Fast Planning with Iterative Macros

Adi Botea
National ICT Australia
Australian National University
Canberra, ACT
adi.botea@nicta.com.au

Martin Müller
Dept. of Computing Science
University of Alberta
Edmonton, Canada
mmueller@cs.ualberta.ca

Jonathan Schaeffer
Dept. of Computing Science
University of Alberta
Edmonton, Canada
jonathan@cs.ualberta.ca

Abstract

Research on macro-operators has a long history in planning and other search applications. There has been a revival of interest in this topic, leading to systems that successfully combine macro-operators with current state-of-the-art planning approaches based on heuristic search. However, research is still necessary to make macros become a standard, widely-used enhancement of search algorithms. This article introduces sequences of macro-actions, called iterative macros. Iterative macros exhibit both the potential advantages (e.g., travel fast towards goal) and the potential limitations (e.g., utility problem) of classical macros, only on a much larger scale. A family of techniques are introduced to balance this trade-off in favor of faster planning. Experiments on a collection of planning benchmarks show that, when compared to low-level search and even to search with classical macro-operators, iterative macros can achieve an impressive speed-up in search.

1 Introduction

Research on macro-operators has a long history in planning and other search applications. Recent years have shown a revival of this topic, leading to systems that successfully combine macro-operators with current state-of-the-art planning approaches based on heuristic search. However, macros have significant capabilities yet to be exploited. There is a need to continue the previous efforts on this topic, aiming to reach a point where macros would be considered to be a standard performance enhancement (e.g., such as hash tables for fast detection of duplicate nodes).

In this article, we introduce sequences of macro-actions called iterative macros. Figure 1 illustrates the differences between low-level search, search with classical macros, and search with iterative macros. First, consider low-level search versus search with classical macros. Macros add the ability to travel towards a goal with big steps, with few intermediate nodes expanded or evaluated heuristically. However, macros increase the branching factor, and often also the processing cost per node. Inappropriate macros guide the search in a wrong direction, which increases the total search time while solution quality decreases. Addressing this *performance trade-off* is the key to making macros work.

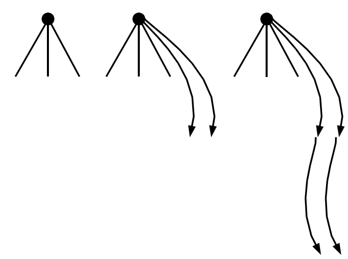

Figure 1: State expansion with atomic actions (left), atomic actions + macros (center), and atomic actions + iterative macros (right). Each short line is an atomic action. Each curved arrow is a macro-action.

Iterative macros are macros of macro-actions. They have similar potential benefits and limitations as classical macros, only on a much larger scale. Iterative macros progress much faster down a branch of the search, with exponentially larger possible savings. On the downside, there can be exponentially more instantiations of iterative macros, with many of them leading to dead ends. An iterative macro is more expensive to compute, being the sum of instantiating each contained macro. Tuning the performance trade-off is more challenging than for classical macros.

The model discussed in this paper extends the approach in Botea *et al.* 2005, which offers a framework for generating, filtering, and using macros at runtime. The contributions of this paper are:

1. *Iterative macros*, a runtime combination of macros to enhance program performance,
2. New techniques to address the performance trade-offs for iterative macros: algorithms for offline filtering, dynamic composition (i.e., instantiating an iterative macro at runtime), and dynamic filtering (i.e., pruning instantiations of an iterative macro at runtime); and
3. Experiments using standard planning benchmarks that show orders of magnitude speed up in several standard domains, when compared to low-level search and even to a search enhanced with classical macros.

Section 2 briefly reviews related work on macros. Section 3 introduces the necessary definitions. Section 4 introduces iterative macros and the algorithms for offline filtering, dynamic composition, and dynamic filtering. Experimental results are given in Section 5. Section 6 contains conclusions and ideas for future work.

2 Related Work

Related work on macros in planning dates back to the STRIPS planner [Fikes and Nilsson, 1971]. Subsequent contributions include off-line filtering of a set of macros [Minton, 1985], partial ordering of a macro's steps [Mooney, 1988], and generating macros able to escape local minima in a heuristic search space [Iba, 1989]. In a problem representation with multi-valued variables, McCluskey and Porteous [1997] use macros to change the assignment of a variable to a given value in one step.

Several recent contributions successfully integrate macros with state-of-the-art heuristic search planners such as FF [Hoffmann and Nebel, 2001]. Vidal [2004] composes macros at runtime by applying steps of the relaxed plan in the original problem. Botea *et al.* [2005] prune instantiations of macros based on their similarity with a relaxed plan. Coles and Smith [2005] generate macros as plateau-escaping sequences. Newton *et al.* [2005] use genetic algorithms to generate macros. The contributions of Vidal [2004] and Botea *et al.* [2005] are the most closely related, since all three approaches exploit the similarity between a macro and a relaxed plan.

Application-specific macros have been applied to domains such as the sliding tile puzzle [Korf, 1985], Rubik's Cube [Korf, 1983; Hernádvölgyi, 2001], and Sokoban [Junghanns and Schaeffer, 2001]. While interesting, a detailed discussion and comparison of all these approaches is beyond the scope of this paper.

3 Framework and Basic Definitions

The basic framework of this work is planning as forward heuristic search. To guide the search, a relaxed plan that ignores all delete effects of actions is computed for each evaluated state [Hoffmann and Nebel, 2001]. Search is enhanced with iterative macros as illustrated in Figure 1 (right) and detailed in Section 4. The strategy for using iterative macros consists of three steps: (1) extract macro-operators from solutions of training problems, (2) statically filter the set of macro-operators, and (3) use the selected macro-operators to compose iterative macros at runtime. Steps 1 and 2 deal only with classical macro-operators. Only step 3 involves the new iterative macros. The model of Botea *et al.* 2005 serves as a starting point for implementing the first two steps. It provides a framework for generating, filtering, and using classical macro-operators at runtime in planning. However, experiments with iterative macros showed that more powerful filtering capabilities were needed. The new enhanced method for filtering in step 2 is described in Section 4.1.

The rest of this section contains definitions of concepts used in the following sections. For simplicity, totally ordered macros are assumed (all definitions can be generalized to partial-order macros). The macro extraction phase builds macros with partial ordering of the steps. However, to save computation time, only one possible ordering is selected at runtime.

Let $\mathcal{O}$ be the set of all domain operators and $\mathcal{A}$ the set of all ground actions of a planning problem. A *macro-operator* (macro-schema) is a sequence of domain operators $ms[i] \in \mathcal{O}$ together with a parameter binding σ: $ms = ((ms[1], ms[2], \ldots, ms[l]), \sigma)$.

Partially instantiating a macro can be defined in two equivalent ways as either (1) replacing some variables with constant objects or (2) replacing some operators with ground actions. The second definition is more appropriate for this work, since macros reuse actions from a relaxed plan and hence action-wise instantiation is needed. A partial instantiation of a macro is $mi = ((mi[1], mi[2], \ldots, mi[l]), \sigma)$, where $(\forall i \in \{1, \ldots, l\}) : (mi[i] \in \mathcal{A} \vee mi[i] \in \mathcal{O})$.

A total macro-instantiation (shorter, macro-instantiation) has all steps instantiated ($\forall i : mi[i] \in \mathcal{A}$). Macro-operators and macro-instantiations are the extreme cases of partial macro-instantiations. When the distinction is clear from the context, the term "macro" can refer to any of these.

When instantiating one more step in a partial instantiation mi, it is important to ensure that the new action is consistent with all the constraints already existing in mi. More precisely, given a partial instantiation mi, a position i and a state s from which mi is being built, define the *consistency set* $\mathrm{Cons}(mi, i, s)$ as containing all actions $a \in \mathcal{A}$ such that: (1) a corresponds to the operator on the i-th position of mi, (2) a does not break the parameter bindings of mi, and (3) if either $i = 1$ or the first $i-1$ steps are instantiated, then adding a on the i-th position makes this i-step sequence applicable to s. Only actions from $\mathrm{Cons}(mi, i, s)$ can be used to instantiate the i-th step of mi. Obviously, instantiating a new step can introduce additional binding constraints. Instantiating steps in a macro can be done in any order. When step i is instantiated, its bindings have to be consistent with all previously instantiated steps, including positions larger than i.

Finally, let $\gamma(s, a_1 \ldots a_k)$ be the state obtained by applying the action sequence $a_1 \ldots a_k$ to state s. For the empty sequence ϵ, $\gamma(s, \epsilon) = s$. If $\exists i \leq k$ such that a_i cannot be applied to $\gamma(s, a_1 \ldots a_{i-1})$, then $\gamma(s, a_1 \ldots a_k)$ is undefined.

4 Iterative Macros

This section describes a technique for speeding up planning using iterative macros. Section 4.1 presents a method for statically filtering a set of macro-operators to identify candidates that can be composed to form iterative macros. Section 4.2 focuses on integrating iterative macros into a search algorithm. Methods that effectively address the challenging tasks of instantiation and pruning are described.

4.1 Static Filtering

The model introduced by Botea *et al.* [2005] was implemented and enhanced. Botea *et al.* analyze solutions to a set of test problem instances to extract a potentially useful set of macro-operators. The macros are then ranked by favoring those that 1) appear frequently in solutions, and 2) significantly reduce the search effort required for each application.

Two important limitations of this ranking model are that it ignores the interactions of macros when used together, and that it provides no automatic way to decide the number of selected macros.

Our enhancement first selects the top K macros (where K is a parameter) returned by the original procedure and then tries to filter this down to a subset that solves the training set most efficiently in terms of expanded nodes. Since enumerating all subsets of a set with K elements is exponentially hard, we use an approximation method whose complexity is only linear in K. For each i from 1 to K, the training set is solved with macro m_i in use. Macros are reordered according to the search effort. More precisely, m_i is better than m_j if $N_i < N_j$, where N_l is the total effort (expanded nodes) to solve the training set using macro m_l. Ties are broken according to the original ranking.

Based on the new ordering, the training set is solved using the top i macros, $1 \leq i \leq K$. Assume N is the total number of nodes expanded to solve the training set with no macros in use, N_i^T the total effort to solve the training set with the top i macros, and

$$b = \arg \min_{1 \leq i \leq K} N_i^T.$$

If $N_b^T < N$, then the learning procedure returns the top b macros. Otherwise, no macros are learned for that domain.

In the experiments described in Section 5, small training instances are used, to keep the learning time low. K is set to 5, since the number of useful macros in those domains is typically less than 5. For larger domains, where more macros could be beneficial, a larger value of K might produce better results at the price of longer training time.

4.2 Iterative Macros in Search

Integrating iterative macros into a search algorithm raises two major challenges: instantiation and pruning. In the most general case, the total number of iterative macros applicable to a state is in the order of B^D, where B is the number of classical macro instantiations applicable to a state, and D is the number of macros contained in an iterative macro. Each instantiation can be expensive to compute, since its cost is the total cost of instantiating all the contained macros.

If instantiation and pruning were performed separately, a large effort could be spent on building elements that would be rejected later. Therefore a combined algorithm tries, for a given state, to build only one iterative macro which shows promise to take the search closer to a goal state. The guidance in building this iterative macro is given by the relaxed plan of the state being expanded. Building a macro instantiation is founded on two simple, yet powerful ideas. First, when deciding how to instantiate a given step, heuristics are used to select an action that will allow a large number of relaxed steps to be subsequently inserted. Second, for the steps not filled with relaxed plan actions, other actions are used that preserve the correctness and the variable bindings of the iterative macro. This completion is an important feature of the algorithm, since a relaxed plan often misses steps that have to be part of the unrelaxed solution.

Figure 2 shows the procedure for building an iterative macro in pseudo-code. It takes as input a global list of macro-

```
ComposeIterativeMacro(MS, s, RP)
  U ← ∅; itm ← empty sequence;
  while (true)
    for (each ms ∈ MS)
      mi ← Instantiate(ms, γ(s, itm), RP \ U);
      if (instantiating mi succeeded)
        U ← U ∪ [mi ∩ RP]; // mark used steps
        itm ← itm + mi; // concatenate
        break; // restart outer loop
    if (no iteration of last for loop instantiated a macro)
      return itm;
```

Figure 2: Composing an iterative macro at runtime.

```
Instantiate(ms, s, RS)
  for (each a ∈ Cons(ms, 1, s))
    mi ← Matching(a, ms, s, RS);
    if (|mi ∩ RS| ≥ threshold)
      fill remaining gaps in mi;
      if (all steps of mi are instantiated)
        return mi;
  return failure;
```

Figure 3: Instantiating one macro-action.

schemas (*MS*), a current search state (s), and the relaxed plan computed for that state (*RP*). Each iteration of the main loop tries to append one more macro to the iterative macro. The inner loop iterates through the global list of macro-schemas. As soon as instantiating such a macro-schema succeeds, the algorithm greedily commits to adding it to the iterative macro and a new iteration of the outer loop starts. This procedure automatically determines the length of an iterative macro (the number of contained macros).

In the code, U is the set of all relaxed plan steps already inserted in the iterative macro. During subsequent iterations, the used relaxed steps will be ignored when the *matching* of a macro instantiation with a relaxed plan is computed. Intuitively, the Matching procedure tries to maximize the number of relaxed steps used in a macro-instantiation. More formal details on matching are provided later.

Figure 3 presents the Instantiate procedure that instantiates one macro-action as part of an iterative macro. The input parameters are a macro-schema (*ms*), a search state (s), and a set of relaxed steps (i.e., the original relaxed plan minus the already used steps). The main loop iterates through all actions that could be used as the first step of the macro *ms* (i.e., are applicable to s and are instantiations of the first macro's operator).

For each action $a \in \text{Cons}(ms, 1, s)$, the method Matching$(a, ms, s, RS)$ creates a partial instantiation of *ms* with first step a, followed by zero or more steps instantiated with elements from *RS*, and zero or more uninstantiated steps (see Figure 4 and a discussion later). If the number of relaxed steps is below a given threshold, the corresponding partial instantiation is abandoned. Otherwise, an attempt is made to fill the remaining gaps (uninstantiated steps) with *any* consistent actions. As soon as a complete instantiation is built, the method returns without considering any other possible out-

Matching(a, ms, s, RS)
  $mi \leftarrow ms$; // create local partial instantiation
  $mi[1] \leftarrow a$;
  for ($i = 2$ to length(ms))
    if (Cons(mi, i, s) $\cap RS = \emptyset$)
      continue; // leave $mi[i]$ uninstantiated
    for (each $rp \in$ Cons(mi, i, s) $\cap RS$)
      undo the instantiation of $mi[i]$, if any;
      $mi[i] \leftarrow rp$;
      count how many subsequent positions j
        can be filled with elements from Cons(mi, j, s) $\cap RS$;
    select the element rp with the highest count value;
    undo the instantiation of $mi[i]$;
    $mi[i] \leftarrow rp$;
  return mi;

Figure 4: Matching a macro instantiation with a relaxed plan.

comes. For simplicity, the pseudo-code skips the details of how the threshold is computed. An effective heuristic is to set the threshold to the largest matching encountered when the *ms* macro-schema is used as a parameter, regardless of the values of the other parameters a, s and *RS*.

The matching attempts to use as many elements from *RS* as possible in a macro instantiation. An exact computation of the maximal value can be expensive, since it might require enumerating many possible instantiations of *ms* applicable to a state. Instead, the greedy procedure presented in Figure 4 tries, at step $i, 2 \leq i \leq$ length(ms), to commit to using a relaxed step *rp* for instantiating $mi[i]$. If no such step exists (i.e., $RS \cap$Cons(mi, i, s) $= \emptyset$), then $mi[i]$ is left uninstantiated. Otherwise, an element from $RS \cap$Cons(mi, i, s) is selected using a heuristic test (see the pseudo-code for details). In practice, the number of consistent actions quickly decreases as new steps are instantiated, since each new step can introduce additional binding constraints.

5 Experimental Results

Classic and iterative macros were implemented on top of FF [Hoffmann and Nebel, 2001]. FF 3.4 handles both STRIPS and ADL domains, but not numeric variables, axioms, or temporal planning.

This research was tested on a large set of benchmarks from previous international planning competitions. Both STRIPS (Satellite, Blocksworld, Rovers, Depots, Zeno Travel, DriverLog, Freecell, Pipesworld No Tankage Nontemporal, Pipesworld Tankage Nontemporal) and ADL (Promela Dining Philosophers, Promela Optical Telegraph, Airport, Power Supply Restoration Middle Compiled—PSR) representations were used.

Experiments were run on a 3.2GHz machine, with a CPU limit of 5 minutes and a memory limit of 1GB for each problem instance. Planning with iterative macros, planning with classical macros, and planning with no macros were compared. To plan with classical macros, the length of an iterative macro was limited to one macro instantiation. Results are shown for 11 of the 13 domains. In the two remaining domains, PSR and Pipes Tankage, no macros were learned, since their performance on the training set was worse than low-level search (see Section 4.1 for details).

Figure 5 shows the number of expanded nodes in each domain on a logarithmic scale for each of no macros, classical macros and iterative macros. Note that some lines are missing a data point—this represents a problem instance that was not solved by that planner.

When analyzing the expanded nodes performance, the tested application domains can roughly be split into two categories. In the first category of eight benchmarks (all 11, less DriverLog, Freecell and Pipesworld), planning with macros is much better than low-level search. Iterative macros are better than classical macros, with the notable exception of Philosophers, where both kinds of macros perform similarly. In this application domain, classical macros are enough to achieve impressive savings, and there is little room for further improvement. In Zeno Travel, the savings in the search tree size come at the price of a relatively large increase in solution length. See Figure 6 and a discussion later. When comparing iterative macros vs classical macros, in domains Satellite, Blocksworld, Rovers, Depots, and Airport a reduction in the number of expanded nodes by at least an order of magnitude is seen for the hard problem instances.

In the second category, the benefits of macros are more limited. In DriverLog, macros are usually faster, but there are a few exceptions such as data point 7 on the horizontal axis, where classical macros fail and iterative macros are much slower than low-level search. In Freecell, classical macros and iterative macros have similar performance in all instances. For many Freecell problems, planning with macros is similar to planning with no macros. When differences are encountered, the savings are more frequent and much larger as compared to cases where macros are slower than low-level search. Finally, in Pipesworld No Tankage the performance of macros compared to low-level search varies significantly in both directions. Iterative macros are faster than classical macros, but the latter solve one more problem. No clear conclusion is drawn for this domain. Further analysis of these three domains is left as future work.

Macros often lead to solving more problems than low-level search. Given a domain, assume P_{im}, P_{cm} and P are the numbers of problems solved with iterative macros, classical macros, and no macros respectively. For our data sets and time constraints, the value of $(P_{im} - P, P_{cm} - P)$ is $(1, 1)$ in Satellite, $(3, 2)$ in Blocksworld, $(37, 34)$ in Optical, $(36, 36)$ in Philosophers, $(5, 5)$ in Zeno Travel, $(3, 2)$ in DriverLog, and $(1, 1)$ in Freecell, and $(0, 1)$ in Pipesworld.

Figure 6 illustrates how macros affect the quality of solutions and the cost per node in search. Each chart has 11 two-point clusters, one for each domain. First, consider the top chart. Given a problem instance, assume L_{im}, L_{cm}, and L are the lengths of solutions when iterative macros, classical macros, and no macros are used respectively, $R_{im} = L_{im}/L$ and $R_{cm} = L_{cm}/L$. The leftmost data point of a cluster shows the average, minimum, and maximum value of R_{im} over the problem set of the corresponding domain. The rightmost data point shows similar statistics for R_{cm}. Macros slightly improve the average solution length in Freecell and leave it unchanged in Optical and Philosophers. In all domains but Zeno Travel, the average overhead is at most 20%

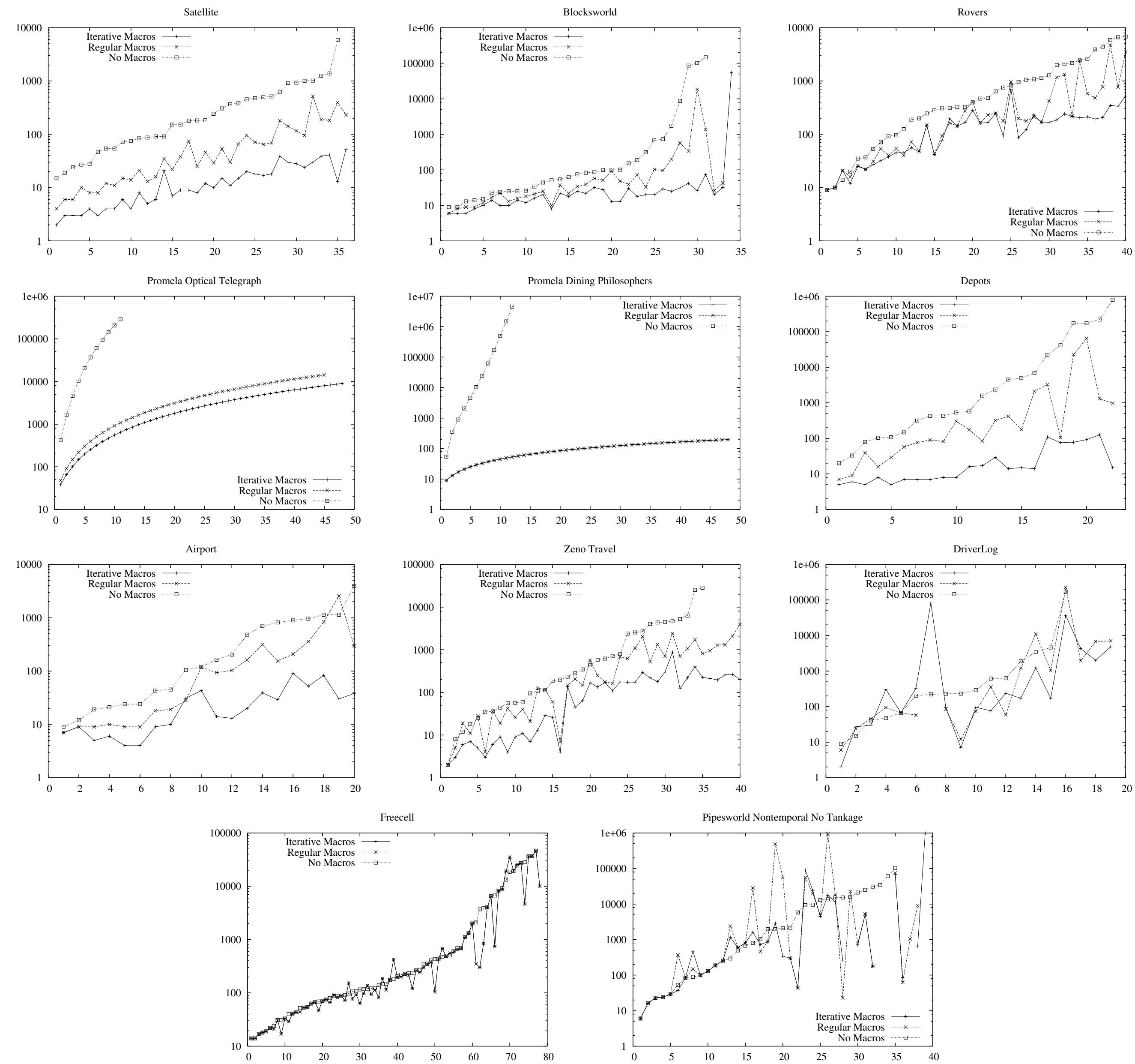

Figure 5: Search effort as expanded nodes. Problem sets are ordered so that the "No Macros" curve is monotonically increasing.

for iterative macros and at most 12% for classical macros.

The bottom chart in Figure 6 presents similar statistics for the cost per node $C = \frac{\text{search time}}{\text{expanded nodes}}$ instead of solution length L. To include a problem instance into the statistics, it has to be solved by both the corresponding type of macros and the low-level search within a time larger than 0.05 seconds. We included the time threshold for better accuracy of the statistics. There always is a small noise in the reported CPU time and, if the total time is in the same order as the noise, the cost per node measurement becomes unreliable. No statistics could be collected for Philosophers (both kinds of macros) and for Blocksworld (iterative macros), where macros solve problems very fast.

Processing a node in low-level search includes computing a relaxed plan and checking whether that node has been visited before. Macros add the overhead of their instantiation. Even if much smaller than the expanded nodes savings shown in Figure 5, the overhead can be surprisingly high. Profiling tests have shown that the main bottleneck in the current implementation of macros is attempting to fill gaps in a partial instantiation (Figure 3, line 5). Fortunately, this step can be implemented much more efficiently. When looking for a consistent action to fill a gap, the corresponding operator schema

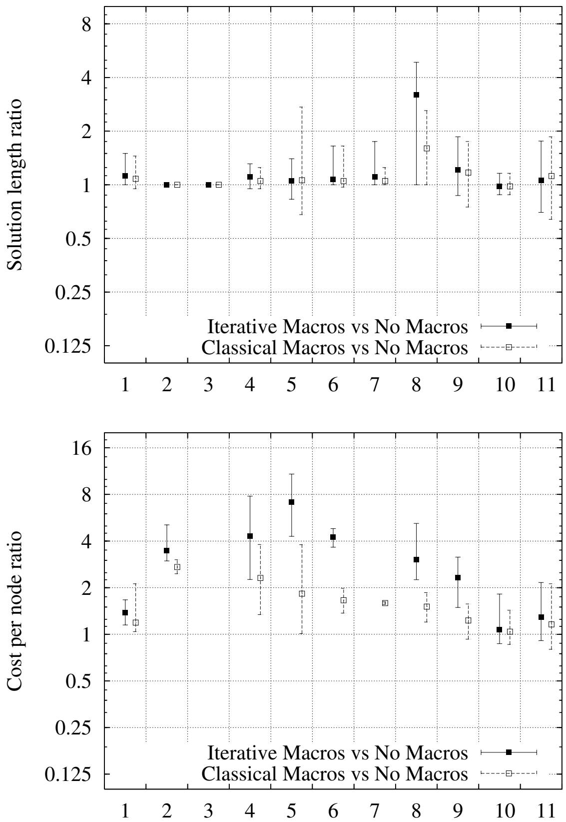

Figure 6: Effects of macros on solution quality (top) and cost per node in search (bottom). The two-point clusters correspond in order to (1) Satellite, (2) Optical, (3) Philosophers, (4) Rovers, (5) Depots, (6) Airport, (7) Blocksworld, (8) Zeno Travel, (9) DriverLog, (10) Freecell, and (11) Pipesworld.

is known from the structure of the macro. Often, the values of all variables are already set by the previously instantiated steps. This would be enough to determine the corresponding instantiated action. However, to the best of our knowledge, no mapping from an operator together with a list of instantiated arguments to the resulting ground action is available in FF at search time. Instead, our current implementation generates states along a macro instantiation and calls FF's move generator when a gap has to be filled. If an applicable action exists that is consistent with the current partial instantiation, it is used to instantiate the given step in the macro.

6 Conclusion

This paper describes how macros of macro-actions, called iterative macros, can be used to speed up domain independent planning. Techniques for static filtering, dynamic composition and pruning of iterative macros have been introduced to turn the trade-off between the benefits and the limitations of iterative macros in favor of the former. Experiments in several standard benchmarks demonstrate impressive savings that iterative macros can achieve as compared to low-level search and even to a search enhanced with classical macros. Worst-case behavior and solution quality remain acceptable.

Future work includes faster processing per node when searching with macros. Another avenue of research is to investigate how iterative macros and relaxed plans interact with each other, and how macros can be used to improve the accuracy of the heuristic state evaluation. Based on macros' success in classical planning, research should be done on using macros in areas such as temporal planning and planning with uncertainty.

References

[Botea *et al.*, 2005] A. Botea, M. Müller, and J. Schaeffer. Learning Partial-Order Macros From Solutions. In *ICAPS-05*, pages 231–240, 2005.

[Coles and Smith, 2005] A. Coles and A. Smith. On the Inference and Management of Macro-Actions in Forward-Chaining Planning. In *UK Planning and Scheduling SIG*, 2005.

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

[Hernádvölgyi, 2001] I. Hernádvölgyi. Searching for Macro-operators with Automatically Generated Heuristics. In *Canadian Conference on AI*, pages 194–203, 2001.

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

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

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

[Korf, 1983] R. Korf. *Learning to Solve Problems by Searching for Macro-Operators*. PhD thesis, Carnegie-Mellon University, 1983.

[Korf, 1985] R. Korf. Macro-Operators: A Weak Method for Learning. *Artificial Intelligence*, 26(1):35–77, 1985.

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

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

[Mooney, 1988] R. Mooney. Generalizing the Order of Operators in Macro-Operators. In *ICML*, pages 270–283, 1988.

[Newton *et al.*, 2005] M. Newton, J. Levine, and M. Fox. Genetically Evolved Macro-Actions in AI Planning. In *UK Planning and Scheduling SIG*, 2005.

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