LEARNING META-RULE CONTROL OF PRODUCTION SYSTEMS FROM EXECUTION TRACES

Malcolm C. Harrison
Courant Institute, NYU

and

Salvatore J. Stolfo
Columbia University

CUCS-10-80

LEARNING META-RULE CONTROL OF PRODUCTION SYSTEMS FROM EXECUTION TRACES*

Malcolm C. Harrison
Courant Institute, NYU

and

Salvatore J. Stolfo
Columbia University

Summary

In the last decade, work in Artificial Intelligence has stressed the importance of having both declarative and procedural forms of knowledge available. Recently, a number of workers have pointed out that procedural information can be regarded as control information which can be used to control the sequencing of application of the declarative components, and have remarked on the advantages of keeping these two types of information separate [see, for example, Sigart Newsletters #63, June 1977 and #70, Feb. 1980]. In the last few years we have been looking at the problem of automatically inferring this control component from the behaviour of the declarative component [7]. In this paper we describe briefly some experiments we have done, discuss some of the difficulties we have encountered, and show how they might be overcome.

Introduction

Our previous work [7,8] has shown that it is possible to synthesize procedures (sequencing information) by analyzing successful execution traces provided by a trainer of a nondeterministic production system (PS) program [3]. Our approach was as follows:

(1) Select a 'typical' input to the program and run the program repeatedly on this input.

(2) Record the sequence of rules selected together with input/output information and the set of rules which could have been fired (called the conflict set of rules).

(3) Repeat this for other typical inputs.

(4) Describe the better (i.e. shorter) successful sequences in a control language, CRAPS, designed for this purpose (and described below).

(5) Generate a set of meta-rules whose objective is to aid the CRAPS description if the sequencing is inappropriate.

(6) Use the CRAPS description and the meta-rules to guide the program's subsequent decisions.

The CRAPS language provides a semantic framework with which to specify or describe sequences of rule applications in the execution of the PS program. The basic primitive of CRAPS is called a *unit*. A unit specifies either a rule application (with preconditions), in which case it is called a *simple unit*, or a control operation applied to a sequence of units. The control operations are Permutation of a set of sequences, Alternative or conditional selection of a sequence from a set of sequences, Repetition of a sequence controlled by simple Boolean assertions (described below) in Disjunctive Normal Form (DNF) and (implicit) Concatenation of units producing sequences. (From which we have derived the acronym CRAPS.).

The control primitives are represented syntactically in Cambridge form by PERMUTE, IF-THEN-ELSEIF, and REPEAT, respectively, while concatenation is represented by a list of units enclosed in double pointed brackets (< < >>) .

The CRAPS operators correspond to various control primitives of conventional programming languages, and to that of Regular Expressions (where permutation corresponds to a *shuffle* operator). The choice of using Regular Expressions for control depends on several considerations. First, people generally use descriptions of their own actions which appear very much like Regular Expressions. Secondly, since they are

one of the simplest formalisms, it would appear that they would be easier to induce from examples than other more complicated formalisms. Lastly, they are easily implemented and easy to understand.

However, it would appear that Regular Expressions are too limited in their expressive power to be of much interest. However, coupled with a powerful PS program as we use here, the total system is at least as powerful as the PS representation and is capable of a wide range of behavior with the additional control constraints. For example, consider the following example taken from Georgeff [4] (interpreting this production system in the usual formal grammar sense).

P1: S → ABC

P2: A → aA

P3: B → bB

P4: C → cC

P5: A → a

P6: B → b

P7: C → c.

Beginning with the initial sentential form (WM) containing S, these productions generate the language $\{a^i b^j c^k, i,j,k \geq 1\}$ which is context-free. If we restrict the permissible sequence of rule applications to be a member of the language generated by the following Regular Expression:

(1) p1 (p2 p3 p4)* p5 p6 p7, then the language generated is $\{a^n b^n c^n \mid n \geq 1\}$ which is context-sensitive. Notice that

p2 p3 p4 can be used in any order as can p5 p6 p7. We can describe these additional control constraints in CRAPS as:

(2) <<p1 [REPEAT [PERMUTE <<p2>> <<p3>> <<p4>>]]
[PERMUTE <<p5>> <<p6>> <<p7>>] >>.

Georgeff describes the use of Regular Expression control in this fashion to both limit the number of productions to be tested on each cycle and to leave nondeterministic selection points in tact and at well specified points within the control. For example, in (1) above, at the end of the repetition, only p2 and p3 can enter the conflict set of rules (severely limiting the number of productions to be tested, which can obviously be useful for large systems, but also be too restrictive in general) and leaving the final decision as to which production to select up to the conflict resolution strategy (or meta-level knowledge base, see [2]).

There is some attempt in CRAPS to lessen the responsibility of the conflict resolution strategies built in to the PS interpreter by allowing explicit specification of conditions under which repetitions should be allowed and alternatives should be selected. The repetition operator in CRAPS, therefore, contains both a *While* and *Until* clause and the alternation operator contains conditional expressions for each alternative very much like the LISP COND. Further, even the simple unit, which specifies the next rule to apply, contains a precondition for that rule to be applied. In total, this wealth of specified conditions is intended to move many of the nondeterministic decisions out of the PS interpreter and into the control

mechanism *explicitly*. The human expert who defined the PS and trained the system was allowed to view only the conflict set of rules during training, which proved to be an adequate model of the state of the problem-solving system. (Dynamic additions to the production memory were possible.) Therefore, the exact form of the conditions we used in the original language were (disjunctive) sets of rules whose left-hand sides matched the current contents of the data base. (The latest version of the language permits additional information on how data is shared between the rules in the conflict set.)

There are four types of meta-rules which assist a CRAPS description in controlling a PS. It is the simple unit which actually selects the next rule to fire on each cycle (the higher level control units produce sequences of simple units), and if in the event that the DNF expression evaluates to false or the specified rule is not active, the meta-rules are called upon to suggest a list of rule names to try in order to force the DNF expression to evaluate to true. (We use the control in an irrevocable fashion, not wanting to resort to backtracking.) For example, suppose that the simple unit (A (B C)) is in control, E was the previously fired production, and the current conflict set of rules is {B D}. This situation may be described as:

(1) A and C should be active

(2) D should (perhaps) be inactive

(3) {B D} is currently active

(4) E was just fired.

Accordingly, the meta-rules which we have developed are designed to deal with the four cases listed using the primitive functions *Want-active*, *Want-inactive*, *Currently-active*, and *Just-fired* respectively. In each case, a meta-rule may suggest a list of rules to try in that situation using the primitive function *Try-to-fire*. The suggestions are weighted since a rule may be suggested several times by different meta-rules.

In spite of the apparent sparsity of information, we were able to construct very useful and interesting procedures for solving our first experimental problem a slightly idealized jigsaw puzzle. In fact, our program synthesized, with one or two minor errors, all the strategies used by the trainer (separating pieces into piles, building the outer edge first, etc.). Our program also constructed a set of meta-rules, (see for example [2]) which in most cases had the ability to correct the behavior of the jigsaw puzzle program when the CRAPS description controlling it was not adequate.

Since the technical problem with our approach is equivalent to the problem of Inductive Inference of the Minimum Regular Expression from incomplete samples, which was proven by Angluin [1] to be NP-complete, the procedures we developed were necessarily heuristic in nature.

We encountered a number of other difficulties with this approach. The first difficulty is the (lack of) power of the CRAPS description, which corresponds to an extended form of finite state control. Thus, for example, a tree-traversal program with productions:

start

go-left, can't go left

go-right, can't go right

go-up

print

stop

and a finite-state control language cannot be used to implement an inorder scan procedure. To do this would require a context-free control language, permitting definition of self-embedding sequences (which correspond to

recursive procedures). See the section below for the details of this example.

A second difficulty is the lack of flexibility in the CRAPS description, which in essence specifies a set of acceptable execution sequences. Thus when the description is not applicable (e.g. the recommended production is not fireable) no heuristic information is available. This suggests that the rigid forms of control investigated by Georgeff [4] may not be adequate for real-world problems. We attempted to make up for this in the set of meta-rules, which in effect contained local information about fragments of the original sequences (e.g. if you want to fire x, try to fire y). The meta-rules are of course more flexible, but do not contain as much specific information as the description; an experiment using the meta-rules alone failed to solve the problem.

A third defficiency of our approach was that the explicit contents of the working memory was completely ignored.

In the following section we describe the characteristics of a new control language, tentatively called MCL, which we believe will contribute to a solution of these problems.

The MCL Control Language

The MCL control language is designed within the following constraints:

a. A MCL description should consist of a set of meta-rules, rather than a description of a set of sequences.

b. These meta-rules should be probablistic rather than deterministic, based on the closeness of the matching of their conditions to the current situation.

c. The meta-rules should make use of a goal-subgoal structure.

d. The meta-rules should be able to refer to the current state of the working memory.

Thus a meta-rule in MCL will be of the form

> if the working memory is similar to M
> and the current goal structure is similar
> to G, then the correct action is similar
> to A.

A may specify a firing of a production, or modifying the goal structure.

The goal structure will be a tree, with each node being an OR node (its goal might be achieved by first achieving the goal of any of its child nodes), an SEQ node (its goal might be achieved by first achieving the goals of its child nodes in the sequence specified), an IND node (its goal might be achieved by first achieving the goals of its child nodes, in any order), or a REP node (its goal might be achieved by first repeatedly achieving the goal of its child node).

The working memory will be described by giving the set of productions which are fireable, together with any necessary instantiation information. Previously this information just specified the set of productions; the new scheme will show how the various possible instantiations are related.

Notice the closeness of this meta-rule formalism to that of Davis [2]; the essential differences are the richness of the goal structure in our design, and the certainty of usefulness of a rule is dependent on its dynamic applicability rather than static specification.

Automatic inference of MCL meta-rules

In order to simplify the problem of analysing the execution traces, we will require that the trainer specify goal information for each action. This will be in one of the forms:

I can achieve the current goal by firing production P next.

To achieve the current goal I will first try the following (OR,SEQ,IND,REP) subgoal structure.

I have achieved the current goal.

This is no good, backtrack to goal G.

Goals may be specified either by arbitrary name, by specifying a production which is to be fired, or by specifying a desired working memory property. The only information available to the trainer will be the description of the working memory in terms of the productions which can fire and the current goal structure. If this is inadequate, the trainer will be permitted to introduce new productions.

Each action of the trainer thus provides a meta-rule; this meta-rule may have been used before, or may be new. The set of raw meta-rules will be used to construct the MCL description. To do this, the raw meta-rules will be refined as follows (M_i refers to a meta-rule):

if M_i is an instance of M_j, delete M_i;

if M_i has strictly stronger constraints than M_j, and recommends the same action, delete M_i;

In addition, we will attempt to generalize the set of meta-rules.

This set of generalizations will include [cf. 6]:

if M_i contains a condition which can be deleted without causing a conflict with some M_j, add a new meta-rule M'_i which is obtained from M_i by deleting the constraint.

if M_i and M_j only differ slightly, with $t_i \in M_i$, $t_j \in M_j$, $t_i \neq t_j$, add a meta-rule which says that t_i is (in context M_j) similar to t_j.

if M_i and M_j only differ slightly as above and t_i is similar to t_j, add a meta-rule with (t_i or t_j)replacing t_i in M_i.

if M_i and M_j are generalizations of some meta-rule M_k not in the set, add M_k.

As we note below, the effect of some of these operations can also be achieved by the MCL interpreter.

If the meta-rules are written as conjunctions of literals, the generalization procedure described by Vere [9] can be used to compute the common generalization M_k of two meta-rules M_i and M_j. Vere uses the notation $[\gamma]\ \alpha \rightarrow \beta$ to describe a production $\gamma \wedge \alpha \rightarrow \gamma \wedge \beta$, where α, β, and γ are conjunctions of literals. He observes that if $[\gamma_3]\ \alpha_3 \rightarrow \beta_3$ is a maximal common generalization of $[\gamma_1]\ \alpha_1 \rightarrow \beta_1$ and $[\gamma_2]\ \alpha_2 \rightarrow \beta_2$, then γ_3 may have less literals than γ_1 and γ_2, but $\alpha_3(\beta_3)$ must have the same number of literals as α_1 and α_2 (β_1 and β_2). In our case, meta-rules can be written in the form of conjunctions of literals of the following forms:

(GOALS futuregoals)

(OR goal subgoal)

(SEQ goal goalsequence)

(IND goal subgoal)

(REP goal subgoal)

(production instantiation)

(FIRE production instantiation)

The GOALS literal can describe the unexpanded part of the goal structure, while the OR, SEQ, IND and REP literals can describe the part already expanded. Information about the state of working memory can be given as a conjunction of (production instantiation) literals specifying which instantiations of which productions can fire; this generalizes the notion of *association chains* used by Vere to association graphs, since the instantiation information can specify substitutions which can be linked in an arbitrary way.

Interpretation of MCL descriptions

The meta-rules available to control the sequencing of the production system will be of the following types

$W \wedge G \rightarrow PA$, $W \wedge G \rightarrow GA \wedge PA$, $W \wedge G \rightarrow GA$

where W is a working memory description, G is a goal structure description, PA is a sequence of productions, and GA is a sequence of goal structure modifications. At each point that a production must be selected to fire from the conflict set, the MCL interpreter will consult these meta-rules to determine the appropriate action. This will be done as follows:

Each meta-rule will be matched against the current situation, using a partial matching algorithm (for example [5]). The action recommended by a meta-rule will be weighted according to the closeness of the match, with an exact match getting the highest weight, a match requiring substitution of variables somewhat lower weight, and a match in which both matcher and matchee are instances of the same expression lowest weight still. This will be done by looking for exact matches first, etc. In the case when no action is recommended with a weight

greater than some threshold, the interpreter will backtrack to a point at which two or more actions with similar weights were recommended.

An Example

Consider the following nondeterministic PS program which scans a binary tree. In the exact form of the PS representation we use, data elements can be any LISP data structure. An atomic data element in the LHS of a production must match an exact data element in WM and a list must match a list with the same structure and content. A symbol preceded with an equals sign represents a pattern variable which can match any data structure. The symbol ! is an operator which matches the entire remaining portion of the list that contains it, assigning the list value to the variable which follows it. Where it appears in the RHS of a production, it deposits the matching list into WM but without the enclosing parentheses. Data elements are deleted from WM only if they are included as arguments to the < delete > system function in the RHS of a production.

A binary tree is represented in WM by the following data format: The root of the tree is represented by (ROOT =x). If node B has a left son A, it is represented by (LEFT B A) and similarly (RIGHT B C) represents C as the right son of B. The father B of a node A is represented by (FATHER B A), and the current node scanned is (NODE = x). Production memory contains the following productions:

```
START
[(ROOT = X) - (NODE =X)   --> (NODE =X)]
GO-LEFT
[(NODE =X)   (LEFT =X =Y) -->  (< delete > (NODE =X)) (NODE =Y) (FATHER =X =Y
PRINT
[(NODE =X) - (ALREADY-PRINTED =X) -->  (<write> =X) (ALREADY-PRINTED =X)
GO-UP
[(NODE =X)   (FATHER =Y =X) --> (< delete > (NODE =X))
                                (NODE =Y)]
CAN-T-GO-LEFT
[NODE =X)  - (LEFT =X =Y) -->]
CAN-T-GO-RIGHT
[(NODE =X)  - (RIGHT =X =Y) -->]
STOP
[(NODE =X)   (ROOT  =X) --> (< halt >)]
```

Previously, we were able to infer a CRAPS description controlling this PS to deterministically scan in-order balanced binary trees. The CRAPS language is equivalent to finite state control and therefore lacks the power of a push-down automaton, making it impossible to construct a description to scan arbitrary binary trees with the existing PS. With the introduction of a goal structure, and allowing for recursive definitions of goals, the following (abbreviated) set of meta-rules achieves the desired deterministic control of the PS program:

```
M1  [(START = d1)    —>   (FIRE  START)]

M2  [(GOALS  INSCAN  !  =  rest)  —>
                                  (GOALS  (SEQ  INSCAN  (INSCAN-LEFT
                                                        (FIRE  PRINT)
                                                         INSCAN-RIGHT))
                                          !  = rest)]

M3  [(GOALS  (SEQ  INSCAN  (INSCAN-LEFT  ! = r))  ! = rest)
                                  —>   (GOALS  INSCAN-LEFT
                                               (SEQ  INSCAN  ! = r)
                                               ! = rest)]

M4  [(GOALS  INSCAN-LEFT  ! = rest)
     (GO-LEFT  = d1 = d2)    —>   (FIRE  GO-LEFT)
                                  (GOALS  INSCAN
                                          (FIRE  GO-UP)
                                          ! = rest)]

M5  [(GOALS  INSCAN-LEFT  ! = rest)
     (CAN-T-GO-LEFT = d1)    —>   (GOALS  ! = rest)]

M6  [(GOALS  (SEQ  INSCAN  (FIRE PRINT)  ! = r)  ! = rest)
                                  —>  (GOALS  (FIRE PRINT)
                                              (SEQ  INSCAN  ! = r)
                                              ! = rest)]

M7  [(GOALS  (FIRE PRINT)  ! = rest)
     (PRINT = d1)            —>   (FIRE PRINT)
                                  (GOALS  ! = rest)]
```

The remaining meta-rules specify how to satisfy the primitive goals of firing productions (as in meta-rule M7, if PRINT is active, and the goal is fire PRINT, then select PRINT from the conflict set), and the goal structure for the symmetric case of traversing the right son.

Further Applications

The application area that we have chosen for this work is graduate student advisement. This will be a question-answering program whose data-base is essentially the contents of the Computer Science Department bulletin. Questions will range from working out a suitable course schedule to requests for advice on possible careers - we anticipate being able to solve the former but think that we will have difficulty with the latter. The input will be in declarative form, with no control or sequencing heuristics at all. Our objective will be to infer, from sessions in which the program is guided by a trainer, this control information.

References

1. Angluin, D., On the Complexity of Minimum Inference of Regular Sets, unpublished manuscript, 1977.

2. Davis, R., Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases, Ph.D. thesis, Stanford U., 1976.

3. Davis, R., and King, J., An Overview of Production Systems, Stanford U., AI Lab Memo, AIM-271, 1975.

4. Georgeff, M.P., A Framework for Control in Production Systems, Proc. IJCAI6, Tokyo, 1979.

5. Hayes-Roth, F., and McDermott, J., Knowledge Acquisition from Structural Descriptions, Proc. IJCAI5, Cambridge, 1977.

6. Ditterich, T.G. and Michalski, R.S., Learning and Generalization or Characteristic Descriptions: Evaluation Criteria and Comparative Review of Selected Methods, Proc. IJCAI6, Tokyo, 1979.

7. Stolfo, S.J., and Harrison, M.C., Automatic Discovery of Heuristics for Non-deterministic Programs, Proc. IJCAI6, Tokyo, 1979.

8. Stolfo, S.J., Automatic Discovery of Heuristics for Non-deterministic Programs from Sample Execution Traces, Ph.D. thesis, Courant Institute, NYU, 1979.

9. Vere, S.A., Induction of Relational Productions in the Presence of Background Information, Proc. IJCAI 5, Cambridge, 1977.