches for a goal state in the problem space. EHC starts from the initial state and performs a local search using a breadth-first strategy. When a state with a better evaluation than the starting state is found, the current local search stops and a new local search is launched starting from the newly found state.

In EHC, the GRAPHPLAN computation for a state is used not only to find a heuristic evaluation, but also to further prune the search space through a mechanism called helpful action pruning. When a state is expanded, only moves that occur in the relaxed plan and belong to level 0 of the GRAPHPLAN (i.e., can be applied to the current

state) are considered. With no helpful action pruning, EHC is complete in undirected search spaces.

EHC stops when either a goal state is found, or the open list associated with the current local search is empty. When the second alternative occurs (i.e., EHC fails because of its incompleteness), a complete best-first search (BFS) algorithm is launched to find a path to a goal state.

Learning and Using Macro-Operators

When treated as single moves, macro-actions have the potential of influencing the planning process in two important ways. First, macros can change the search space, adding to a node successor list states that would normally be achieved in several steps. Intermediate states in the macro sequence do not have to be evaluated, reducing the search costs considerably. In effect, the maximal depth of a search could be reduced for the price of slightly increasing the branching factor. Second, macros can improve the heuristic evaluation of states. As shown before, FF computes this heuristic by solving a relaxed planning problem (i.e., the delete effects of actions are ignored) in a GRAPHPLAN framework. Consider two normal actions that occur in a sequence in a relaxed plan. It is not guaranteed that this chaining translates to a valid action sequence in the real world (e.g., when the first action has a delete effect that is a precondition for the second action). Consider now the case when two actions compose a macro, so that the relaxed plan contains that macro rather than two separate actions. A relaxed macro can always be translated to its correspondent in the real world, as any other action does.

Learning Phase

Macro-FF learns a set of macros through a training phase that uses several sample problems of a domain. Each training problem is first solved with no macros in use. The found plan P is represented as a directed *solution graph*, where each node represents a plan action, and edges show the relative order and distance between two actions in the solution. If action a_1 occurs before action a_2 in P , then a weighted edge $e = (a_1, a_2)$ is added to the graph. The weight is the distance between a_1 and a_2 in the solution.

We define a macro-action as a linear sequence in the solution graph, with the corresponding parameter mapping. To reduce the training effort, our implementation considers only sequences of two consecutive actions as possible macros (i.e., only pairs of nodes linked by edges with weight 1).

The macro-actions are mapped to macro-operators by replacing the instantiated parameters with generic variables. Macro-operators have weights (initially set to 1.0) and are stored in a global list ordered by their weights.

For each macro-operator m , the current training problem is re-solved using m . To measure the usefulness of m , we compare the effort to solve the problem with macro m in use to the initial solving effort. We evaluate the effort to solve a problem as the total number of expanded nodes. The weight update formula for m uses the difference between N (the effort for solving the problem with no macros in use) and

N_m (the effort when macro m is used). A sigmoid function maps the difference into the range $(-1, 1)$. The update value further contains the initial solution length as a multiplicative factor, which measures how hard the current problem is. The harder the problem, the larger this weight update should be. After the training phase completes, the best macros can be used in the solving phase.

Solving Phase

Current Implementation. For IPC4, we store the macros using a *compact* representation. This includes the ids of the operators that compose the macro and the variable mapping, but ignores the precondition and effect formulas. In the solving mode, the compact patterns of the best macros are used for online checking if two instantiated actions compose a macro. The current implementation uses macros to change the search space (as shown next), but does not affect the computation of the heuristic state evaluation. Improving the heuristic state evaluation with macros is an important topic for future work.

To explore the search space more efficiently, we exploit the relaxed plan that the system computes for the current state to be expanded. Our idea is to try to execute parts of the relaxed plan in the real world, hoping to move toward a goal state faster. We examine the relaxed plan to find action sequences that match a macro pattern. Each time when such a sequence is identified, we check if this could be executed in the real world, starting from the current state. This verification is fast, as we do not compute the evaluation of the states along the execution path. If executing a macro-action succeeds, we consider the resulting state as a successor of the current state and add it to the open queue.

In enforced hill climbing, we order these macro successors before the regular successors of a state. In effect, macro successors are expanded earlier than regular successors. In addition, our code includes an ordering scheme for normal successors, which we had developed before using macro successors. In the current implementation, this still might be useful in cases when a macro is not part of the relaxed plan, but could occur in the real world. We order the normal successors giving priority to moves that continue as a macro sequence the last action on the current branch (i.e., the action that led to the currently expanded state S). We split the normal successors of state S into two subsets $\text{Succ}_1(S)$ and $\text{Succ}_2(S)$. Assume a_S is the action that we applied to obtain S , and $a_{S'}$ is the action that we apply from S to obtain a successor S' . If pair $(a_S, a_{S'})$ matches the pattern of a learned macro operator, then $S' \in \text{Succ}_1(S)$. Otherwise, $S' \in \text{Succ}_2(S)$. Elements from $\text{Succ}_1(S)$ are ordered before elements from $\text{Succ}_2(S)$. Inside such a set, an additional move ordering scheme, preserved from the original FF, is applied.

In best-first search, macros act as a method for search depth control. In the original implementation, when a node is expanded, all its normal successors are added to the open list, except for states that have been visited before (a transposition table is used to identify duplicates). In addition to this, our new implementation explores branches that compose a macro more deeply. States are further expanded on

the branches that match a macro pattern, and the resulting states are added to the open list earlier than in the original FF.

Alternative Approach. Another possible way of using macro-operators is to add them as normal single-step operators to the initial domain formulation, as described in (Botea, Müller, & Schaeffer 2004). In this way, macro-actions are naturally used in both exploring the search space (i.e., as possible moves when nodes are expanded) and computing the heuristic state evaluation in the relaxed GRAPHPLAN framework, with no need to change the original code of FF. In effect, the number of expanded nodes can be reduced for the price of increased preprocessing time and cost per node at run-time.

This approach was hard to use in IPC4, as the macro-operators added to the domain formulation have to have complete PDDL definitions, including precondition and effect formulas. Expressing these formulas starting from the contained operators is easy in STRIPS, but hard in more complex PDDL subsets such as ADL, where the preconditions and the effects of the contained operators can interact in a very complex way. However, for IPC4, we used the ADL formulation for several domains that were available both in ADL and STRIPS. The reason is that the STRIPS formulation of these domains have a separate operator file for each problem. This makes our learning algorithm hard to apply, as several training problems are necessary for a given domain definition.

Implementation Enhancements

The enhancements described in this section have the goal of reducing the space and CPU requirements of the planner, and do not affect the number of expanded nodes and the quality of found plans. We describe two enhancements, one for speeding-up the best-first search and one for reducing the space needs for the preprocessing.

The best-first search (BFS) algorithm uses an open list of nodes that have been generated but not expanded yet. The elements in this list are stored in increasing order according to their heuristic evaluation, so that the next node chosen for expansion is the most promising in the list. FF version 2.3 implements the open queue as a linear linked list. A node insertion requires a linear traversal of the list, so that the ordering of the list is preserved. Experiments with some of the competition problems have shown that this linear traversal can be a serious bottleneck for best-first search. We changed the original linked list of nodes to a linked list of buckets, where each bucket is a linked list of nodes having the same heuristic value. The insertion of a node requires finding the appropriate bucket for that node, which takes time linear in the number of different heuristic values in the open queue plus a constant time for inserting the node at the end of the bucket (this preserves the existing tie-breaking rule).

FF version 2.3 is optimized for speed by using preprocessing to a large extent. Some of the data structures used for holding the preprocessing information grow exponentially with the problem complexity, so that this method does not scale to more complex problems. We took an initial step to

address this problem by replacing a large lookup table by a different data structure. The lookup table was used for holding instantiated facts that occur in the initial state. The new implementation uses a balanced binary tree for logarithmic lookup time.

Acknowledgment

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE). We thank Jörg Hoffmann for making the source code of FF available.

References

- Botea, A.; Müller, M.; and Schaeffer, J. 2004. Using Component Abstraction for Automatic Generation of Macro-Actions. In *Proceedings of the International Conference on Automated Planning and Scheduling ICAPS-04*.
- Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14:253–302.