POPS: AN APPLICATION OF HEURISTIC SEARCH METHODS TO THE PROCESSINC OF A NONDETERMINISTIC PROGRAMMING LANGUAGE

by

Gregory Dean Gibbons

Naval Postgraduate School
Monterey, California

Abstract

POPS ts a processor for a simple nondeterministic programming language, PSL. POPS accepts a problem stated in PSL and attempts to solve it by finding a successful execution of the PSL program, POPS operates by identifying elements of the input program with elements of the heuristic search paradigm, analyzing the input program to obtain information about the problem operators, and applying methods borrowed from GPS to solve the problem. In addition to the goal-directed methods based on GPS, POPS contains the methods developed by Fikes in his program REF-ARF.

Key words: Problem solving. Heuristic search, Nondetermlnistic programming language.

Background

Nondetermlnistic Algorithms

Robert Floyd (Floyd, 1967) suggested that a compiler for a nondetermlnistic programming language (NDPl.) could be used as a problem solver. His NDPL was obtained by adding to Algol the function CHOICE (N) and two special labels for exits, SUCCESS and FAILURE. The value of CHOICE(N) is an unspecified Integer between 1 and N. Only executions that terminate with a SUCCESS exit are considered to be computations of the algorithm. The programmer may impose constraints on the values of the program variables by inserting, for example, IF statements that direct the flow of control to a FAILURE exit unless the constraints are satisfied. In this way the programmer specifies what constitutes a solution to his problem.

The proposed problem solver would operate as a compiler by translating a nondetermlnistic program into a deterministic one and executing the resulting program. The deterministic program is so constructed as to simulate the input program by enumerating the possible combinations of values of the CHOICE function.

Floyd illustrated his proposal with a complete statement, translation, and solution of a sophisticated representation of the Eight Queens problem. This example shows clearly that programming a problem in his nondetermlnistic language is much easier than programming the corresponding search in a conventional language. However, it is also clear that the solution is obtained by a blind generate and test search. Thus, unsophisticated representations of problems having large search sp.aces would result In unacceptably long execution times.

Interpretation of Nondetermlnistic Programs

Nondetermlnistic programs are convenient for stating problems because they leave unspecified the order in which some alternatives are to be considered. Consequently, the process that interprets a nondeterministic program must determine for itself which alternative to consider at a given time.

A processor for a language containing a CHOICE function must select values for the function. If the language contains a nondeterministic branch instruction, or if the particular program's flow of control depends on the value of a CHOICE function, the processor must select possible control paths as well. A problem represented by a nondeterministic program is, therefore, a search problem In the space of possible combinations of values of the CHOICE function and possible control paths through the program.

Nondeterministic programs fall into two categories: those in which the flow of control is deterministic, and those in which it is not. In the former case, the processor may be required to select values for the program variables so that certain constraints are satisfied. If this selection is made symbolically, the processor can execute the program until and END statement ts encountered, at which time the symbolic selections must be replaced by actual values. Such problems are referred to as constraint satisfaction problems. Fikes's program REr-ARF solves a class of constraint satisfaction problems of considerable difficulty.

To execute a program having nondetermlnistic conflow, a processor must select the control path it will follow, as well as the values for the program variables. The problems represented by programs inthe second category are heuristic search (HS) problems.

REF-ARF

Richard Fikes (Ftkes, 1968, 1970) implemented a problem solving system, REF-ARF, based on Floyd's suggestion, but containing much improved" solution methods. Tikes represents nondeterministic values symbolically by creating internal variables. Constraints are represented as formulas involving these variables. This representation allows the use of algebraic simplification methods to reduce the size of the space to be searched. Fikes's most powerful solution method consists of alternating value instantiation and algebraic simplification, effectively reducing large spaces to anything from a few to a few hundred nodes. This method enables REF-ARF to solve some constraint satisfaction problems that would not only swamp a system designed along the

lines suggested by Floyd, but would prove very difficult for humans to solve as well.

P.EF-ARF will accept HS problems, but is essentially limited to a generate and test algorithm in its search for an executable path through the program. HS problems can induce REF-ARF to expend thousands of times more effort than is Intrinsically required by the problem. For example, tf the Missionaries and Cannibals problem Is represented by a program with a single loop representing a crossing of the river, and the loading of the boat is selected nondetermintstically, then each path through the program will represent a constraint satisfaction problem having two selections for each crossing of the river. Ref2 , an early version of POPS which used the same search mechanism as REF-ABF, was presented with such a program. Calculation showed that the solution would have required over *32* hours of 360/67 time.

POPS

POPS (Procedure Oriented Problem Solver) extends Fikes's work by including the successful methods of REF-ARF in a system designed to solve heuristic search problems. POPS was written in Lisp 1.5 and run on the 360/67 computer at the Naval Postgraduate School using the Waterloo Lisp Interpreter (Bolce, 1967).

The Problem Statement Language. PSL

PSL is a simple algebraic language, similar to Fikes's language, REF. PSL contains a nondeterministic choice function SELECT(A), whose value is some element of the named range A. The user may impose constraints on the values of the program variables by using the statement cONDTCION(B), which means that at the time the statement is executed the boolean expression B must be satisfied. PSL also contains a nondetermintstic branch Instruction, GOTOL (LI,1,2 ... Ln). GOTOL is essentially a computed GO TO with the index unspecified.

An Approach to Heuristic Search Problems: *GPS*

REF-APF's considerable success with constraint satisfaction problems is due largely to algebraic simplification methods which allow it to reduce its search space without eliminating valid solutions. Unfortunately, there is no similar way of reducing the space of possible execution paths when a program's flow of control is nondeterministic. The alternative is to provide some mechanism whereby the processor can make informed decisions in its search for a successful control path. The GPS work of Newell and others (Newell, Simon, and Shaw, 1963; Ernst and Newell, 1969) provides such a mechanism, namely, the selection of operators on the basis of their apparent usefulness In the current situation.

A problem statement for GPS contains a set of objects which are transformable by a given set of operators. A problem takes the form: given an Initial object X., find a sequence of operators q such that the predicate P is true of the object q(XJ. GPS attempts to transform the initial object into a desired object by matching the initial object to the desired object (or to the properties of a desired object) in order to obtain a difference; GPS then attempts to reduce the difference by applying an operator to the object. The operator is selected by means of a table of connections which indicates the relevance of the operators to the various differences.

The POPS Program

POPS solves heuristic search problems by applying GPS techniques to the interpretation of PSL. Lke GPS, POPS operates by transforming an initial object into a desired object by the application of operators.

The task environment of POPS. In the task environment of POPS, an object is a path from the beginning to the end of a PSL program. If the entry and exit points of the program are given labels, and tf every point in the program at which control paths join is labelled, then every execution path corresponds to a string of labels. The set of such strings of labels can be given by a grammar.

POPS obtains a suitable grammar by identifying all simple closed paths in the program and all simple paths from the beginning to the end of the program. For example, the program whose flowchart appears in Figure 1 has possible execution paths given by the following grammar:

S - BEGIN A EXIT

A - A MIfI | AM2 A | ...)A Mn A

Thus, solving a problem stated as a PSL program can be considered as searching in the space of strings of labels. The initial object is (BEGIN A EXIT), and each subsequent object results from applying one of the rules of the grammar for the program to a previously generated object.

The final object is a string of labels representing a path through the program such that If the path were executed, all the constraints would be satlsfiable. Clearly not all grammatically legal paths have this property. Thus, the search process must select the rules to apply at each point, and verify that at each point in the path the current constraints can be satisfied. Such a path will be referred to as a legal path.

Because applying a rule of the grammar to a string can result In an illegal path, the process of applying an operator in the original problem is represented by applying a rule of grammar and then executing the PSL program along the control path represented by the resulting string. If this execution results in the creation of an unsatlsfiable constraint, then the operator represented by rule does not apply to the object represented by the previous string of labels.

Description of operators. POPS uses a technique similar to that of GPS to make rational selections of operators. In place of a table of connections, POPS uses a description of the effects of the problem operators. To obtain these descriptions, POPS executes each operator in isolation, i.e., with no assumptions about the effects of any prior operators. By examining the assignments and the constraints so generated, POPS constructs a simple description of the operator's effects. Presently, this description consists of 1) a list of the program variables that are used before they are assigned, 2) a list of *program* variables that *are* changed, 3) the

set of constraints that must be satisfied before the operator may be applied, and 4) the size of the operator, as measured by the number of symbols in the data structure generated by the trial execution of the operator. While the current set of descriptors is minimal. it Is sufficient to guide POPS directly to the solutions of several simple heuristic search problems, as will be seen. More sophisticated analysis of operators would facilitate more intelligent selection of operators by the problem solver, and should enable it to solve more substantial problems.

Differences. Since any complete executable path is an acceptable final object, the match is accomplished by attempting to execute the path represented by the object. If a path is not executable, it is because at some point in the execution of the path, the processor encounters a constraint that cannot be satisfied. The processor then returns a triple (L, R, D), where L is the portion of the path that can be executed, R is the remainder of the path, and D is the unsatisfiable constraint, together with a description of what changes to the program variables must be made in order to satisfy the constraint. This description is the POPS version of the GPS difference.

Selection of operators. To select an operator, POPS compares the difference with the descriptions of the available operators, and selects the operator that appears most promising. Currently, POPS selects the operator that changes the most program variables needing change. In case of a tie, it selects the operator that changes the fewest variables having acceptable values. Finally, if there is sttll more than one candidate operator, POPS selects the smallest one.

Example. Consider, for example, the Msnkey problem , given in Figure 2. The grammar for the PSL program is:

S - BEGIN A GET_BANANAS

A - A WALK A (A CARRY A I A CLIMB A

The initial object is (BEGIN A GET_BANANAS). If this path is executed, a contradiction will be found after passing A, because the necessary conditions for exiting the program have not been met — the monkey doesn't have the bananas. The goal then becomes that of making the offending constraint TRUE. This requires changing the values of M(l), M(2) and B(l). Since WALK changes M{1), CARRY changes M(l) and B(l), and CLIMB changes M(2), it is reasonable to guess that rule A - A CARRY A should be applied. The proposed path then becomes:

(BEGIN A CARRY A GET_BANANAS).

When execution of this path is attempted, however, a contradiction is encountered at CARRY: the monkey cannot carry the box unless he is located where the box is, i.e., M(l) = B(l). The subgoal then becomes: make M(l) = B(l) true. Both WALK and CARRY change M(l), but CARRY does other things as well, so A - A WALK A is the move to try. The proposed path is then:

(BEGIN A WALK A CARRY A GET_BANANAS).

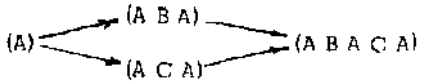
Execution of this path proceeds into GET_BANANAS before finding a false constraint. This one is: M(2)=box, representing that the monkey must be on the box. The correct move is A - A CLIMB A. The path becomes:

(BEGIN A WALK A CARRY A CLIMB A GET_BANANAS).

Since this path can be successfully executed, the solution has been found.

Redundancy Tests

POPS employs tests to prevent two types of redundancy in its search for a complete executable path. A newly generated path is redundant if it is either identical with or equivalent to a previously generated path. Equivalent paths can be generated, for instance, if there are two operators q_1 and q_2 such that cTJd^ $^{an(\wedge q}$29] achieve the same effect. Identical paths can be genated as follows:



(AC A)-^^

Here the two operators A -• A B A and A -• A C A have been used to expand (A) to generate (ABACA) in two different ways.

POPS tests newly generated paths for equivalence to previous paths by comparing the state of the program variables resulting from the execution of the new path with the results of executing previously generated paths. This comparison can be rather expensive if the variable values are formulas; however, the comparison is essential to limit search.

POPS automatically avoids the generation of identical paths by restricting the selection of operators according to a rule known as the O-size rule. Each operator Q Is assigned a unique size, denoted |QJ. The O-size rule provides that an operator Q may be considered for application to a path P = (L, R, G) only if |0 | < QR and \Q \ < QL, where QL is the size of the last operator used to generate L and OR is the size of the next operator in R. Intuitively, one may consider the size of an operator to be its difficulty. The Q-size rule then predicates that POPS will consider applying an operator to an object only if the operator is easier than its immediate supergoal and no more difficult than the previous goal.

The Importance of the Q-size rule is that it prevents the catastrophic proliferation of identical nodes in the program's search space. Without such a test, the number of redundant nodes can grow as fast as N! , where N is the number of operators applied to generate a single node. The redundancy could be eliminated by means of a direct comparison with previous nodes; indeed, this Is the usual practice. However, the number of comparisons required grows exponentially with N. In contrast, the cost of the Q-slze rule is only two comparisons per node generated. These assertions about the Q-size rule are proved elsewhere (Gibbons, 1972).

POPS flowcharts. The flowcharts for the major problem solving processes in POPS are given in Figure 3. POPS itself is an executive routine that analyzes the PSL program to obtain the grammar and operator descriptions, and then enumerates the simple paths through the PSL

program until one is found that can be converted into a legal path. The other programs, TRANSFORM, FIXPATH and APPLY are similar to the GPS methods TRANSFORM , REDUCE, and APPLY, respectively, *as given* in Newell and Simon (1963). In FIXPATH, the list of operators is filtered by the Q-size Pule before an operator is selected. The operation "assign If appropriate" selects actual values for nondeterministic expressions under certain circumstances. Issues surrounding this assignment will be discussed in the following section.

Performance of Pops

The performance of POPS can be easily evaluated by means of a comparison with the performance of Ref2 on the same problem. Ref2 is the immediate predecessor of POPS, and contains the same basic machinery including formula manipulation processes, data representation, and constraint satisfaction methods. The design of Ref2 is sufficiently similar to that of REF-ART, except in its data representation, that comments about the performance of Ref2 apply equally well to REF-ARr. The performances of Ref2 and POPS on the problems discussed are summarized in Figure 9.

The GPS-like control structure of POPS and the basic control of search provided by the Q-size rule and the redundancy test, provide a skeleton of a problem-solver. The sources of problem solving power in POPS are the processes that analyze the PSL *program* to get a description of the operators, select the operator to apply during FIXPATH, and decide whether to assign values to variables during FIXPATH, and if so, what values to assign. The performance of POPS depends directly on the adequacy of these description and comparison processes. These processes are rather rudimentary at present, yet they suffice to guide POPS to the solution of some simple US problems that REF-ARF and Ref2 were unable to solve with reasonable effort.

The Missionaries and Cannibals Problem

Ref2 is unable to *solve the Missionaries and* Cannibals problem as stated in Figure 4. The problem statement contains only one operator. Executing the operator requires that two selections be made from a range of three possible values. Ref2 approaches this program by executing directly to the end, where it encounters a constraint satisfaction problem. If it finds no solution to the constraint satisfaction problem, Ref2 attempts an alternate path to the end. Since the solution of the problem requires eleven crossings of the river, and each crossing is accomplished by one execution of the loop in the program, the first ten constraint satisfaction problems will have no solution. Since each crossing of the river *generates two* additional variables, we may estimate the size of the nth constraint satisfaction problem to be K^n, where K is the average number of possible selections generated by one crossing of the river.

On being given this program, Ref2 ran out of time at eight minutes, having failed to complete the search on the fifth constraint satisfaction problem. If we assume K = 3, then the total space Ref2 would search through the fifth constraint problem would contain about 540 nodes, so that the search rate established by Ref2 is roughly 540 nodes per eight minutes. Under these assumptions, the total space to be searched by Ref2 up through the tenth crossing of the river is roughly 133,000 *nodes. In short, Kef2 would* start on the final crossing of the river after computing for something in excess of 32 hours on the problem.

Whether the above estimate is accurate is immaterial; the point of consequence here is that delaying the determination of actual values for the nondeterministic selections causes Ref2 to expend a tremendous amount of time on redundant search. The space of the Missionary problem is actually remarkably small - only sixteen distinct legal configurations can be reached from the initial node.

In contrast to Ref2's estimated time of 32 hours, POPS solves the Missionary problem in somewhat less than six minutes. The search conducted by POPS is shown in Figure 5. This success is based on a method of assigning actual values to nondeterministic selections before the end of the program has been reached.

When to assign actual values. An assignment is made only if failing to do so would deepen the space of the constraint satisfaction problem at the end of the program. For instance, if an operator contains an assignment such as X = X + SELECT (A), and the current value of X is already SELECT (A), then the eventual constraint satisfaction problem would *contain* variables representing both selections. If the range A contains N elements, the number of possible selections would be N^, but the number of distinct values of X could be as small as 2N. Before applying the operator in such a case, POPS chooses an actual value for the first selection. Then after the operator is applied, the value of X still depends on only one SELECT operator. Consequently, In the Missionary problem, each of the constraint satisfaction problems encountered at the end of the program has only two variables, and the constraint satisfaction problem space searched by POPS in the first ten crossings contains roughly 30 nodes.

Selection of actual values to assign. In choosing an actual value, POPS attempts to find the value that is most likely to be helpful in solving the problem. For example, in the Missionary Problem, where the selections determine the number of missionaries and the number of cannibals to place in the boat for a crossing of the river, POPS makes selections that maximize the number of persons being transported to the right side of river and minimize the number being transported to the left side of the river. The sense of direction this gives the search can be seen in Figure 5.

When POPS selects an operator, it is attempting to find one that will modify the current state of the program in a direction that will tend to satisfy some currently unsatisfied constraint. This constraint is also used to guide the selection of actual values. For example, in the Missionary problem, the constraint on the exit branch of the program says, among other things, that the number of missionaries on the right side of the river, MR, should be 3. Suppose the current value of MR is SELECT (A), where A = {0, 1, 2}. Then the first choice POPS will make for the value of this selection is 2, since that choice minimizes the difference between the

current and desired values of MR. If the current value of MR were -SELECT(A), then the first choice would be 0, for the same reason.

The Monkey Problem

The Monkey problem has been formulated in three different ways. The performance of POPS on these three problem statements illustrates its dependence on the information it can derive from the PSL program. Briefly, one version contains sufficient information that POPS makes no wrong decisions in the solution process. The other two formulations require some search. Flowcharts for these formulations are shown in Figure 6.

MB4 .and MB2. The first version of the Monkey problem will be referred to as MB4, because the constraint on the exit branch of the program specifies what the values of each of the four variables in the program must be. The second version will be called MB2, because it is identical to MB4 except that the constraint on the exit branch of the program specifies the values of only two variables. MB4 requires the monkey to be on the box under the bananas end the box to be on the floor under the bananas. MB2 requires only that the monkey be on the box under the bananas, and says nothing about the box. Of course, because of the structure of the problem, the box must be under the bananas and on the floor in order for the monkey to be on the box under the bananas, so that any solution of MB2 is a solution of MB4.

POPS does not recognize the necessity of moving the box in MB2 until it fails to solve the problem without moving the box. Thus, POPS obtains less guidance from the constraints in MB2 than it does from the constraints in MB4; as a result, while POPS goes directly to a solution in MB4, it is forced to perform some search in MB2. The executions of MB4 and MB2 by POPS are *shown in* Figure 7.

Because M B4 and MB2 are identical except for the constraint on the exit branch, and because Ref2 gets no direction from the constraints, Ref2 carries out the same search on MB4 as it does on MB2. Thus, because of its ability to use constraints to guide the selection of operators, POPS not only searches a smaller space than Ref2 on MB2, but is also able to make use of the additional Information in MB4 to reduce the search further.

MBF. The third version of the Monkey problem that POPS has run, called MBF, is a translation of Fikes's statement of the problem for BEF-ARF. MBT was considered in order to get a direct comparison of Ref2 and POPS with REF-ARF, since Fikes does not report running problems stated like MB2 and MB4. The search carried out by REF-ARF on MBF generated 28 nodes.

The chief difference between MBF and MB4 is that the control structure in MBF carries some information of potential value to the problem solver, whereas MB4 is in strictly HS form. Ref2 performs considerably better on MBF than on MB4. POPS, however, finds a solution for MBF after applying only one operator, generating a space of only two nodes.

Robot and Two Boxes Tasks*

This is a sequence of simple robot tasks with a single initial situation. In the initial situation, there is a room, and A, B, O, and D are locations within it. The robot is at location A, and two boxes, B1 and B2, are at location B. The following actions are available to the robot: it can walk to any location in the room; if the robot and box B2 are at the same location as box B I, the robot can stack B2 on top of BI; and if the robot is at the same location as Bl , the robot can push Bl to any location in the room. Thus, the only way the robot can move B2 to another location is by stacking it on Bl and pushing B1. Also, the robot must know whether B2 has been moved or not.

There are five tasks in the sequence. They are;

1: The null task
2: Move the robot to location C
3: Move Box Bl to location C
4: Move Box B2 to location C
5: Move Box 82 to location C and Box Bl to location B

These tasks were intended to be of gradually increasing complexity so that some indication of the effect of increasing complexity on the performance of POPS could be obtained. This sequence of problems was also given to Ref2 for comparison purposes.

POPS executions of the robot problems. The null task, problem1, is included in order to determine the cost of initialization; POPS spends 9.9 seconds solving this problem. There is no search. The second task requires *one operator,* WALK. POPS finds and appllestbis operator in 2,5 seconds. Problem 3, move box Bl to C, is also simple, because the robot can push the box. The solution requires two steps, however- POPS first attempts to apply the operator that pushes B1. Before that operator can be applied, however, it is necessary to apply WALK, to get the robot to the same location as B I. POPS selects and applies these two operators in 6.9 seconds, or 3.45 seconds per subgoal. To move Box B2 to C {problem 3*),* it is necessary to push Bl over to B2, stack B2 on B I, and then push them both to C, requiring a total of four operators and taking POPS 17,8 seconds plus initialization, or 4.45 seconds per subgoal. In the fifth problem, POPS must move B2 to O as in problem 4 and then unstack B2 from Bl and push Bl to location B. After stacking B2 on Bl and pushing them to location C, POPS erroneously selects WALK Instead of attempting to push B1, thereby generating seven subgoals Instead of the necessary six, for a cost of 35.1 seconds, or 5 seconds per subgoal.

Figure 8 shows the search POPS conducted on these problems. The performance of POPS on these problems

*In fact, Ref2 runs one second faster on MB2 than on MB4; the reason is that the shorter constraint in MB2 takes less time to evaluate.

*These problems were suggested by a similar set of tasks given by Raphael (1971). Raphael uses these tasks to illustrate the frame problem.

is summarized in Figure 9.

Ref2 executions of the robot problems. Because Ref2 performs essentially no preliminary analysis of a PSL program, it can do the null robot task faster than POPS. On the more complex task, however, Ref2 becomes less effective than POPS. This is to be expected in problems where POPS is able to derive some guidance from the problem statement. See Figure 9.

Conclusion

The mafor accomplishment of POPS is the application of goal directed methods to the execution of nondeterministic programs. The analysis of the effects of operators and the selection of operators on the basis of. a match between the current situation and the desired situation allows POPS to display a sense of direction in its search, and not rely merely *on* generate and test search, as other processors for nondeterministic languages do. POPS contains discrete processes for describing operators, describing goals, selecting operators, and selecting values to assign. These processes may be improved almost independently of each other, and without modification of the basic system. Consequently, the design will support substantial improvement in problem solving ability.

Bibliography

1. Bolce, J.F., "LISP/360, *A* Description of the University of Waterloo LISP 1. 5 Interpreter for the IBM System/360," Computing Centre University of Waterloo, Waterloo, Ontario, Canada, 1967.

2. Ernst, George W. and Newell, Allen, GPS: A Case Study in Generality and Problem Solving, Academic Press, N. Y., 1969.

3. Feigenbaum, E. and I. Feldman (eds.), Computers and Thought, McGraw-Hill, 1963.

4. Fikes, Richard Earl, A Heuristic Program for Solving Problems Stated as,Nondeterministic Procedures, Doctoral thesis, Carnegie-Mellon University, 1968.

5. Fikes, Richard Earl, REF-ARF: A System for Solving Problems Stated as Procedures, J. Art. Intel. 1 (1) 1970.

6. Floyd, Robert, "Nondeterministic Algorithms, " J. ACM 14 (4) 1967.

7. Gibbons, G, D., Beyond REF-ARF: Toward an Intelligent Processor for a Nondeterministic Programming Language, Doctoral Thesis, Carnegie-Mellon University, 1972.

8. Newell, A. and Simon, H. A.,"GPS, A Program that Simulates Human Thought," in Feigenbaum and Feldman, 1963.

9. Raphael, B., "The Frame Problem in Problem-Solving Systems," in Artificial Intelligence and Heuristic Programming,, Edinburgh University Press, 197].

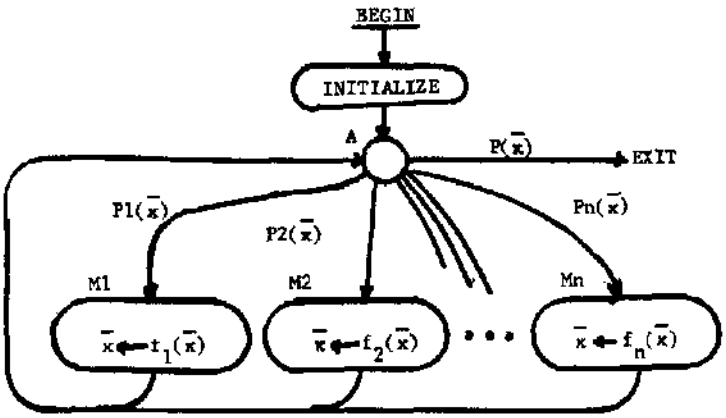


Figure 1.

In a room is a monkey, a box, and some bananas hanging from the ceiling. The monkey wants to eat the bananas, but he cannot reach them unless he is standing on the box when it is sitting under the bananas. How can the monkey get the bananas?

Figure 2a. English Statement of the Monkey Problem

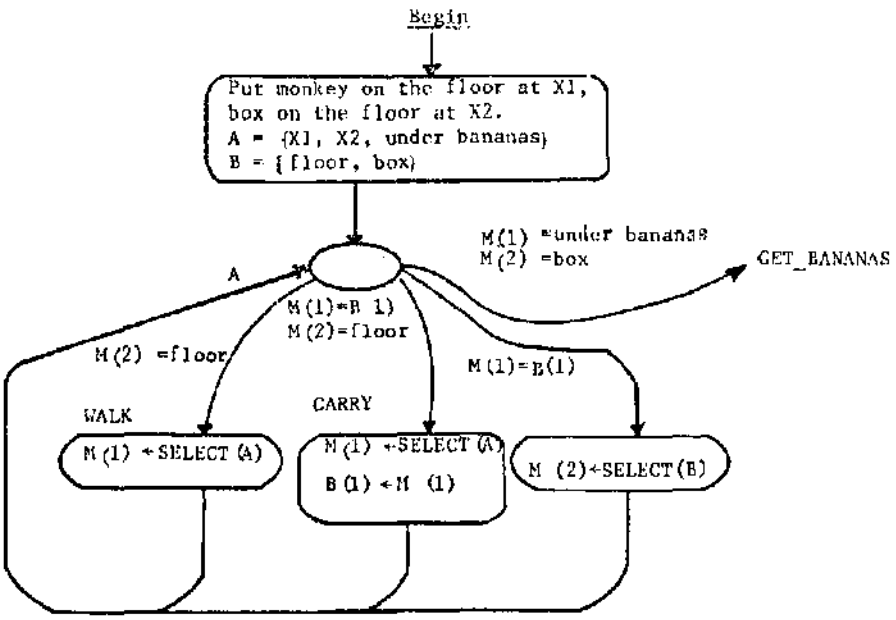


Figure 2b. Flowchart Statement of the Monkey Problem

The English statement of the problem is taken from Fikes (1968). In the flowchart, each time the program arrives at A it may choose any branch. Set A contains the locations on the floor of the cage, and set B is the set of vertical locations (on the floor and on the box.)

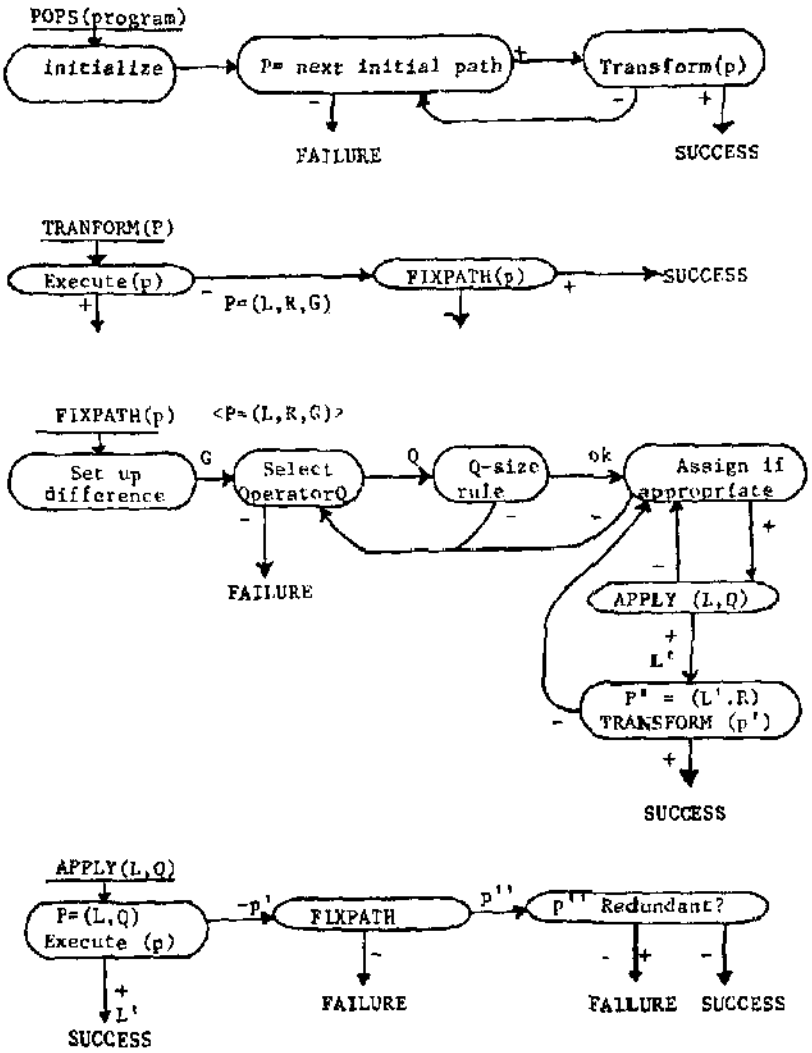


Figure 3 POPS Control Structure

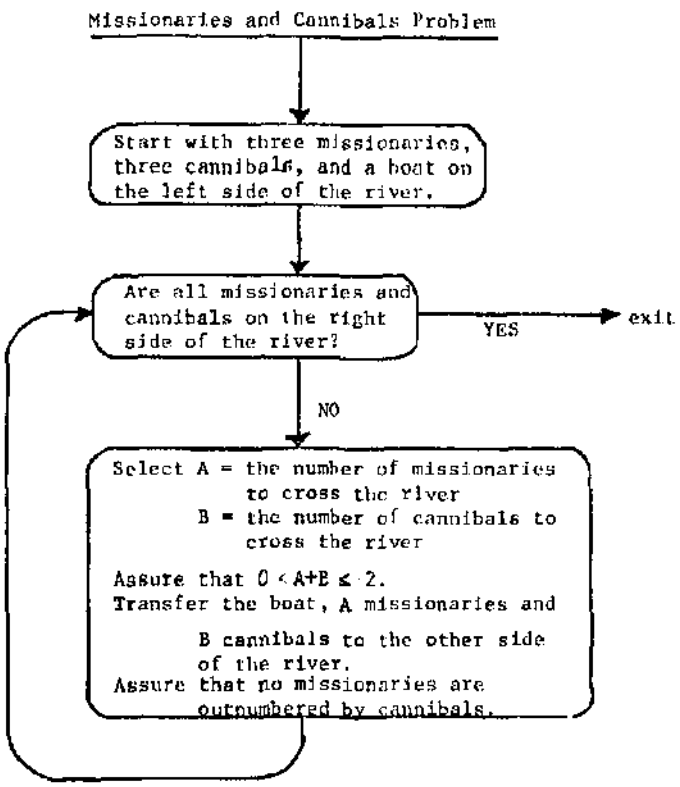

Missionaries and Cannibals Problem
Start with three missionaries, three cannibals, and a boat on the left side of the river.
Are all missionaries and cannibals on the right side of the river?
YES
exit
NO
Select A = the number of missionaries to cross the river
B = the number of cannibals to cross the river
Assure that 0 < A+B ≤ 2.
Transfer the boat, A missionaries and B cannibals to the other side of the river.
Assure that no missionaries are outnumbered by cannibals.

Figure 4.

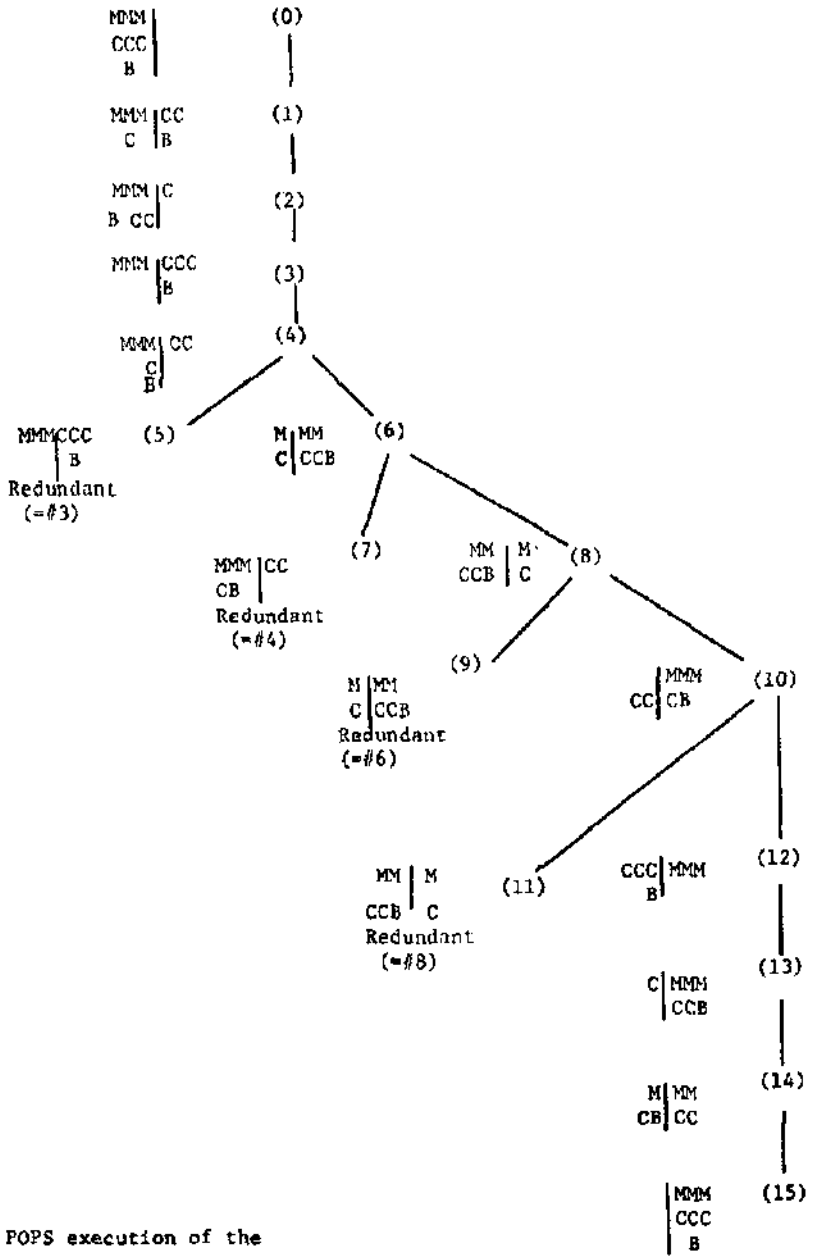


Figure 5 POPS execution of the Missionary Problem

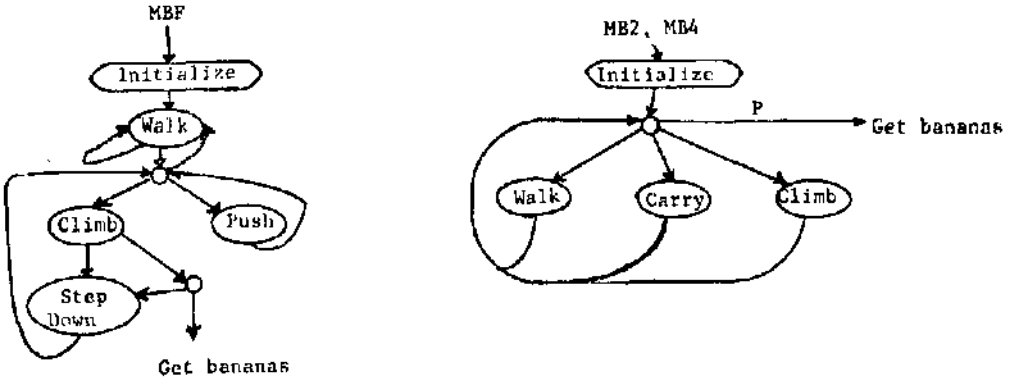


Figure 6. Flowcharts of Three Versions of the Monkey Problem.

MBF is the version Fikes programmed for REF-ARF. MB2 and MB4 differ in the condition P on the exit branch: MB4 requires the monkey to be on the box and under the bananas, and the box to be on the floor under the bananas. MB2 requires only that the monkey be on the box and under the bananas. In all three versions the monkey must be on the floor to be able to walk, and must be where the box is to be able to climb or push the box.

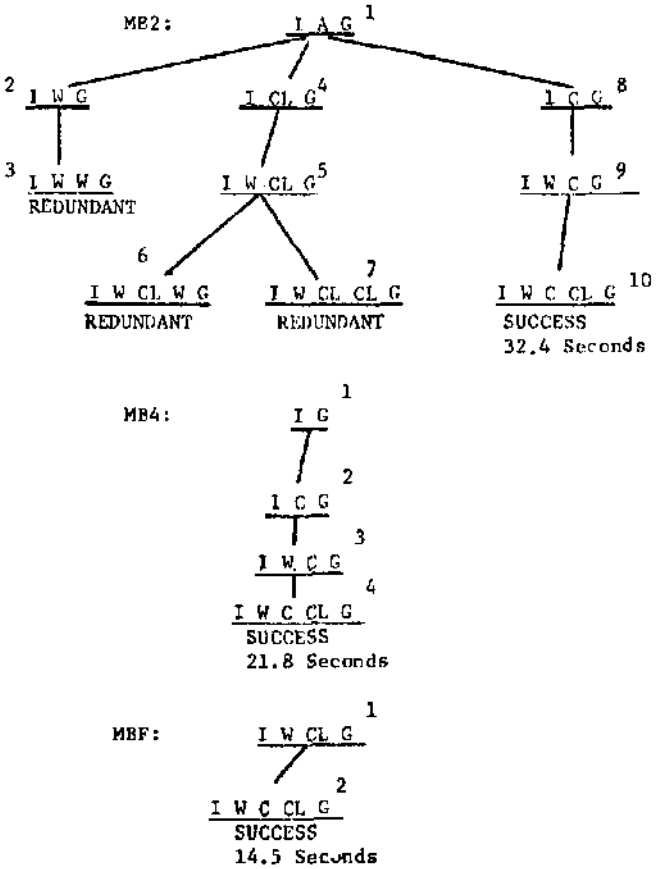


Figure 7. Spaces Searched by POPS on the Three Versions of the Monkey Problem.

Each node is a path through the PSL program, indicated by abbreviations for the labels on the path. The abbreviations are: I (initialize), W (walk), C (carry, or push), CL (climb), G (get bananas). POPS obtains sufficient guidance from the constraint in MB4 and the control structure of the PSL program in MBF to go directly to the solution. In MB2, POPS does not attempt to move the box until node 8, because the box is not mentioned in the constraint that is used to guide the search.

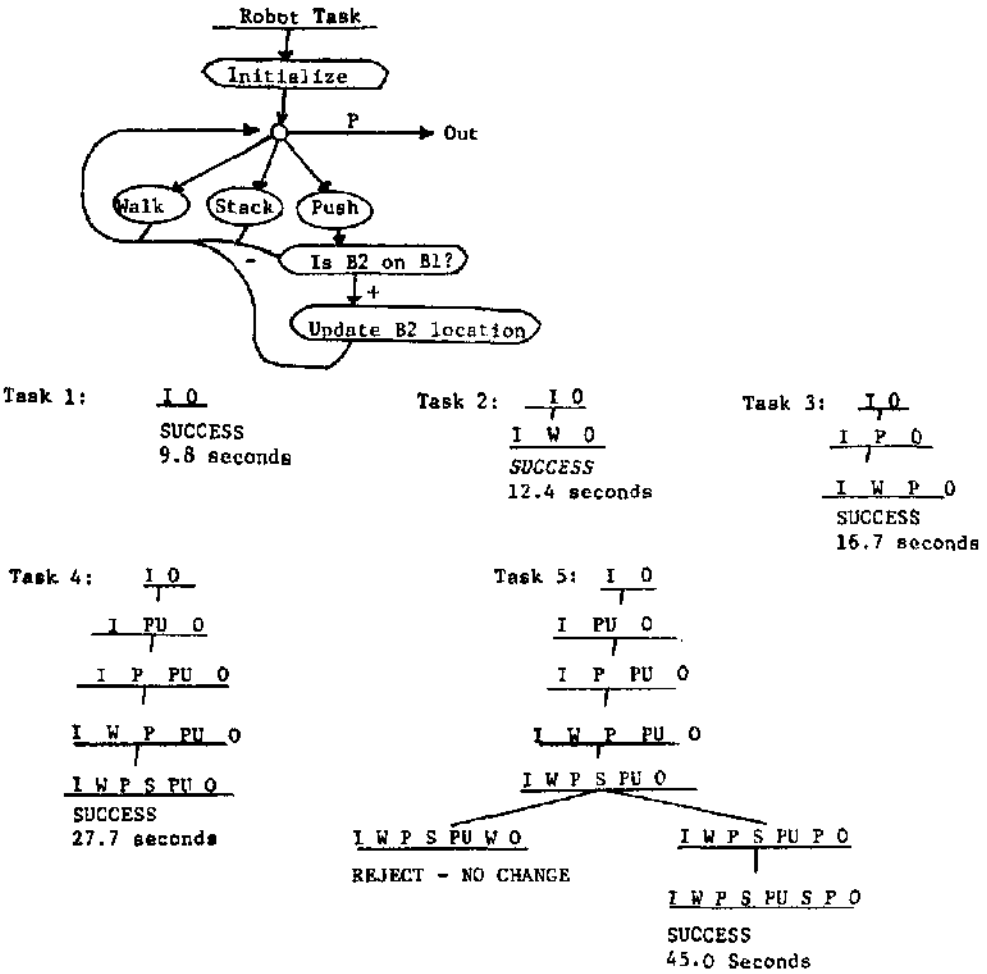


Figure 8. Flowchart and Search Spaces Generated by POPS on the Sequence of Robot Tasks.

In the flowchart, the task is imposed by the condition P on the exit branch. The robot may walk to any point in the room, stack box B2 on box B1 or remove B2 from B1 if both boxes and the robot are at the same location in the room, or push box B1 to any location in the room if the robot and B1 are at the same location. If the robot pushes B1 while B2 is stacked on B1, then the location of B2 must be updated. The nodes in the search spaces are paths through the flowchart. The abbreviations are: I (initialize), W (walk), S (stack or unstack), P (push), PU (push and update B2), and O (out).

	POPS Nodes	POPS Time (Sec/Node) with (without) set up time	POPS Total Time with (without) setup	Ref2 Time (sec)
MB4	3	7.26 (5.3)	21.8 (15.9)	29.0 sec
MB2	9	3.6 (2.9)	32.4 (26.5)	28.0 sec
MBF	1	14.6 (7.6)	14.6 (7.6)	11.7 sec
MISSIONARY	15	23.7 (22.7)	355 (341)	Unknown
R2B1	0		9.9 (0)	2.9 sec
R2B2	1	12.4 (2.5)	12.4 (2.5)	7.7 sec
R2B3	2	8.4 (3.45)	16.8 (6.9)	16.9 sec
R2B4	4	6.93 (4.45)	27.7 (17.8)	34.7 sec
R2B5	7	6.43 (5.01)	45.0 (35.1)	54.5 sec

Figure 9. Performance of POPS and Ref2 on Selected Problems.

R2Bi is the i^{th} Robot and Two Boxes task. The times given for Ref2 include the small time Ref2 spends setting up the problem. The setup time in the figures for POPS refers to the time POPS spends translating the PSL program into a MS problem statement and obtaining a description of the detours.