A CONSTRAINED MECHANISM FOR PROCEDURAL LEARNING

Stellan Ohlsson
Uppsala Programming Methodology and Artificial Intelligence Laboratory
Computing Science Department, Uopsala University
Box 2059, S-750 02 Uppsala
Sweden

ABSTRACT

The problem studied is that of learning problem solving heuristics by doing. The purpose is to explore the learning behavior of a highly constrained mechanism. The constraints are choosen on the basis of Dsychological considerations. Computer runs show that the constraints do not prevent successful learning. The learning style of the program turns out to be a function of the structure of the problem space.

I HEURISTICS LEARNING

The task studied in this paper is that of discovering problem solving heuristics by doing. We presuppose a task-independent, weak problem solver which can take a problem space and a problem as inout, and search for the solution to the latter. We add a set of learning mechanisms which specify how the information generated during search should be encoded for future use. The task of the system is to construct a strategy for searching a problem space on the basis of repeated problem solving trials in that space. Systems of this kind have been described by Anzai & Simon (1979), by Langlev (198?), by Ohlsson (1982; 1983), as well as by others. They will here be called heuristics learners.

The purpose of the work reported here is not to construct the most intelligent learning system which the state of the art allows, but to study how a particular set of constraints on an information processing system affects its learning behavior. The constraints chosen for study are based on psychological considerations, although no detailed comparison between the Drogram and human behavior will be made in this paper.

II THE UNIVERSAL PUZZLE LEARNER

An earlier version of the Universal Puzzle Learner (UPL) program has been described in Ohlsson (1982; 1983). The third and current version consists of three layers: the implementation language, the problem solver, and the learning mechanisms.

A. Implementation Language

UPL is written in PSS, a neo-classical production system language (Ohlsson, 1979). It allows the user to organize production memory into nodes, each node containing one or more productions. When PSS is "in" a node, it will try to fire a production from that node before it consides productions from other nodes. Control can be transferred from one node to another by the firing of a production.

The PSS condition matcher recognizes a construct called sequence variables, which fulfill the same function as the "three dots" of informal mathematics; the expression "(A <SEQ> B)" in which <SE0> is a sequence variable is interpreted as "any list containing A, followed by any number of expressions, followed by B". Sequence variables match against the empty sequence, so that "(A B)" is an Instance of "(A <SEQ> B)". As an example of the use of sequence variables, the expression "(<SE0> X1)" will bind the variable X1 to the last expression in any list which has at least one element. As a second example, the expression "(ALTERNATIVES: X1 <SEQ>)" will recognize a list with one or more alternatives. The sequence variables are bound to the sequences they match against. The PSS sequence variables can be seen as a development of the "remaining segment" feature ("!") of the OPS language family (Forgy & McDermott, 1977).

B. The Problem Solver

The next layer of UPL is a task-independent, weak problem solver consisting of 13 nodes with various tasks, such as setting goals, maintaining the goal-stack, generating actions, censoring actions, doing conflict resolution, executing actions, evaluating results of actions, backing up, and restarting. The program takes a problem space and a problem as input. A problem space is given to the program in two parts, a BNF grammar defining a knowledge-state, and a list of operator definitions. A problem is presented as an ordered pair of states, the initial state and the goal state. The problem solver is task-independent in the sense that it does not make any assumptions about the expressions which are used as the knowledge-elements of the problem space.

The problem solver works as follows. The current goal is analyzed by subgoaling rules. Actions to be taken are generated either by forward-searching proposers, by a rudimentary form of means-ends analysis, or bv random generation. When one or more actions have been generated, they are subject to scrutiny by censors which may or may not reject some of them. (Notice that the censors apply before the action is taken.) If more than one action survive the censor, conflict resolution is applied. Execution of the selected action is done by an interpretative procedure which takes an operator definition as input. An operator definition is a schema-like structure with slots for the name of the operator, central parameters, auxiliary parameters, background, inputs, outputs, and side-effects. The outcome is evaluated, and the program then either restarts from the initial state, backs up to the immediately preceeding state, or returns to the goal-handling node.

The problem solver is constrained in three ways.

1. Context independence—The rules by which the system sets subgoals, generates actions, and censors actions are restricted to perform tests on the current knowledge-state only. This constraint implies that the choice of action in the current knowledge-state cannot depend on the history of the problem solving attempt. Looking forward in time, it implies that the problem solver cannot plan in the sense of deciding upon a sequence of actions. In short, action selection is localized to the current knowledge state.

2. Fragmentary path memory—The program has four different knowledge-states available: the initial state, the state in which the current goal was set, pushed, or popped, the current state, and the immediately proceeding state. Immediately after action execution, five states are available, because what was formerly the "immediately preceeding state" is not deleted until after evaluation of the new state. Thus, the evaluation node has a slightly wider view of the solution path than the other nodes.

3. Incomplete search control—Since deleted states are lost the program cannot backup to any arbitrary state. Search is organized as follows: Immediately upon entering a new state, that state is evaluated statically. If the outcome is negative, a backup is made to the immediately preceeding state. If the outcome is positive, an effort is made to push ahead: if dynamic evaluation reveals that the state is, after all, bad, then one of the bad actions is taken anyway. If dynamic evaluation shows that there are no "legal" actions in the current state, the program restarts from the initial state.

These constraints are studied because they exemplify the kind of constraints which can be caused by psychologically relevant memory limitations, eg limited capacity working memory.

C. Learning Mechanisms

The learning strategy of the program is to notice situations in which some production rule should have fired, but did not. For example, an action with a good outcome should have been suggested by some proposer rule, because it is the task of those rules to know about good actions. If it were, in fact, generated by task-independent methods, then some change in the set of proposers is called for. The program then assembles a production instantiation which corresponds to that instantiation which would have fired, had there been any proposer capable of firing. Next, it tries to generalize existing proposers in such a way that one of them acquires the ability to generate that instantiation. If no existing proposer can be generalized to cover the instantiation, a situation-specific production is created and added to production memory (to constitute raw material for future generalization attempts). Similarly for the creation of the other types of rules.

Revision of existing productions is done through a generalization algorithm. The algorithm is written as a Lisp program, and constitutes about half the code of the entire UPL program. It takes any two Lisp S-expressions as input and generalizes the first of them in such a wav that it would match against the second, if the two of them were given as inputs to the PSS pattern matcher. The complexity of the algorithm depends to a large extent on its ability to handle PSS sequence variables (see above). For example, given the three expressions

```
(EP (P => Q) P = Q),
(EP (A r> B) A * B), and
(EP (C1 C? => B) C1 C2 • B),
```

where all atoms are interpreted as constants, the algorithm responds with the expression

```
(EP (X1 <SEQ>) X1 <SEQ> * X2),
```

where X1 and X2 are ordinary variables and <SEQ> is a sequence variable.

The algorithm delivers some output for any pair of inputs; if the two S-expressions given to it have nothing in common, it will report a single variable. The algorithm computes an interpretable measure of how much its first argument had to be changed in order to "cover" its seoond argument. This measure is used to direct search through the space of possible generalizations. If the algorithm finds a generalization, the corresponding production is revised. There is only one generalization algorithm; thus, proposers, censors, and subgoaling rules are revised in a uniform way.

Revision of productions is attempted under the following conditions. If an action has a good outcome, try to revise the proposers so that they can produce that action in the future. The criterion for a good outcome is that the current knowledge-state has a higher overlap with the active goal than the preceeding state, ie that it has more knowledge-elements in common. (The program can use a task-specific evaluation function, if the user cares to specify one.) The ease for censor rules is analoguous: If an action has a negative outcome, try to revise the censor rules. If a state is interesting, without being either good or bad, try to revise the subgoaling rules so that they will set up that state as subgoal. In short, the learning behavior of the program is governed by its conception of good, bad, and interesting knowledge-states. Fine tuning of the learning mechanisms proceeds mainly by adjusting the criteria for those three categories.

III RESULTS AND DISCISSION

The UPL3 program learns successfully in several puzzle domains, including the Missionaries and Cannibals problem, the Tower of Hanoi puzzle, and the Tiles and Squares task. It solves the first-mentioned problem without search or unnecessary steps on its fifth pass over the problem. The simpler Tiles and Squares problem is solved through the miminal solution path already on the first trial.

Interestingly, the "learning style" of the program varies with the task to which it is applied. In the Tiles and Squares Puzzle, the program learns mainly through the acquisition of proposers. It creates rules of the form "If it is part of the goal to place object X in position Y, and position Y is empty, then move X to Y". The censors and the subgoaling rules created are unimportant for subsequent problem solving. Indeed, the two acquired subgoaling rules are never applied. In contrast, in the Tower *of* Hanoi task, UPL3 improves mainly by learning to decompose the top goal into the appropriate sequence of subgoals, ie it discovers that getting discs 1, 2, and 3 onto some peg X implies getting 2 and 3 onto X, which in turn implies getting 3 onto X. The difference between the two tasks which is responsible for the difference in learning behavior is that more work is necessary before part of the top goal becomes satisfied in the Tower of Hanoi problem than in the simpler problem.

On the Missionairies and Cannibals problem, on the other hand, UPL3 learns mainly by the acquisition of censors. The only general proposer it can find for this task is "When there are two cannibals and a boat on one side, and a single cannibal on the other, then let the two cannibals row across", which, as it happens, correctly generates two of the steps on the solution path. The difference between the problems which causes this difference in learning behavior is simply that there are fewer regularities to be detected in the problem space for this problem than in the very regular spaces associated with the other two tasks.

There are two basic phenomena of human learning which UPL3 seems incapable of mimicking.

1. Exponentially decreasing learning curves—It is a basic feature of human learning that change is most rapid during the initial encounter with a new task. An earlier version of UPL had relatively flat learning curves, and so does UPL3. We conjecture that "empirical" or "inductive" learning systems in general will have difficulties in mimicking this property of human learning. There is a certain tension between rapid initial change and learning through generalization over a sequence of successively encountered items, or instances.

2. Automatization—By automatization is meant that the amount of computational effort (eg number of production system cycles) to reach solution on a problem decreases in the abscence of improvements in the solution. The same sequence of "moves", steps, inferences, etc., is generated, but with less work. This is a central feature of human learning. However, it does not appear in the behavior of UPL3, because the number of production system cycles needed to make a move does not decrease as new heuristics are acquired. Automatization requires a move-chunking or composition mechanism.

REFERENCES

Anzai, Y. & Simon, H. A. The theory of learning by doing. Psychological Review, 124-140, 86, 1979.

Forgy, C. & McDermott, J. The OPS2 Reference Manual. Department of Computer Science, Carnegie-Mellon University, 1977.

Langley, P. Strategy acquisition governed by experimentation. Proceedings of the European Conference on Artificial Intelligence, Paris, 1982.

Ohlsson, S. PSS3 Reference Manual. Working Papers from the Cognitive Seminar. Department of Psychology, University of Stockholm. No. 4, 1979.

Ohlsson, S. On the automated learning of problem solving rules. In R. Trapp (Ed.) Cybernetics and systems research. Vol 8. Amsterdam: North-Holland, 1983.

Ohlsson, S. Transfer of training in procedural learning: A matter of conjectures and refutations. UPMAIL Technical Report, No. 13, 1982.