A TASK-INDEPENDENT EXPERIENCE-GATHERING SCHEME FOR A PROBLEM-SOLVER

J. R. Quinlan
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania U.S.A.

Abstract. A scheme for allowing a problem-solver to improve its performance with experience is outlined. A more complete definition of the scheme for a particular problem-solving program is given. Some results showing the effectiveness of the scheme are reported.

Key Words. Problem-solving, heuristic search, depth-first, classification of operators, ordering of operators.

Introduction

It is not at all clear how a problem-solver should Improve its performance as it solves, or fails to solve, the problems presented to it. It is clear, however, that such improvement is necessary if problem-solvers are ever to compete successfully with human beings. Moreover, the improvement cannot be restricted to rote learning of results; what is needed is some equivalent of a human being's ability to detect analogies between problems. This paper outlines one scheme that has been implemented and tested fairly extensively. Although details are given for a particular problem-solving system, the general philosophy is applicable to most depth-first problem-solvers.

A problem, in this context, is a triple consisting of a *state*, a *goal* and a set of *operators,* A state is some object: a well-formed formula, a pattern or somesuch. A goal describes some hypothetical state or set of states. Each operator maps some set of states into another set. A *solution* to a problem is a sequence of operators that maps the given state into another state that matches the goal. For example, a state could be an algebraic expression, a goal another algebraic expression, and the operators rules for changing expressions into equivalent expressions. A solution would then be a schoolboy-type demonstration that the state and goal expressions are equivalent. This definition of problems and solutions clearly includes heuristic search problems.1

Figure 1 shows a partial outline of a depth-first method for finding solutions to such problems. For any (sub)problem *(s,g,R)* a set *A* of operators is selected, one or more of which may hopefully lead to a solution. Each of the operators is tried until one leads to a solution or the set is exhausted. Many essential mechanisms have been omitted from this skeleton, principally how the set of operators is selected, what new problem is tried when an operator is chosen, and what happens when a subproblem is solved. These vary from system to system, and, while they are critical to the performance of any system, the experience-gathering scheme is independent of their form.

Given a problem (state a,goal *g,*operators *R)* proceed as follows:

1. If *8* matches the description *g,* report success on this problem. Otherwise, select a subset *A of R;* these are the operators to be tried.
2. If *A* is empty, report failure on this problem. Otherwise, choose an operator *x* in *A* and delete *x* from *A.*
3. Attempt a new problem based on the current problem and operator *x.* If failure is reported on this new problem return to step 2.

Figure 1. Skeleton of depth-first problem-solver.

Such a problem-solving schema could make use of 'experience* in a number of ways. The two obvious ones are to guide the selection of the set *A* of operators to try, and to choose which operator *x* In *A* to try next. The selection of the set *A,* however, is typically performed by some algorithm which is not sufficiently flexible to allow modification by experience.13 The scheme given here is based on the second alternative. Each operator *x* in *A* will be assigned a weight, determined from past experience, which represents an estimate of how likely it is that trying *x* will lead to a solution of the problem *(8yg,R).* When choosing an operator from *A,* the one with the highest weight will be tried first.

The first question is, where is this weight to come from? We would like to have weights associated with operators in the context of problems. In any interesting task environment, the operator and problem spaces are large, if not infinite; it is clearly impractical to associate a weight with every possible operator-problem pair. The way out adopted here is to categorize operator-problem pairs, and associate a weight with each class. The weight assigned above to *x* will then be the weight associated with the class into which the pair *(x, (S,g,R))* falls. Whenever a problem is solved, the table of weights associated with the classes will be adjusted in an attempt to correct any mistakes in the order operators were tried.

The critical factor here is the classification of the pair *(x, (8,g,R)).* One reasonable approach would be to look at how easy it would be to apply a: to e (i.e., to get the state into the domain of the operator), and how much the use of *x* would advance s towards *g.* Suppose we had a measure *Q(y,z)* of the 'similarity. of state *y* to goal 2, and that this measure took on one of *n* possible values. Then the approach would classify *(x,s,g,R))* by examining *Q(s, 'x should be applio-*

able') and *Q(x(s),g)**. In fact, the simplest such categorization is used; two operator-problem pairs fall into the same class iff each of the above similarities is the same for both pairs.

The next section is an amplification of the above for the Fortran Deductive System (FDS).3,4,5 For this system it is possible to develop a task-independent similarity measure, and thus a task-independent classification scheme.

A More Detailed Look

At this point it becomes necessary to discuss some details of FDS so that a more complete treatment of the experience-gathering scheme for this system can be given.

States are represented as trees with a symbol at each node, where a symbol is a binary or unary connective, free variable or constant. Each possible node position on such a tree is numbered as follows: the root of the tree is numbered 0, and the right and left successors of node *n* are numbered 2nc+l, *2n+2* respectively. Figure 2 shows two states with the number of each node in parentheses beside the symbol**.

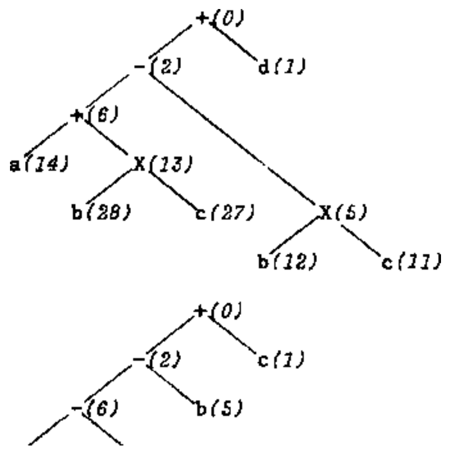


Figure 2. Two states.

Goals are represented as strings of *conditions,* each of which is of the form 'node *n* should be the symbol *q'* or 'the subtree whose root is node *n* should be identical to the subtree whose root is node m'. Each of the above is called a condition *on node n.*

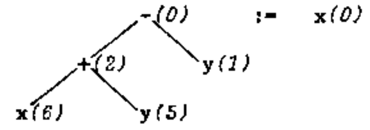
**8* may not be in the domain of *x*. In this case an estimate of *x(s)* is used.

**For the examples given in this section, the notation of elementary algebra will be used.

Rewriting rules take the form *yi=z,* where *y* and *z* are states. A rewriting rule informs the system that any instance of *y* can be mapped into the corresponding instance of *z.* Each rewriting rule defines an infinite number of operators of the form 'use rule number *m* to rewrite the subtree whose root is node w^1 for any non-negative *n.* The above operator will be written *0[m>n]* .

The domain of an operator *0[i,j]* can be described by a goal *G[i,j]* as illustrated in figure 3. First, a rewriting rule (number t, say) is

Rewriting rule number *i*



Goal *G*[*i*,2]

node 2 should be the symbol '-',
node 6 should be the symbol '+',
subtree whose root is node 13 should be identical to subtree whose root is node 5.

Operator *O*[*i*,2] rewrites the first state of figure 2 to the state

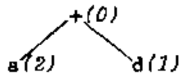


Figure 3. A rule, goal and mapping.

given, then the goal which is satisfied by a state iff that state lies in the domain of *0[i,2].* The first state *8* in figure 2 satisfies *G[i,2]* the state *0[i,2](s)* into which *0[i,2]* maps *8* is shown.

Let *8* be a state, *g* a goal. A condition (on a node *rri)* in *g* is *satisfied* by a if

i. *8* satisfies all conditions in *g* on nodes from which node *m* is descended, and

ii. *8* has the property required by the condition.

For example, consider the second state of figure 2 and the goal *G[i,2]* of figure 3. The first condition is satisfied, since there are no conditions on antecedents of node 2 and the symbol at node 2 is ^{f}-'. The second condition is not satisfied, for although *i* above is true, node 6 is not the symbol '+'. Finally, the third condition is not satisfied, even though the subtree whose root is node 13 is identical to the substate whose root is node 5. Requirement *i* above is not met, since the second condition is not satisfied and node 13 is a descendant of node 6.

The definition of states, goals and operators in this section is more restrictive than the corresponding concepts presented in the introduction; within this limited framework we can define a reasonable function *Q(e$_3$g)* to measure the similarity of state *s* to goal *g.* Actually, the word 'measure* is too strong; all that is needed for the purpose of classification is to know whether *Q(ab))* and *Q(o,d)* are equal, i.e., whether the two states are equally similar to their respective goals. Such an indicator is much easier to develop than a true measure.

We could say that *Q(a,b)* **and** *Q(c,d)* are equal if

i. goals *b* and *d* have the same number of conditions, and

ii, the number of conditions of *b* satisfied by *a* is the same as the number of conditions of *d* satisfied by *c.*

This does not fit the description of similarity given in the introduction, since there is an infinite number of possible values of *Q(s,g)*. Due to the way FDS is set up, however, goals with more than five conditions are rare. Taking account of this, requirement *i* is relaxed to allow goals with similar numbers of conditions to be lumped together, and *ii* is changed to specify roughly equal proportions of conditions satisfied. Let *x* be the number of conditions in *g, n* the number of these satisfied by s, and *y* the ratio *n/x*. Values 1 through 20 are assigned to *Q(s,g)* as shown in table 1. Note that these values are to be used

x	*y*	*Q(s,g)*	*x*	*y*	*Q(s,g)*
0	0	1	4,5	0	11
1	0	2	4,5	(0,1/4]	12
1	1	3	4,5	(1/4,1/2]	13
2	0	4	4,5	(1/2,3/4]	14
2	1/2	5	4,5	1	15
2	1	6	>5	0	16
3	0	7	>5	(0,1/4]	17
3	1/3	8	>5	(1/4,1/2]	18
3	2/3	9	>5	(1/2,3/4]	19
3	1	10	>5	1	20

Table 1. Values of *Q(s,g)*

only to establish equal similarity; if *Q(a,b)* is greater than *Q(o,d)* it does not follow that *a* is more similar to *b* than *c* is to *d.*

Suppose now that the 20X20 classes of operator-problem pairs are numbered 1 through 400. Two pairs are to fall in the same class iff their states are equally similar to the goals of applying the operators, and the resulting states are equally similar to the problem goals. Putting this another way, the operator-problem pair *(0[i,j]j (8,g,B)J* falls in class

$$20\text{X}(Q(s,G[i,j])-1)+Q(O[i,j](s),g)$$

where, if *e* does not satisfy *G[i,j], O[i,j](s)* is a synthetic state which looks like the result of using o[t,j] to rewrite some state. If *0[i,J]* were selected to solve the problem *(s,g,R),* the weight assigned to *0[ijj)* would be the weight associated with the above class. FDS goes one step further; it uses this weight as a base for determining a final weight, but the discussion of this process lies beyond the scope of this paper.

The experience-gathering, then, consists of adjusting the table *T* of weights associated with the classes. When a problem is solved, each step of the solution is examined to see whether any operators were tried before the 'correct' one (the one in the solution). If this is the case, the weights associated with the classes of the incorrect operators are decreased, while the weight associated with the class of the correct operator is increased. The adjustment formulae appear in 5.

Two comments ought to be made here. First, the classification rule uses only the similarity measure defined above. This, in turn, is defined in terms of properties shared by all states and goals in FDS. Thus the classification mechanism is task-independent, in the sense that it is defined for any problem which can be presented to FDS. Since FDS is a general-purpose system, this feature was a critical factor in the design of the scheme.

The second comment is really a query: given that the classification is defined for an arbitrary problem, is it appropriate? Saying that two operators in the context of a problem fall in the same class is asserting that, in this scheme, they are equally likely to lead to a solution. This is a strong statement. If the classification has no connection with reality, adjusting weights associated with classes is unlikely to produce anything except random behavior. On the other hand, if significant improvement results then the classification scheme may represent a useful way to categorize operators in the context of problems. The question of the appropriateness of the scheme is best answered, then, by examining some results of its use.

Results

The performance of FDS in four task environments will be summarized. Each of these consists of an ordered set or *block* of problems to be solved in sequence. In about half the cases the problems are theorems to be proved. A brief description of the blocks is given below; a complete definition appears in 5,6, Some of the problems have been presented to human beings, who find them non-trivial.3

Block *5* contains fifteen problems in an algebra pertaining to flowchart equivalence developed by Sanderson.^ (The problems are taken from this thesis.) Initially there are twenty-six rules, but, as each theorem is proved, it is retained as a new rewriting rule. Figure 4 shows the given rules, the theorems, and a sample proof of the first theorem. For convenience, states are represented in conventional bracketed notation rather than as trees.

Block *A* consists of eighteen theorems of

Notation: 'S' is a unary connective, 'I' and 'Z' are constants, and all other alphabetic characters represent free variables. The binary connectives '+' and '/' have no relation to the symbols used in arithmetic.

Initial rewriting rules given

1. (a+b)+c:=a+(b+c)	14. (a/b)+c:=(a+c)/(b+c)
2. a+(b+c):=(a+b)+c	15. (a/b)/c:=a/c
3. I+a:=a	16. a/c:=(a/b)/c
4. a:=I+a	17. a/(b/c):=a/c
5. a+I:=a	18. a/c:=a/(b/c)
6. a:=a+I	19. S(a):=(a+S(a))/I
7. Z+a:=Z	20. (a+S(a))/I:=S(a)
8. a+Z:=Z	21. S(a/b):=S(a)
9. Z:=a+Z	22. S(a):=S(a/b)
10. Z:=Z+a	23. S(a)+b/c:=S(a)+c
11. I/I:=I	24. S(a)+c:=S(a)+b/c
12. I:=I/I	25. S(I):=Z/I
13. (a+c)/(b+c):=(a/b)+c	26. Z/I:=S(I)

Theorems

1. a/a:=a	9. S(a)/I:=S(a)
2. a:=a/a	10. S(a):=S(a)/I
3. ((a/b)+c)/d:=(a+c)/d	11. S(a)+S(b):=S(a)
4. a/((b/c)+d):=a/(c+d)	12. S(a):=S(a)+S(b)
5. (a+b)/c:=((a/d)+b)/c	13. S(a)/S(b):=S(a)
6. a/(b+c):=a/((d/b)+c)	14. S(a):=S(a)/S(b)
7. S(Z):=Z/I	15. S(S(a)):=S(a)
8. Z/I:=S(Z)	

Proof of first theorem

a/a	
:=a/(I+a)	by *O*[4,1]
:=(I+a)/(I+a)	by *O*[4,2]
:=(I/I)+a	by *O*[13,0]
:=I+a	by *O*[11,2]
:=a	by *O*[3,0]

Figure 4. Block *S* and a sample solution

elementary algebra concerned with the manipulation of binary addition and subtraction. There are six rules given and, as before, each theorem proved is added to the list of rules.

Block *H* is composed of twenty-five problems in lexical pattern recognition in a generalized form of that found in Ledley^. There are twenty rules defined throughout.

Block P is a statement of a well-known puzzle. A philosopher is walking in a land peopled exclusively by Goodies (who only tell the truth) and Baddies (who always say the exact opposite of the truth). Coming up to two of the residents of this land, our philosopher asks the way to the library. One mutters something unintelligible, and the other says, "He says east. He's a Baddie." Which way should the philosopher go? In the formulation used there are eight rewriting rules.

For an individual problem, the performance of FDS will be measured by *efficiency,* the ratio of the number of states in the solution to the number generated while searching for it. If a problem is solved with 100% efficiency, then there is not much wrong with the ordering of operators for trial! For a block, the performance measure will be the *average efficiency* on the problems in that block.

The first set of experiments is designed to demonstrate that the scheme allows FDS to improve its performance on a given block of problems. Successive *passes* are made through the block, as follows. Each weight is initialized at 0.5 and the problems of the block solved in sequence with the adjustment of weights suppressed; this is called *pass 0,* (The average efficiency on pass 0 represents the performance of FDS with no re-ordering of selected operators.) Any problems retained as rules are then forgotten, and *pass 1* is made permitting the adjustment of weights. Any rules kept are again discarded, but the adjusted table of weights is retained, and *pass 2* is made. The above is repeated until *pass 10* is finished. Since nothing changes between passes except the table of weights, any improvement must be due to the 'experience' represented by this table.

The results of this set of experiments are summarized in table 2. Notice that, even with no

pass	block *S*	block *A*	block *H*	block *P*
0	23	28	50	18
1	26	51	71	18
2	50	68	90	77
3	67	70	88	42
4	53	75	90	38
5	57	75	91	42
6	55	88	88	42
7	56	88	88	91
8	65	88	91	91
9	67	88	88	91
10	67	88	87	91

Table 2. Average efficiency (%) on passes.

experience, the average efficiency on each block is quite high. FDS incorporates a powerful algorithm for selecting operators and screening out useless ones, and the order in which it discovers operators gives some clue to the order in which they ought to be tried. The ordering, and hence the efficiency, still improves significantly on the series of passes; on pass 10, sixteen of the eighteen problems of block *A* are solved with 100% efficiency. Although only four tasks have been explored to this length, comparable improvements have been noted in all the dozen or so tasks presented to the system.

But the concept of experience is stronger than this. Solving problems in some area should allow the system to solve new problems better. In other words, experience should be transferable from one set of problems to another set in the same task environment. Generally, this turns out to be the case with this scheme. Consider, for example, the last five problems of each of the blocks S, *A* and *H;* call them blocks 5', *A'* and *H'.* We will compare the average efficiency with which FDS solves each of these subblocks using two different tables of weights: *E,* the table of pass 0, and *E',* that obtained during pass 1 through the block after all but these problems have been solved. Note that these tables of weights contain no experience from any of the problems in the subblocks. Table 3 shows that, for blocks *A* and #,

experience	block *S'*	block *A'*	block *H'*
E	0.7%	27.0%	50.1%
E'	0.8%	64.8%	79.5%

Table 3. Within-task transferability

solving all but the last five problems of each significantly helps the solution of these last five. For block *S* the improvement is very slight. The last five problems of block S_9 however, are different from the preceeding ones (they all have two S's). In this case, then, a notable improvement was not to be expected. As a further test, a relatively difficult algebra problem was presented to FDS. With no experience, the system (on an IBM 7040-7094 DCS) was unable to solve it in an hour. On the other hand, when the experience from block *A* was given to FDS, it found a solution in five and one half minutes.

Some incomplete experiments have been made to test whether this form of experience is transferable, not from one set of problems to another within the same task, but from task to task. It has been found that this is sometimes useful, but that in most cases the improvement is insignificant. One interesting point which has emerged from these experiments is that it is possible to combine tables obtained from several tasks; the resulting synthetic table leads to better performance averaged across all the tasks than any of the individual tables from which it was formed.

Conclusion

On the basis of the experiments performed with the scheme, it seems fair to say that it works. This is encouraging when one considers the simple-mindedness of the approach. While it could only benefit from a more powerful measure of similarity, the basic idea of classifying operators in the context of problems seems to be appropriate.

References

1. Ernst, G. and Newell, A. Generality and GPS. Technical report, Carnegie Institute of Technology, January 1967.

2. Ledley, R. *Programming and Utilizing Digital Computers.* McGraw-Hill, New York, 1962.

3. Quinlan, J. R. and Hunt, E. B. A formal deductive problem-solving system. J. *ACM 15,* 4 (October 1968), pp. 625-646.

4. ——— and ———. The Fortran Deductive System. *Behavioral Science*, January 1969.

5. Quinlan, J. R. An experience-gathering problem-solving system. Technical report (Ph.D. thesis), Computer Science Group, Univ. of Washington, May 1968.

6. ———. Fortran Deductive System: experiments with two implementations. Technical report, Computer Science Group, Univ. of Washington, May 1968.

7. Sanderson, J. Theory of programming languages. Ph.D. thesis, Univ. of Adelaide, Australia, 1966.