Empirical and Analytical Performance of Iterative Operators

Peter Shell *(pshell@cs.cmu.edu)*
Jaime Carbonell *(jgc@cs.cmu.edu)*

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
412-268-7651

Abstract

Macro-operators and chunks have long been used to model the acquisition and refinement of procedural knowledge. However, it is clear that human learners use more sophisticated techniques to encode more powerful operators than simple linear macro-operators: specifically, linear macro-operators cannot represent arbitrary repetitions of operators. This paper presents a process-model for the acquisition of *iterative* macro-operators, which are an efficient representation of repeating operators. We show that inducing iterative macro-operators from empirical problem-solving traces provides dramatically better efficiency results than simple linear macro-operators. This domain-independent learning mechanism is integrated into the FERMI problem-solver, giving more evidence that humans have a similar learning capability.

1. Introducing Iterative Macro-operators

One of the main goals of knowledge compilation is to reformulate or recycle problem solving knowledge in order to reduce the amount of search necessary to solve problems. A number of different methods for reducing search have evolved. For example, if the derivational analogy algorithm (Carbonell, 1986, Hickmanetal90, 1990) can retrieve the solution to a problem similar to the current problem, then it will adapt the previous solution to the new problem. Explanation-based learning (DeJong & Mooney, 1986), on the other hand, reformulates axioms which were used to solve a problem, into operators which can more efficiently solve the problem the next time it is encountered. Another reformulation method is chunking in Soar (Laird *et al*, 1986) and macro-operators (Anderson, 1983, Fikes, 1971). By summarizing the behavior of a sequence of forward-chaining operators, macro-operators also reduce the amount of operator-space search required by the problem-solver.

There are a number of problems with both linear macro-operators and chunks, which iterative macro-operators address. Simple linear macro-operators can never generalize to an arbitrary number of applications of atomic operators. For example, different macro-ops must be learned to solve the 2-disk tower of Hanoi, the 3-disk one, etc. It's clear that people have a more flexible learning method than linear macro-operators, and that they can somehow generalize to an arbitrary number of applications of an operator. Furthermore, both linear macro-ops and Soar chunks which encode multiple applications of a rule turn polynomial-time matches into exponential ones (Tambe, 1988). Iterative macro-operators, as depicted in figure 1-1, would solve the "expensive chunk" problem in SOAR if adapted to that paradigm. They also address the increased match-time search problem which occurs when large numbers of macro-operators are learned (Minton *et al*, 1988).

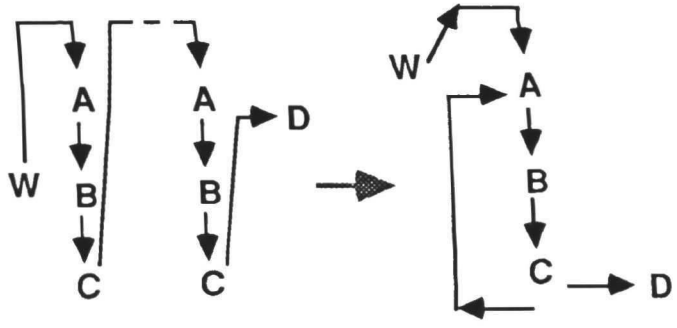

Figure 1-1: Formation of iterative macro-operators

In principle, there are two different ways in which macro-operators (iterative or otherwise) can reduce search. First, they can reduce the amount of search necessary in the operator state-space. Secondly, they can compile away temporary computation in the matching process itself. Thus match-level optimizations can reduce "fine-grain search" while operator-level aggregations can reduce "coarse-grain search".

Although previous approaches to iterative operators such as (Cheng & Carbonell, 1986, Riddle, 1988, Shavlik, 1990) address the coarse-granularity search reduction problem, they do not address the equally important issue of optimizing the macro-op itself. If one knows that a set of operators will be applied in a fixed

iterative sequence, then the operators themselves can be transformed into more efficient iterative macro-operators.

The transformation algorithm has been generalized since (Shell & Carbonell, 1989) to cover more domains, and is summarized in section 2. Section 4 presents empirical results in these new domains, showing speedups as high as 14-fold. This is significant given that the operators are implemented in a RETE-based system called FRulekit (Shell & Carbonell, 1986), which is already fairly efficient. Furthermore, new theoretical results in section 3 show that iterative operators significantly improve over linear operators in both the coarse-granularity operator search space and in the operator match space. These theoretical and empirical results of the process model give more credence to the idea that people have a similar technique for learning general iterative macro-operators.

2. Iterative-Operator Composition

To explain how iterative operators are faster than linear operators, this section will summarize the iterative-operator composition algorithm, and will show a sample iterative operator. A more detailed description of the algorithm is presented in (Shell and Carbonell, 1991).

Generating iterative-operators involves:

1. *Detecting the Repeated Operator.* When a linear operator or macro-operator repeats, it is eligible for transformation into an iterative macro-operator.

2. *Encapsulation of the Invariant Structure.* Variables matched in the preconditions that bind identically each cycle need not be rematched each iteration.

3. *Solution of Recurrence Relations.* Local data updates between each cycle (such as decrementing an array index) are extracted into a single step external to the loop. More complex or open-form updates remain inside the iteration loop.

4. *Composition of the Total Iterative Macro-operator.* The preconditions, iterative body and exit conditions are deduced from the source atomic operator sequence and compiled into production rules.

Iterative operators are efficient because conditions and actions in the source rule that are unnecessary are eliminated. Unnecessary conditions and actions are defined as:

- Conditions that match objects that never change while the iterative operator runs,
- Conditions that match objects that are modified by the operator,
- Actions that modify objects that are matched by the operator,
- And repeated actions which can be induced to a single step as a function of N.

All other conditions and actions are retained.

The iterative operator is always equivalent to the original. The outline of an iterative macro-operator is shown in figure 2-1. This operator is used in the FERMI system for solving systems of simultaneous algebraic equations.

```
RULE Pre-Iterative-solve
 CONDITIONS:
  If all conditions of solve-unknown exist,
  And we are not currently iterating
 ACTIONS:
  Add the goal to iterate solve-unknown

RULE Iterative-solve
 CONDITIONS:
  The goal is to iterate solve-unknown
  And all of the retained tests of
    solve-unknown are still satisfied
 ACTIONS:
  Perform the actions which were retained
    from the original operator.

RULE Post-Iterative-solve
 CONDITIONS:
  The goal is to iterate solve-unknown
  And the retained tests of solve-unknown
    are no longer satisfied
 ACTIONS:
  Remove the goal to iterate solve-unknown
  And update working memory with the actions
```

Figure 2-1: Iterative operator to find the next equation to eliminate, generated by FERMI.

3. Analytical Results

This section mathematically describes how much iterative operators can reduce search over linear operators. We derive complexity analyses of both the time for searching through a set of operators, and the time to match and execute individual operators. This analysis must be approximate, as there is not sufficient room to look in detail at each test and action of every operator.

3.1. Search Speed-up

This section presents an average-case complexity analysis of how much iterative operators reduce search in the operator space. Assume a breadth-first search discipline with **b** operators and a solution at depth **n**. The exponential complexity of search is: $\sum_{i=0}^{n} \mathbf{b}^i$, which is dominated by the last term: $\mathbf{b}^n$ for values of $b > 2$. If we introduce **c** linear macro-operators we can estimate the dominant term as:

$$(\mathbf{b} + \mathbf{c})^{n/(1+p_m(m-1))}$$

In the formula, **m** is the average length of a macro-operator (i.e., the number of base operators) and $\mathbf{p_m}$ is the ratio of macro-operators to the total (base and macro) operators in the solution sequence. Adding many useless macro-operators increases the base **b** + **c** considerably but does not reduce the exponent, whereas adding a few very useful operators reduces the exponent without significantly increasing the base.

An iterative macro-operator consists of repeated applications of a base operator or a linear macro-operator. An iterative macro-operator is operationally equivalent to **K** linear macro-operators, where M is the maximal number of iterations reasonable in the given domain. The search reduction, if all macro-operators could be reduced to iterative ones would be calculated by dividing the macro-operator branching factor by the average **K**:

$$(\mathbf{b} + \mathbf{c}/\mathbf{K})^{n/(1+p_m(m-1))}$$

The idea is to reduce search depth while minimally increasing search breadth. This translates into generating the most powerful possible macro-operators.

3.2. Match Speed-up

In addition to speeding up the operator search time, iterative operators match and execute faster than their linear counterparts. This section will mathematically analyze the amount of match speed-up realized by individual iterative macro-operators.

3.2.1. Sources of Speed-up

There are three different ways that iterative operators optimize the matching of linear operators:

1. They eliminate *condition elements*;
2. They eliminate *actions*;
3. They move tests to earlier parts of the rule.

The first two are the most common and will be the focus of this analysis. Eliminating unnecessary condition elements makes the rule faster by decreasing the number of tests which the rule must perform. Eliminating actions also speeds up the process since the actions cause the system to re-match.

3.2.2. The Cost of Performing an Action

The cost of performing an operator is the sum of the costs of the actions in that rule. The costs of those actions in turn depend on how much matching the rules perform. The cost $Act\text{-}cost_a$ of performing any action a in a RETE-based operator is derived in the report (Shell and Carbonell, 1991). It is:

$$Act\text{-}cost_a = \beta_{n-2}C + AC\beta_{n-2}\sum_{i=1}^{L-n}(AD)^{i-1}$$

Where:

$\underline{A}$ is the average size of the input;
$\underline{\beta_n}$ is the number of tuples of objects which have successfully combined at condition element n;
$\underline{C}$ is the average cost of testing a condition element;
$\underline{D}$ tells us how discriminating the conditions are. If D is small then few objects pass the tests in the conditions. If D is big then there are more partial matches. I.e., D times $[\beta_n]$ times A equals $[\beta_{n+1}]$;
and $\underline{L}$ is the number of conditions in the rule.

3.2.3. Speed-up of the Iterative-Operator

As can be seen from the last equation, the speed of a rule is determined by the (AD) product - the rate at which the size of the partial match grows. In expensive rules, the number of partial matches grows as the RETE net is traversed. Thus, (AD) is greater than 1 and the cost increases exponentially.

In this case, the second half of the above equation dominates. Suppose that in the iterative operator, one condition element has been removed. Then the cost of an action in this operator would be approximately:

$$AC\beta_{n-2}\sum_{i=1}^{L-n-1}(AD)^{i-1}$$

Thus the speed-up of the iterative operator over the linear operator, is the quotient:

$$\frac{AC\beta_{n-2}\sum_{i=1}^{L-n}(AD)^{i-1}}{AC\beta_{n-2}\sum_{i=1}^{L-n-1}(AD)^{i-1}}$$

$$= \frac{\sum_{i=1}^{L-n}(AD)^{i-1}}{\sum_{i=1}^{L-n-1}(AD)^{i-1}}$$

which, as L increases, approaches $\underline{AD}$.

When t conditions are removed from the iterative operator, the speed-up is:

$$\frac{\sum_{i=1}^{L-n}(AD)^{i-1}}{\sum_{i=1}^{L-n-t}(AD)^{i-1}}$$

which, as L increases, approaches $\underline{(AD)^t}$.

In efficient rules such as the ones which were profiled for this paper, the product AD is 1 for some conditions and greater than 1 for others. I.e., the partial match does not always grow. Thus, the speed-up will be polynomial or linear instead of exponential, depending on how many expensive tests there are.

4. Empirical Results

Timings of the iterative-operator learning method on three different domains is shown here. We first present a summary of the efficiency gains in each domain, and then show graphs of the performance improvement of an iterative-operator in two of the domains.

4.1. Speed-up of Iterative Operators

The domains for which iterative operators were tested range from a nine-rule circuits problem solver to a more realistic ninety-rule expert system. The second column shows the total run-time speed-up (time required to run the linear operator divided by the time required to run the iterative operator). The next column shows the speed-up in match time, which is the amount of time spent matching in the RETE net.

Speed-up of Iterative Operators			
Domain	Run-time Speed-up	Match-time Speed-up	# Rules
Circuits	1.69 x	1.88 x	9
Algebra	4.66 x	4.72 x	23
Machining	11.3 x	14.3 x	90

As the table shows, the larger the domain, the better the speed-up.

4.2. Detailed Timings

For two of the above domains, the match times will be graphed for the iterative macro-operator, the linear macro-operator, and the set of base operators without any macro-operators. The input-size is plotted on the X-axis and the match time is on the Y-axis. As we shall see, the iterative operator was always faster than the original ones.

The Algebra Domain. The algebra system is a set of productions which solve simultaneous systems of N equations in N unknowns. It is a module of the FERMI general scientific problem-solver (Larkin, Reif & Carbonell, 1986). The X-axis plots the number of equations and unknowns given to the system (figure 4-1).

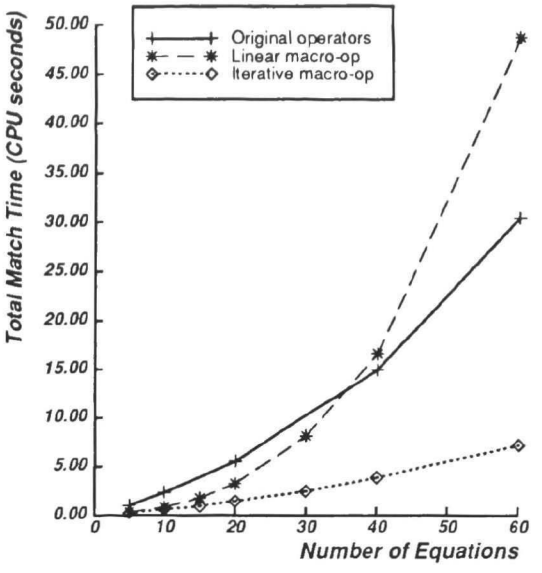

Figure 4-1: Time to Solve Systems of Equations

As the graphs show, the "expensive macro-operator" (Minton *et al*, 1988) problem rears its head again. The linear macro-operator is initially faster than the original rules, but as the size of the input increases, it becomes more expensive. However, the iterative macro-op eliminates the expensive part of the linear macro-op and is much faster than both the linear macro-op and the original rules.

The Machining Domain. The machining domain is an expert system for process planning. The task is to produce machine parts that have specified characteristics, such as a rectangular block of given dimensions made with a certain material and containing a centered hole. This domain is the largest production set that the iterative-operator algorithm has been tested on yet, and it shows the best improvement over linear operators. The X-axis displays the number of holes it drills (figure 4-2).

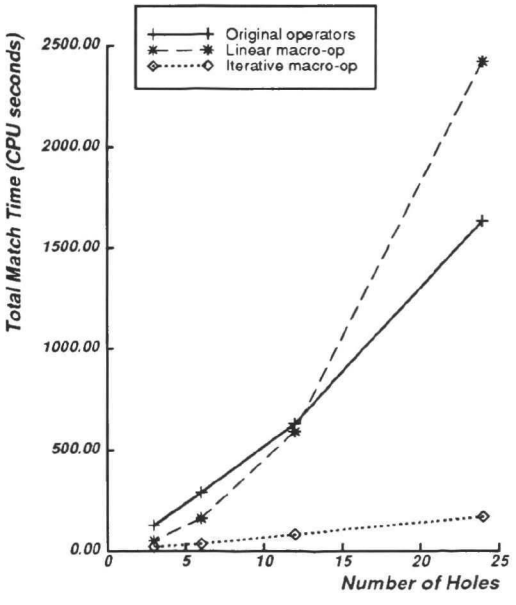

Figure 4-2: Time to Drill N Holes

As in the algebra domain, the linear macro-op is more expensive than the original operators, but the iterative operator is faster than both.

5. Conclusions

Iterative operators generalize to N by efficiently composing recursive subsequences. They have been shown to improve on linear operators by as much as 14-fold. As we have seen through complexity analysis and detailed empirical results, iterative operators effectively

address some fundamental shortcomings of the standard macro-operator and chunking knowledge compilation methods. They do this by requiring a smaller number of total macro-operators, avoiding the "expensive-chunk" and "expensive-macro-operator" problems, and by making the macro-operators more efficient.

Although this paper makes no claims about how people form iterative macro-operators, this production-system model for iterative macro-operators may be a starting point for a cognitive model. Key questions to ask would be:

- Under what conditions do people form iterative rather than linear macro-operators?
- How do people form iterative macro-operators?

6. Acknowledgments

We would like to thank Patty Cheng for opening the door to a much more powerful class of macro-operators; and Angela Hickman, Dan Kahn and Ben MacLaren for helpful comments on earlier drafts of this paper.

References

Anderson, J. A. (1983). Acquisition of Proof Skills in Geometry. In R. S. Michalski, J. G. Carbonell and T. M. Mitchell (Eds.), *Machine Learning, An Artificial Intelligence Approach*. Palo Alto, CA: Tioga Press.

Carbonell, J. G. (1986). Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), *Machine Learning, An Artificial Intelligence Approach, Volume II*. Morgan Kaufmann.

Cheng, P. W. and Carbonell, J. G. (1986). Inducing Iterative Rules from Experience: The FERMI Experiment. *Proceedings of AAAI-86*. .

DeJong, G. F. and Mooney, R. (1986). Explanation-Based Learning: An Alternative View. *Machine Learning Journal*, Vol. *1*(2).

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

Hickman, A. Kennedy, Shell P. and Carbonell, J. G. (1990). Internal Analogy: Reducing Search during Problem Solving. Catherine Copetas (Ed.), *The Computer Science Research Review 1990*. , The School of Computer Science, Carnegie Mellon University.

Laird, J. E., Rosenbloom, P. S. and Newell, A. (1986). Chunking in SOAR: The Anatomy of a General Learning Mechanism. *Machine Learning*, *1*(1), 11-46.

Larkin, J., Reif, F. and Carbonell, J. G. (1986). FERMI: A Flexible Expert Reasoner with Multi-Domain Inference. *Cognitive Science*, Vol. *9*.

Minton, S. N., Carbonell, J. G., Knoblock, C. A. and Kuokka, D. R. (1988). Explanation-Based Learning: Improving Problem Solving Performance through Experience. In Carbonell, J. (Ed.), *Paradigms for Machine Learning*. (to appear).

Riddle, Patricia J. (1988). An Approach for Learning Problem Reduction Schemas and Iterative Macro-operators. *Proceedings of the First International Workshop in Change of Representation and Inductive Bias*. .

Shavlik, Jude W. (1990). Acquiring Recursive and Iterative Concepts with Explanation-Based Learning. *Machine Learning*, *5*(1), 39-70.

Shell, P. and Carbonell, J. G. (1986). The FRuleKit Reference Manual. CMU Computer Science Department internal paper.

Shell, P. and Carbonell, J. G. (1989). Towards a General Framework for Composing Disjunctive and Iterative Macro-operators. *Proceedings of IJCAI-89*. .

Shell, P. and Carbonell, J. (1991). *A General Framework for Composing Iterative Macro-Operators*. (Tech. Rep.). Carnegie Mellon University School of Computer Science. Forthcoming.

Tambe, M. and Newell, A. (1988). Some Chunks are Expensive. *Proceedings of the Fourth International Workshop on Machine Learning*. .