The FERMI System: Inducing Iterative Macro-operators from Experience

Patricia W. Cheng and Jaime G. Carbonell
Computer Science Department
Carnegie-Mellon University
Pittsburgh PA 15213

Abstract

Automated methods of exploiting past experience to reduce search vary from analogical transfer to chunking control knowledge. In the latter category, various forms of composing problem-solving operators into larger units have been explored. However, the automated formulation of effective macro-operators requires more than the storage and parametrization of individual linear operator sequences. This paper addresses the issue of acquiring *conditional* and *iterative operators,* presenting a concrete example implemented in the FERMI problem-solving system. In essence, the process combines empirical recognition of cyclic patterns in the problem-solving trace with analytic validation and subsequent formulation of general iterative rules. Such rules can prove extremely effective in reducing search beyond linear macro-operators produced by past techniques.*

1. Introduction

Automated improvement of problem-solving behavior through experience has long been a central objective in both machine learning and problem solving. Starting from STRIPS [9], which acquired simple macro-operators by concatenation and parameterization of useful operator sequences, chunking control knowledge has proven a popular method for reducing search in solving future problems of like type. More comprehensive chunking disciplines have been studied; for instance, SOAR [18] chunks at all possible decision points in the problem solving, whereas MORRIS [15] and PRODIGY [14] are more selective in their formulation of useful macro-operators. Other forms of learning particularly relevant to problem solving include strategy acquisition [11, 17], and various forms of analogical reasoning. Transformational analogy [5] transfers expertise directly from the solution of past problems to new problems that bear close similarity, and derivational analogy [6] transfers problem-solving strategies across structurally similar problem-solving episodes. Both forms of analogy provide the positive and negative exemplar data required to formulate generalized plans [4, 7, 8, 11, 16, 17, 20].

This paper discusses the need for the formulation of a more general class of macro-operators that enable conditional branching and generalized iteration. It then presents a method for automated induction of such macro-operators from recursive and iterative problem-solving traces. Inducing iterative rules (macro-operators) from behavioral traces involves the detection of repetitive patterns in the subgoal structure of the problem-solving episodes. This process includes analysis of the trace to determine the common operations at an appropriate level of abstraction, and extraction of conditions necessary for success.

Major savings in problem-solving efficiency accrue not just from collapsing many rules into a single one with an iterative right-hand-side, but also from advancing tests necessary for success from arbitrary points in the problem-solving process to up-front left-hand-side (LHS) conditions. For instance, an iterative rule acquired by FERMI solves independent linear equations in multiple unknowns by repeated substitution of expressions containing progressively fewer variables. This (or any other) method can yield a unique solution only if there are as many linearly independent equations as there are variables. Such a condition is deduced automatically by analysis of the problem and is subsequently added to the LHS of the iterative rule, eliminating the need to perform all the step-by-step substitutions in order to discover at the end of the process that there are remaining variables and no remaining equations, or that there is a contradiction. The techniques for developing and implementing this type of learning, as elaborated in subsequent sections, provide a useful addition to the repertoire of machine learning methods in problem solving.

2. Overview of FERMI

FERMI**, our experimental testbed for iterative rule induction, is a general problem solver in the natural sciences. Its flexible architecture has been described elsewhere [3, 13]; here we focus only on those aspects directly relevant to automated induction of iterative rules. FERMI separates problem solving knowledge from domain knowledge, representing the former as strategies and the latter as factual frames at different levels of abstraction in a semantic frame network. Thus, general concepts such as conservation of mass or equilibrium conditions need be represented only once and inherited where appropriate. Similarly, problem-solving knowledge, such as iterative decomposition***, are encoded as general strategies applicable to a wide variety of problems. FERMI has successfully solved problems in areas as diverse as fluid statics, linear algebra, classical mechanics, and DC circuits applying the same general problem solving strategies, and some of the same general domain concepts.

We have implemented several different control strategies for FERMI, experimenting most extensively with an augmented means-ends method, and more recently with analogical transfer, and with rule-based forward chaining. This paper confines its discussion to the third technique, whose implementation was greatly facilitated by the use of RuleKit [19], a new rule-based language improving upon OPS5 by accessing structured frame representations and providing a multi-level task agenda to prioritize rules dynamically. The overall control strategy was inspired by Larkin's study of expert problem solving protocols [12], but its real utility takes the form of a multi-level trace mechanism that enables FERMI to introspect on its own problem-solving behavior at an appropriate level of granularity to formulate new problem solving operators, including the conditional iterative operators discussed in the following section. The trace mechanism keeps track of the emerging

*The research reported in this paper was funded in part by the Office of Naval Research under grants number N00074-82-50767 and number N00014-84-K-0345 and in part by a gift from the Hughes Corporation. We thank every member of the FERMI project -- Angela Gugliotta, Angela Hickman, Jill Larkin, Fred Reif, Peter Shell, and Chris Walton -- for their valuable discussions. We are especially grateful to Peter Shell and to Chris Walton for their indispensible help on using Rulekit and on programming respectively.

**FERMI is an acronym for Flexible Expert Reasoner with Multi-Domain Inference, and a tribute to Enrico Fermi, who displayed abilities to solve difficult problems in many of the natural sciences by the application of general domain principles and problem solving strategies.

***Iterative decomposition proceeds as follows: 1. Isolate the largest manageable subproblem. 2. Solve that subproblem by direct means. 3. If nothing remains to be solved, compose the solutions to the subproblems into a solution to the original problem. 4. If part of the problem remains to be solved, check whether that remaining part is a reduced version of the original problem. 5. If so, go to 1, and if not halt with failure.

goal-subgoal tree, the methods used to attack the each problem, and the causes of success or failure at every intermediate step in the reasoning.

3. Acquiring Iterative Macro-operators

Many problems share an implicit recursive or iterative nature. These problems include mundane everyday activities such as walking until a destination is reached and eating until hunger is satisfied as well as problems in mathematics and science such as those solved by FERMI. The underlying recursive or iterative structure may become apparent only through analysis of a successful solution path produced by the problem solver. The detection of an iterative pattern in the solution trace starts by pruning unsuccessful branches in the search tree and removing irrelevant steps, if any. Then, the recurring pattern in the trace sequence must be identified – a non-trivial process, as there may be no repetition at the instantiated operator level, but rather at the more abstract level of recurring changes in the subgoal structure. After detection, our learner will establish the conditions for iteration, extract the necessary components from the operator sequences and state descriptions, and construct a conditional iterative macro-operator. The detailed description of this process and its implementation in FERMI follows in the subsequent sections.

When given a trace that exhibits a fixed number of iterative cycles before solution is reached, current methods of forming macro-operators such as STRIPS, ACT*, and SOAR [1, 9, 10, 18] cannot produce operators that will generalize to an arbitrary number of iterations. Indeed, they cannot even detect the iterative nature of the problem. The MACROPS facility in STRIPS [9], for instance, would add all subsequences of primitive operators for as many cycles as the instance problem required into its triangle table – generating huge numbers of macro-operators and failing to capture the cyclic nature of the solution. Anderson's ACT* [1, 2] would compile one (or more) linear macro-operators for each number of repetitions, also failing to capture the iteration. Thus, for any single cycle of iteration, existing macro-operator formation systems will, at best, produce macro-operators that will apply to a predetermined number of iterations, which would not generalize to a fewer or greater number of cycles. Moreover, as we remarked earlier, each cycle may select different methods for solving the same subgoals, and the regularity exists at a higher level of abstraction in the subgoal trace. Most earlier systems (SOAR partially excepted) do not chunk problem solving traces at higher level of abstraction than the sequence of instantiated operators. As we will illustrate with an example problem in the familiar domain of solving simultaneous linear equations, the exact sequence of rules may vary from cycle to cycle while preserving an overall subgoal structure.

The learning in our program proceeds in three steps:

1. detection of an iterative pattern in the solution trace at the appropriate level of abstraction and granularity,
2. formation of a macro-operator that transforms a state at the beginning of a single iterative cycle to the state at the beginning of the next cycle, and
3. formation of an iterative operator that checks for generalized applicability conditions inferred from the macro operator together with conditions immediately following the iterative sequence in the successful solution trace.

Below we elaborate on each step with illustrations drawn from our example problem on solving simultaneous linear equations. A set of operators for solving such systems of equations is listed in Figure 1. The operators are all in the form of standard condition-action rules. (Variables in the figure are preceded by an = sign, all LHS conditions are conjoined, and all RHS actions are evaluated sequentially.) A trace using these operators solving an algebra problem is shown in Figure 2. The solution path involves: 1. selecting an appropriate variable, 2. rearranging an equation to express this variable in terms of others, if the equation does not already appear in that form, 3. substituting the equivalent expression for the variable whenever the variable occurs in the remaining set of equations, 4. eliminating the equation used for substitution, and 5. repeating the above steps until only an equation that contains no variable other than the desired unknown remains.

3.1. Pattern Detection

What type of repetitive pattern in the solution trace would warrant the formation of an iterative rule? We think that requiring identical sequences of rules would be too restrictive, because partially matched sequences may nonetheless contain information on equivalent choices and orderings of operators. Consider repeated instances of the same subproblem – say to establish a precondition on occasions when it is not already satisfied. The instances may (or may not) require different operators. In our algebra example, after the execution of the rule select-var+, if the problem state happens to include an equation that has the variable returned by select-var+ on its LHS, then the rule var-on-lhs+ would apply. Otherwise, rule·var-on-lhs- would have to be executed before var-on-lhs+ applies. Thus, what rule follows select-var+ could vary depending on the particular problem state. Nonetheless, the specification that either var-on-lhs- or var-on-lhs+ – and not other operators irrelevant to variable substitution – follows the execution of select-var+ is useful. It reduces the number of matches to be done by an amount proportional to the number of operators excluded. Notice that the two alternative rules are different paths satisfying the same subgoal. To capture information on sequencing that is common across differing circumstances, our pattern detector looks for consecutive, identical repetition of any sequence of *changes in subgoals*. Subgoals are generated and removed by rules during problem solving. Each column in Figure 3 shows a type of trace of the solution path for the example problem in Figure 2:

1. a trace of the sequence of operators executed,
2. a trace of the sequence of subgoals for these operators, and
3. a trace of the sequence of changes in these subgoals.

As can be seen, consecutive identical repetition is apparent only in the last type of trace. (Identical repetitions are bracketed in the figure.) The second type of trace varies across cycles because the number of rules required to satisfy a subgoal may vary from cycle to cycle.

3.2. Formation of Macro-operators with Conditionals

After an iterative pattern is detected, the program forms a macro-operator by composing rules in a single cycle of the iteration as an intermediate step towards forming an iterative operator. The sequence of operators that should be composed is therefore determined by the pattern detected and is less arbitrary than in systems such as STRIPS [9] and ACT* [2] in which any sequence can be a candidate for a macro-operator. Although our trigger for forming a macro-operator differs from others, the actual formation is in the tradition of macro-operator learning systems such as STRIPS and ACT*, with the exception that we allow for alternative actions conditional on the problem state within the same operator. The greater generality engendered by this feature helps avoid the proliferation of macro-operators in a problem solver [15, 14]. Assuming that each conditional consists of a simple if-then-else branch, that there is a series of n conditionals in a cycle of iteration, and that these conditionals are independent of each other, the number of traditional macro-operators – which do not allow internal conditionals – required to cover the same state space would be 2^n.

Internal conditionals are implemented in our program by an *agenda* control structure on the right-hand-side (RHS) of the macro-operator. An agenda consists of an ordered list of "buckets" that contain ordered sets of operators. In our program the buckets, and the operators in each bucket, are tested in order. When an operator does not apply, the next operator in the same bucket is tested. When an operator does apply, control is returned to the first operator in the bucket. Control is passed on to the next bucket when no rule in the bucket applies or when a rule that applies halts execution of the bucket. When no rule in the last bucket applies or when it halts, control leaves

name of rule	LHS: conditions	RHS: actions
solve-unknown-1-equation	the current goal is to solve for unknown, the number of equations is 1, the number of variables is 1, the desired unknown is =u, there is an equation =e that contains =u, there are no other equations	solve for =u in =e, and pop success
solve-unknown-n-equations	the current goal is to solve for unknown, the number of equations is > 1, the number of variables is > 1, the desired unknown is =u, there is no equation that contains =u and has no other variables in it	set up subgoals to (1) select a variable for substitution (2) get an equation with the selected variable on its LHS (3) form a new equation by substitution (4) solve for unknown
selectvar+	the current goal is to select a variable for substitution, the desired unknown is =u, there is a variable =v that is not =u, and =v appears in at least 2 equations	mark =v as selected, and pop success
select-var−	the current goal is to select a variable for substitution, there is no variable that is not the unknown and appears in at least 2 equations	pop failure
var-on-lhs+	the current goal is to get an equation with the selected variable on its LHS, the selected variable is =v, and there is an equation with =v on its LHS	mark the equation as selected, and pop success
var-on-lhs−	the current goal is to get an equation with the selected variable on its LHS, the selected variable is =v, and there is no equation with =v on its LHS, but there is an equation =e that contains =v	rearrange =e so that =v is on its LHS
replace+	the current goal is to form a new equation by substitution, the selected equation is =e1, the selected variable is =v, and =v occurs in a second equation =e2	substitute occurrences of =v with the RHS of =e1
replace−	the current goal is to form a new equation by substitution, the selected equation is =e, the selected variable is =v, no equation other than =e contains =v, the number of equations is =nume, and the number of variables is =numv	remove =v from working memory, remove =e from working memory, set =nume to (=nume − 1) set =numv to (=numv − 1) and pop success

Figure 1: operators for algebra problem

the agenda and is returned to the top-level.

We exploit this control structure by placing in each agenda bucket a disjunctive set of operators for satisfying the same subgoal. The automated learner puts operators in a bucket in an agenda when it detects in the solution trace sets of operators that have the same subgoal on their LHS but each member of the set has condition elements that negate condition elements in each of the other members of the set. The negated condition elements obviously cannot be composed on the LHS of a macro-operator, and are instead left to form separate operators in a bucket on the RHS of the macro-operator. In each bucket the operator that checks for satisfaction of the subgoal – and therefore halts execution of the bucket when its conditions are satisfied – is placed in the first position. In such manner conditional branches are formulated in new macro-operators without altering the uniform top-level control structure.

In our algebra example there are two sets of conditionals, with two operators in each set: var-on-lhs+ and var-on-lhs- in one set, and replace- and replace+ in the other set. The first set is a simple if-then-else conditional, the second set is repeatedly tested until replace- is applicable, i.e., when there are no more occurrences of the variable to be replaced. A macro-operator with conditionals for our example algebra problem is shown in Figure 4a.

With the exception of negated condition elements and their corresponding RHSs, whenever we compose a sequence of operators, we aggregate the condition and action elements of the sequence of operators applied in the trace, as in other proposed methods of forming macro-operators (e.g., [1]). We eliminate redundancies in the macro-operator by deleting:

1. duplicate condition elements,
2. condition elements that match a working memory element, including subgoals, created by an earlier rule within the sequence,
3. action elements that create subgoals matched by subsequent condition elements in the sequence, and
4. condition and action elements whose sole function is to pass variables bindings from one rule to the next.

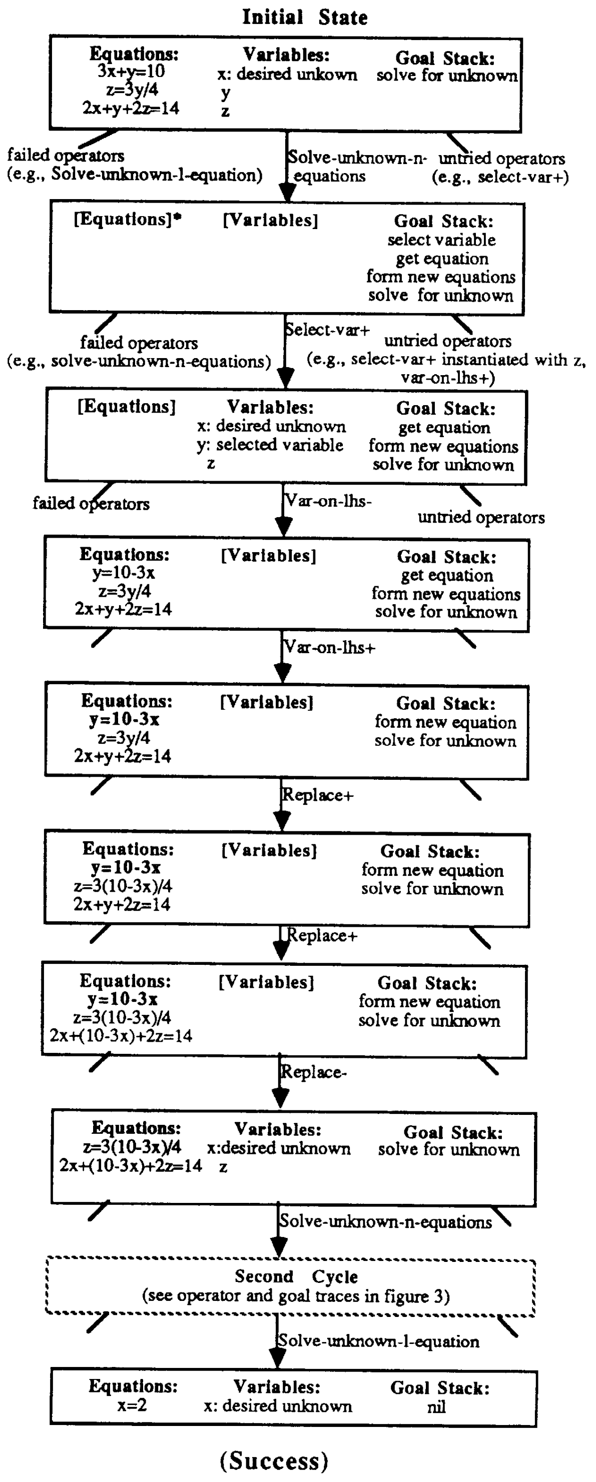
Initial State
Equations: 3x+y=10 z=3y/4 2x+y+2z=14
Variables: x: desired unkown y z
Goal Stack: solve for unknown
failed operators (e.g., Solve-unknown-1-equation)
Solve-unknown-n-equations
untried operators (e.g., select-var+)
[Equations]* [Variables]
Goal Stack: select variable get equation form new equations solve for unknown
failed operators (e.g., solve-unknown-n-equations)
Select-var+
untried operators (e.g., select-var+ instantiated with z, var-on-lhs+)
[Equations]
Variables: x: desired unknown y: selected variable z
Goal Stack: get equation form new equations solve for unknown
failed operators
Var-on-lhs-
untried operators
Equations: y=10-3x z=3y/4 2x+y+2z=14
[Variables]
Goal Stack: get equation form new equations solve for unknown
Var-on-lhs+
Equations: y=10-3x z=3y/4 2x+y+2z=14
[Variables]
Goal Stack: form new equation solve for unknown
Replace+
Equations: y=10-3x z=3(10-3x)/4 2x+y+2z=14
[Variables]
Goal Stack: form new equation solve for unknown
Replace+
Equations: y=10-3x z=3(10-3x)/4 2x+(10-3x)+2z=14
[Variables]
Goal Stack: form new equation solve for unknown
Replace-
Equations: z=3(10-3x)/4 2x+(10-3x)+2z=14
Variables: x:desired unknown z
Goal Stack: solve for unknown
Solve-unknown-n-equations
Second Cycle (see operator and goal traces in figure 3)
Solve-unknown-1-equation
Equations: x=2
Variables: x: desired unknown
Goal Stack: nil
(Success)

Figure 2: Trace of states and operators in an example algebra problem

* Square brackets indicate that the elements enclosed are unchanged.

Because justifications for the above deletions have been discussed elsewhere (e.g., [1, 2]), we are not repeating them in this paper.

3.3. Formation of Iterative Operators

The branching feature above is desirable for an iterative operator because it allows for variation from cycle to cycle. Without it, an important class of iterative operators could not be represented or acquired. The saving due to internal branching on the number of equivalent traditional macro-operators increases exponentially with the number of iteration cycles considered. For problems requiring exactly m iterative cycles with n independent if-then-else conditionals in each cycle, the number of traditional macro-operators required to cover the state space is 2^{nm}. Thus, for problems requiring iteration up to m cycles, the total number of macro-operators grows to $\Sigma_{i=1,m} 2^{ni}$. In contrast, a single iterative, conditional macro-operator independent of m and n suffices. In view of the number of macro-operators required, unless iterative operators are formed, it could easily be less efficient to search through the large space of macro-operators than the original space of operators in problem domains involving iteration. The inefficiency is exacerbated by the fact that the myriad specific macro-operators would share significant common substructure. Restricting ourselves to non-iterative operators would therefore severely limit useful learning in such domains.

An important additional advantage of forming iterative operators is that certain algebraic modifications in the intermediate macro-operator can be related to the number of iterations i. Given a trace of a successful path, we can form equations with variables in these algebraic modifications expressing conditions under which a solution will be reached through the iterative procedure. If there are multiple modifications of this sort, variables in these modifications can be related to each other through i. The inferred relation can help detect solvability by the iterative rule early, before iteration is actually entered. Let us illustrate this principle in our algebra example. As can be seen in Figure 3, each cycle through this macro-operator reduces the number of equations and the number of variables each by 1. The reduction for i cycles would be $(q - i)$ and $(v - i)$ respectively, where q is the number of equations given and v is the number of variables in the given equations. From the solution trace, we know that when the number of equations and number of variables are both 1, i.e., when

$$q - i = 1 \text{ and}$$

$$v - i = 1,$$

then a solution can be reached. Eliminating i from the above equations, we get

$$q = v.$$

Putting this inferred relation in the LHS of the iterative rule helps screen out insoluble problems without actually iterating through the solution procedure. Other information can be similarly precomputed and fronted as operational conditions on the LHS of new iterative operators.

The iterative operator formed in our example problem is presented in Figure 4b. In FERMI, the LHS of iterative operators is formed by an aggregate of condition elements that need no iteration. They are:

1. condition elements in the LHS of the intermediate macro-operator with variables or constants that are not modified by the operator,
2. condition elements whose variables undergo simple algebraic modifications by the operator – modifications such as addition, multiplication, division by a constant or variable, etc., and
3. checks on relations between the above variables inferred through the successful solution trace and the number of iterations – checks such as equating the number of unknowns to the number of equations in our example.

The RHS of the iterative operator consists of

1. a statement initializing a counter for the number of iterations,
2. an iterative agenda call to the intermediate macro-operator formed earlier (See Figure 4b),
3. and simple algebraic modifications based on the number of iterations. For instance, the number of equations in our example algebra problem is reduced by the number of iterations.

trace of rules	trace of subgoals of rules	trace of changes in subgoals
solve-unknown-n-equations	solve for unknown	solve for unknown
select-var+	select variable	select variable
var-on-lhs-	get equation for substitution	get equation for substitution
var-on-lhs+	get equation for substitution	
replace+	form new equation	form new equation
replace+	form new equation	
replace-	form new equation	
solve-unknown-n-equations	solve for unknown	solve for unknown
select-var+	select variable	select variable
var-on-lhs+	get equation for substitution	get equation for substitution
replace+	form new equation	form new equation
replace-	form new equation	
solve-unknown-1-equation	solve for unknown	solve for unknown
(sucess)	(success)	(success)

Figure 3: Three types of traces of the example problem in Figure 2*

*Information extracted from the same problem-solving step appears in the same row

The macro-operator halts when its conditions are no longer satisfied. Note that the iterative call to the macro-operator is the only truly iterative component required.

To coordinate with the iterative operator, the intermediate macro-operator (in the RHS of the iterative operator) is modified as follows: the first two kinds of condition elements just listed are removed from its LHS and the LHSs of operators in the agenda in its RHS. These are elements that require no iteration and have been moved to the LHS of the iterative operator. Corresponding action elements, those that do simple algebraic modifications on the variables in condition elements, are also removed. Such condition elements are no longer necessary because these modifications are done more efficiently in the RHS of the iterative operator

name of rule	LHS: conditions	RHS: actions
m-solve-unknown	the current goal is to solve for unknown,* the desired unknown is =u,* the number of equations =nume is > 1,* the number of variables =numv is > 1,* there is a variable =v that is not =u, =v appears in at least 2 equations, there is no equation that contains =u and has no other variables in it	call agenda with bucket 1: (m-var-on-lhs+ m-var-on-lhs-) bucket 2: (m-replace- m-replace+)
m-var-on-lhs+	there is an equation =e with =v on its LHS	mark equation as selected, and halt bucket
m-var-on-lhs-	there is no equation with =v on its LHS, but there is an equation =e that contains =v	rearrange =e so that =v is on its LHS
m-replace-	the selected equation is =e, no equation other than =e contains =v, the number of equations is =nume,* and the number of variables is =numv*	remove =e from working memory remove =v from working memory set =nume to (=nume - 1)* set =numv to (=numv - 1)* and halt bucket
m-replace+	the selected equation is =e1, and =v occurs in a second equation =e2	substitute occurrences of =v with the RHS of =e1

Figure 4a: Intermediate Macro-operator formed from a single cycle

* Elements marked with an asterisk are removed when the iterative operator is formed, and a counter =iterations for the number of iterations is added.

name of rule	LHS: conditions	RHS: actions
i-solve-unknown	the current goal is to solve for unknown, the desired unknown is =u, the number of equations =nume is > 1, the number of variables =numv is > 1, =nume is equal to =numv	set =iterations to 0, call agenda with bucket: (m-solve-unknown), set =nume to (=nume - =iterations), set =numv to (=numv - =iterations)

Figure 4b: Iterative operator

in a single step by relating the modifications directly to the number of iterations. In their place a counter for the number of iterations is added.

FERMI solves many problems requiring iteration, including simultaneous algebraic equations and physics problems such as finding the pressure difference between 2 points **a** and **b** in a container containing multiple layers of liquids that have various densities. A path from **a** to **b** is repeatedly decomposed until the requirements for applying the formula for pressure difference in a single liquid are met. The iterative operator to be learned from the problem-solving process is equivalent to the formula

$$\Delta p_{ab} = g \Sigma_{i=1,n} \rho_i \Delta h_i,$$

where Δp_{ab} is the pressure difference between **a** and **b**, g is the surface gravity, i is the summation index, n is the total number of liquids between **a** and **b**, ρ_i is the density of $liquid_i$, and Δh_i is the change in height of a path from **a** to **b** in $liquid_i$.

4. Concluding Remarks

Learning in problem solving requires more than rote memorization of linear operator sequences into macro-operators [15]. Parametrizing the macro-operators, and reducing redundant condition and action elements provides only the first step towards more general strategy learning. In the FERMI project we have gone two steps further:

1. automated generation of macro-operators with conditional branching, and
2. automated creation of iterative macro-operators to solve problems with cyclic subgoal structure.

The integrated implementation in FERMI of these two techniques, on top of the traditional macro-operator formation method, provides a theoretical enhancement in the power of macro-operators, and major savings in both the number of requisite macro-operators and the time required to search for applicable operators in future problem solving. Further work should be done on correcting over-generalization in iterative rules by learning from failure, generalizing types of inferences that can be made from the iterative trace to produce up-front LHS tests for an iterative operator, and on specifying types of condition elements that can be transferred from the intermediate macro-operator to the LHS of the iterative operator so as to improve early detection of solvability.

In addition to a larger scale implementation and more extensive testing of our iterative macro-operator formation techniques, future directions for learning in FERMI include:

- Incorporating analogical reasoning techniques [5, 6], which can provide a basis for transferring powerful macro-operators across related domains, as well as the more traditional transfer of solution sequences across related problems.
- Exploring the role of automatic programming in the creation of ever more elaborate macro-operators. Thus far, our primary effort has been in detection, analysis, and evaluation of problem solving traces in order to extract all the information required to formulate useful, generalized macro-operators. But, as the complexity of the task increases, so does the necessity for principled automatic synthesis of such macro-operators.

5. References

1. Anderson, J. R., *The Architecture of Cognition,* Cambridge, Mass.: Harvard University Press, 1983.
2. Anderson, J. A., "Acquisition of Proof Skills in Geometry," in *Machine Learning, An Artificial Intelligence Approach,* R. S. Michalski, J. G. Carbonell and T. M. Mitchell, eds., Tioga Press, Palo Alto, CA, 1983.
3. Carbonell, J. G., Larkin, J. H. and Reif, F., "Towards a General Scientific Reasoning Engine," Tech. report, Carnegie-Mellon University, Computer Science Department, 1983, CIP #445.
4. Carbonell, J. G., "Experiential Learning in Analogical Problem Solving," *Proceedings of the Second Meeting of the American Association for Artificial Intelligence,* Pittsburgh, PA, 1982.
5. Carbonell, J. G., "Learning by Analogy: Formulating and Generalizing Plans from Past Experience," in *Machine Learning, An Artificial Intelligence Approach,* R. S. Michalski, J. G. Carbonell and T. M. Mitchell, eds., Tioga Press, Palo Alto, CA, 1983.
6. Carbonell, J. G., "Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition," in *Machine Learning, An Artificial Intelligence Approach, Volume II,* Michalski, R. S., Carbonell, J. G. and Mitchell, T. M., eds., Morgan Kaufmann, 1986.
7. Dietterich, T. and Michalski, R., "Inductive Learning of Structural Descriptions," *Artificial Intelligence,* Vol. 16, 1981.
8. Dietterich, T. G. and Michalski, R. S., "A Comparative Review of Selected Methods for Learning Structural Descriptions," in *Machine Learning, An Artificial Intelligence Approach,* R. S. Michalski, J. G. Carbonell and T. M. Mitchell, eds., Tioga Press, Palo Alto, CA, 1983.
9. Fikes, R. E. and Nilsson, N. J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence,* Vol. 2, 1971, pp. 189-208.
10. Laird, J. E., Rosenbloom, P. S. and Newell, A., "Chunking in SOAR: The Anatomy of a General Learning Mechanism," *Machine Learning,* Vol. 1, 1986.
11. Langley, P. and Carbonell, J. G., "Language Acquisition and Machine Learning," in *Mechanisms for Language Acquisition,* MacWhinney B., ed., Lawrence Erlbaum Associates, 1986.
12. Larkin, J. H., "Enriching formal knowledge: A model for learning to solve problems in physics," in *Cognitive Skills and their Acquisition,* J. R. Anderson, eds., Lawrence Erlbaum Associates, Hillsdale, NJ, 1981.
13. Larkin, J., Reif, F. and Carbonell, J. G., "FERMI: A Flexible Expert Reasoner with Multi-Domain Inference," *Cognitive Science,* Vol. 9, 1986.
14. Minton, S., Carbonell, J. G., Knoblock, C., Kuokka, D. and Nordin, H., "Improving the Effectiveness of Explanation-Based Learning," Tech. report, Carnegie-Mellon University, Computer Science Department, 1986.
15. Minton, S., "Selectively Generalizing Plans for Problem Solving," *Proceedings of IJCAI-85,* 1985, pp. 596-599.
16. Mitchell, T. M., *Version Spaces: An Approach to Concept Learning,* PhD dissertation, Stanford University, December 1978.
17. Mitchell, T. M., Utgoff, P. E. and Banerji, R. B., "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics," in *Machine Learning, An Artificial Intelligence Approach,* R. S. Michalski, J. G. Carbonell and T. M. Mitchell, eds., Tioga Press, Palo Alto, CA, 1983.
18. Rosenbloom, P. S. and Newell, A., "The chunking of goal hierarchies: A generalized model of practice," in *Machine Learning: An Artificial Intelligence Approach, Vol.2,* R. S. Michalski, J. G. Carbonell, and T. Mitchell, eds., Kaufmann, Los Altos, Calif., 1986.
19. Shell, P. and Carbonell, J. G., "The RuleKit Reference Manual", CMU Computer Science Department internal paper.
20. Winston, P., *Artificial Intelligence,* Reading, MA: Addison Wesley, 1977.