Cognitive Psychology Learning and Problem Solving

Citation for published version:
Bundy, A 2000 'Cognitive Psychology Learning and Problem Solving'.

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
4.5 The horizon effect

Game-playing programs based on the mini-maxing paradigm now seem to have reached a plateau. For instance, chess programs have not improved their performance radically for several years. Their standard can only be improved by increasing the size of the look-ahead tree or by improving the evaluation functions used to determine the static evaluations (and this means figuring out precisely what features are important and what their weights should be). But (even assuming we knew how to improve the static evaluation) both of these take computing time and existing programs are already spending as much time in deciding their next move as is allowed under tournament rules (between two and three minutes). How could we write a program which could play chess at master level?

An analysis of the kind of mistakes made by existing programs has been made by Berliner (1977). He describes the horizon effect in which the program fails to take into account consequences of a move which happens outside its limited look-ahead tree. An example of this is when the capture of a valuable piece (say a queen) is inevitable. Some of the capturing situations are outside the look-ahead tree and so not accessible to the program. It will choose a move which postpones the evil day and may, as a consequence, be forced to sacrifice other pieces.

This is illustrated by the following tree. To understand the example it is not necessary to understand chess. You need only know that losing a bishop is bad (—10 for white), but losing a queen is a disaster (—20 for white).

In this situation, white avoids moves A and B because they result in the capture of his queen. So he chooses instead move C (leading, he believes, to a state worth —10). However, he not only loses his bishop as expected, but two moves later he loses his queen as well (effectively accumulating his total loss to —30)!

This difficulty is a general one, because even if the program manages to look ahead several more moves, there is always the possibility that yet another disastrous situation is looming just beyond the horizon! The problem is that the kind of programs we have been discussing cannot reason strategically, as we did, about the inevitability of the queen's capture, and thus they cannot plan accordingly. Such programs are constrained not only by the arbitrary imposition of a horizon (i.e. some limit at which look-ahead must stop), but also by the fact that they proceed in terms of sequences of isolated moves, with no overall strategy. In addition, their knowledge of what is taking place in the game is blurred by having several weighted features all combined into one numerical value. This numerical value may be useful for mini-maxing, but it is of no help in reasoning.
strategically. A program capable of playing master chess will have to be able to reason at several levels, using the recognition of patterns to evoke overall strategic and tactical reasoning, which is only then checked out at the level of individual moves.

Now read the commentary in Johnson-Laird and Wason, pp. 527-31 on the problems of developing a master level chess program (start in the middle of p. 527, and don’t worry about following through the actual chess moves). You may also be interested in reading Berliner’s article, in which he discusses some of these issues in more detail (in Johnson-Laird and Wason, Reading 34 – optional).

4.6 A new generation of programs

A new generation of chess playing programs are currently being developed which use prestored patterns to evoke plans of action. One of the problems is to get a sufficiently flexible representation of a pattern. Simple structural configurations (e.g. a piece about to capture another piece) are necessary but not sufficient. Generalized structural configurations (e.g. good arrangement of all your pieces on the board) and functional patterns (e.g. a fork, a threat) are also necessary. The problem with representing these more sophisticated types of pattern is that the same pattern may be realizable in a wide variety of structurally different ways. In both the examples below the white queen is in a position to capture either a black knight or a black rook (this is an example of a fork), but notice that the actual positions of the pieces are quite different.

Simple data structures of the kind we have been discussing so far are inadequate to represent such patterns and the programmer needs to look to more sophisticated symbolic descriptions and/or procedures.

The new programs work by recognizing patterns like the forks above and evoking strategic and tactical reasoning to deal with the situation. The reasoning processes ask questions which initiate searches for further patterns, build plans to deal with the situation, and have these checked out at the detailed level of legal moves. Here is an example of such reasoning:

‘Are either of the forked pieces protected by other pieces?’
‘Try to move one forked piece out of danger so that it simultaneously protects the other piece.’
‘Can the rook be moved to protect the knight?’ etc.

Defining these patterns and this strategic and tactical reasoning in sufficient detail for them to be programmed is a hard task. Chess books can be consulted for some of it, but much remains to be supplied by the programmer. Progress in building master level chess programs is slow but advancing.
4.7 Experimental investigations

Experimental investigations of human game playing confirm that humans use such overall patterns to remember board positions and also suggest what form those patterns might take.

De Groot’s (1965) classic study refuted the myth that chess masters have phenomenal memories. He showed that their ability to recall random (non-meaningful) positions of pieces on a chessboard was no better than the ability of chess beginners. However, if the chess positions were meaningful (i.e. arose in a real game) then their ability improved dramatically and the beginner did poorly by comparison. The recall task consisted of allowing them to view a chess position for five seconds, and asking them then to try to reconstruct it from memory.

The experiment suggests that the chess masters were able to structure the meaningful positions into standard patterns, and that such structuring can make the recall task easier, because less things have to be stored in short-term memory. For instance, suppose the board contains twelve pieces. The beginner needs to remember the positions of all twelve pieces. However, suppose the master can structure the board into three well known patterns (e.g. a fork, an attacking pawn position, and defence of the king) containing three, four and five pieces respectively. Assuming the details of each pattern are stored in long-term memory, the master needs to remember only the positions of the three patterns. This example is oversimplified but illustrates the general idea.

Chase and Simon (1973) have conducted experiments to test de Groot’s hypothesis and to discover what these patterns might be. They used a memory reconstruction task like de Groot’s and also a task of copying a meaningful position from one board to another in which subjects were allowed to glance back to the original board. Subjects were observed in order to discover whether they seemed to cluster the chess pieces into groups and, if so, what these groups were. Groups were defined by observing the order of placement of pieces, the times between placements in the memory reconstruction tasks and the groupings of pieces placed between glances in the copying task. Successively placed pieces within the same group tended to exhibit more meaningful relations (e.g. attack or defence) than those between groups.

If you are interested in the way such configurations might be represented in human memory, and the way in which the representation of a configuration can affect the way in which a person visually scans a given board position, look at Reading 33 by Eisenstadt and Kareev in Johnson-Laird and Wason (optional reading).

Other data collected by de Groot (1965) and Newell and Simon (1972) shed some interesting light on the way in which chess players appeared to explore through the tree of move possibilities. Rather than employing either a depth first or breadth first search strategy, like those discussed in sections 2.4 and 2.5, the chess playing subjects appeared to use a ‘progressive deepening’ search strategy: they would explore one pathway for several moves in depth, then back up right to the beginning and explore the same pathway once again, sometimes pushing even further in depth, and sometimes veering off on a new move possibility towards the very end of the path, as illustrated by the three successive exploration
sequences shown below:

The techniques for collecting this kind of data from problem solving subjects, and the way in which it is analysed, will be discussed in depth in Unit 28, under the headings 'verbal protocols' and 'problem behaviour graphs' (sections 2.1 and 2.2).

In analysing the 'tree searching' behaviour of their game-playing subjects, Newell and Simon found that the choice of which pathway to follow, or indeed which move to make, is not determined by a simple mini-maxing rule. A human player may choose the first promising move which is discovered and just not worry about finding a 'great' or 'best' move. The player does this by choosing the first move whose effective worth (i.e. score) is greater than some arbitrary threshold value. Newell and Simon termed this principle 'satisficing', and it appears to be quite a handy short-cut to minimize the amount of search necessary to find a satisfactory move. It is, of course, an error-prone short cut because an even better move can easily be missed! People do make many blunders in the course of game playing, and the principle of satisficing, seen as a short cut variant of mini-maxing, is a useful way of accounting for many of these blunders.

Progress box three

Mini-maxing

<table>
<thead>
<tr>
<th>Type of problem</th>
<th>Problems in which an opponent's moves must be taken into account (e.g. board games), and to which the solutions involve large search trees (e.g. chess).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formalization of problem</td>
<td>Same as exhaustive search and heuristic search (i.e. a tree of states in which the branches represent alternative legal move possibilities). The opponents, of course, take turns, and this is automatically depicted in the search tree.</td>
</tr>
<tr>
<td>Method of solution</td>
<td>Calculates dynamic 'goodness' scores for all your next possible moves by working backwards from the static evaluation of terminal states to assign ('inherit') scores to preceding states, assuming your own moves always aim to achieve best scores (for you), while your opponent inevitably aims to achieve worst scores (for you).</td>
</tr>
<tr>
<td>Advantages</td>
<td>Good move choices can be made despite the intervention of opponent's moves, because the 'harmful' effect of those moves is taken into account. Simple 'learning' can be demonstrated by allowing the weights used by the evaluation function to be altered as more information is obtained.</td>
</tr>
</tbody>
</table>
Disadvantages

As with all heuristic search, mini-maxing is only as good as the evaluation functions it uses. Since these are likely to be imperfect, mini-maxing suffers from the lack of *principled* (strategic) move choices. Another disadvantage is that heuristic search starts off with all possible moves whether they are relevant to the final goal or not.

Relevance to humans

People must take their opponents' moves into account when deciding on a move, although they do not seem to do so in the 'even-handed' manner prescribed by mini-maxing.

5 THE GENERAL PROBLEM SOLVING PROGRAM AND MEANS-ENDS ANALYSIS

5.1 Inadequacies of heuristic search

The heuristic search paradigm, where numerical evaluation functions guide the search, worked quite well on puzzles like the missionaries and cannibals, or the Fifteen Puzzle. Mini-maxing, its modification to game playing, was moderately successful. In all of these cases it was relatively easy to specify a set of legal moves. The performance of heuristic search on forming plans of action, like making a plan to get from my house in Edinburgh to Trafalgar Square, London, is hopeless. The problem is that, starting from 'me at my house in Edinburgh', the number of things I can do is immense (infinite?) and most of them are totally irrelevant to getting to London. A heuristic search would waste an awful lot of time following up false trails. Here is what a search tree might look like:

![Heuristic search tree of Trafalgar Square example]

We could try to guide the search by using a (numerical) evaluation function. An obvious measure to use would be the physical distance of me from Trafalgar Square in the current state. The fewer miles I am from Trafalgar Square the better the states, so that 'me in York' would be better than 'me in Edinburgh' and 'me in St Albans' would be even better. The problem is that many of my initial steps, like packing or phoning for a taxi, would appear pointless, and going to Waverley railway station (which is north of my house) positively counter-productive. Of course, I could improve the evaluation function to take account of...
these things, but then the special purpose evaluation function would be doing nearly all the work and the general heuristic search mechanism would be contributing almost nothing to the solution of the task. What is needed is a new way of thinking about the problem – a new problem-solving paradigm.

5.2 An informal introduction to means–ends analysis

A suitable paradigm was provided by Newell, Shaw and Simon (1958), the authors of the General Problem Solving Program (usually called GPS). The paradigm is called means–ends analysis and was embodied in GPS. The basic idea is that each action I put in my plan (my means) should be there for some reason, i.e. it should contribute to one of my goals (my ends). To see how means–ends analysis can help build up a plan consider the following (invented) protocol of me building a plan:

My goal or end is to transform 'me at home' into 'me in Trafalgar Square'. The first task is to compare these two states and find the difference. I find the difference to be one of location. The means I have of reducing differences of location are operators like 'walk' or 'go by train'. Some operators, like 'walk', can be rejected as not feasible, but 'go by train' is feasible, so my next goal is to apply this operator to the initial state, 'me at home'. Unfortunately the operator will not apply immediately because the conditions are not right – I am not at Waverley station (in Edinburgh). So I set up a new sub-goal to reduce the difference between 'me at home' and 'me at Waverley station'. Again the difference is one of location and again I find the 'travel' operators. I can reject 'walk' as not feasible (I am lazy) and 'go by train' as a potential loop (I am already considering going by train from my home) and select 'go by taxi'. This cannot be applied because the conditions are wrong – the taxi driver does not know I need him. The difference is one of information, so I look for an operator which can reduce differences of information and find the communication operators like 'use the telephone'.

This kind of analysis can be carried on to any required depth and will eventually produce a plan consisting of a sequence of operators (moves). Operators in means–ends analysis correspond to moves in heuristic search and, as in heuristic search, the choice of an operator is guided by the overall aim of reducing the distance between the initial state and the final goal state. But the difference between heuristic search and means–ends analysis is that with heuristic search the distances are all measured quantitatively, whereas means–ends analysis allows the distances (differences) to be evaluated qualitatively (e.g. location versus information). This allows an appropriate and feasible operator to be selected to deal with each particular difference. Thus means–ends analysis uses its own qualitative version of essentially heuristic information to guide problem solving. Notice that the above protocol is similar to the kind of strategic planning which I said in sections 4.5 and 4.6 would be useful to incorporate in a chess playing program.

In the protocol above we dealt with goals (and subgoals) of three types:

1. transforming one state into another, e.g. transforming 'me at home' into 'me at Trafalgar Square';
2. reducing a difference, e.g. reducing differences of 'location';
3. applying an operator, e.g. applying 'go by train' to 'me at home'.

Faced with a goal of the first type, to transform one state into another we first found a difference between the two states and then set up a subgoal of the second type to reduce this difference. To solve 'reduce difference' goals, we first found
an operator relevant to reducing this kind of goal, then checked that it was feasible and finally set up a subgoal of the third type, to apply this operator. 'Apply operator' goals were solved by first finding the difference (if any) between the conditions of the operator and the current state, then setting up a subgoal of the second type to reduce this difference.

5.3 Means–ends analysis – the flow charts

Means–ends analysis is embodied in GPS as a series of procedures, here called methods. There are three methods, one for each type of goal. They are explained by the following flow charts. Each box represents a procedure to be activated. The arrows between them indicate in what order they are to be activated. Some of the labels on the arrows specify under what conditions a particular branch is to be taken, e.g. 'fail' means 'take this branch if the procedure fails'. Some of the labels indicate the result of a procedure, e.g. A' in method 1 is the result of reducing the difference D between A and B.

Method 1
Goal: transform state A into state B

This method tries to transform A into B by setting up a subgoal of reducing a difference D between them. This subgoal will be handled by method 2. Unfortunately, the difference D may be reduced without being eliminated. In our Trafalgar Square example the outcome of reducing the difference of location between 'me at home' and 'me at Trafalgar Square' is 'me at King's Cross'. Also, there may be other differences still to be reduced, for instance, I might be required to be wearing a carnation at Trafalgar Square. Thus, there is also a difference of dress between the initial and final state. In either case there is still some work to be done before state A is transformed into state B. Therefore in method 1 the result of reducing difference D is a new state A'. We hope A' is identical to B, but in case it is not a new subgoal is set up to complete the job and transform state A' into state B.

Method 2
Goal: reduce difference D between state A and state B to produce state A'

This method tries to reduce difference D by applying some operator to state A. First, it finds a relevant operator Q. It tests Q to see if it is feasible and, if it is, sets up a new subgoal of applying Q to A to produce a new state A'. This subgoal will be handled by method 3. Failure at any stage causes backup and a search for another relevant operator. The method fails only when all relevant operators have been tried unsuccessfully.
Method 3

Goal: apply Operator Q to state A to produce state A'

- match condition of Q to A to find difference D
- subgoal: reduce D to produce A''
- subgoal: apply Q to A'' producing A'

produce result A' → A'' → success

This method tries to apply operator Q to A, producing A'. Each operator has a condition which must be satisfied before it can be applied. Satisfying this condition is the main task of this method. Once the condition has been satisfied the actual application of the operator is straightforward and is handled by the box marked 'produce result A''. The condition is a bit like a state. To satisfy it, the difference D between it and state A is found. Difference D is reduced by a call to method 2, producing new state A''. We hope the conditions of Q are now satisfied in A'' and Q can now be applied directly. However, in case there are further differences to be satisfied method 3 is called again to apply Q to A'' producing A'.

These methods are the general part of GPS. To solve a particular problem GPS must be given information about the operators, states, goals and differences which are relevant to that problem. This task specific information will be explained more fully in section 5.6. It consists of descriptions of the initial state; goals and operators; procedures for finding differences, for finding relevant operators, for testing for feasibility, and for applying operators.

5.4 Problem reduction

Note that each method works by breaking (or reducing) its goal into subgoals. This technique is called problem reduction. These subgoals may be of the same or different type to the original goal. Does this process ever end or will it go on forever? It can end if all the goals are eventually reduced to trivial goals, that is goals which can be solved by a method on its own without setting up further subgoals, for example, transforming ‘me at Trafalgar Square’ into ‘me at Trafalgar Square’ (the ultimate in trivial goals!) or applying ‘go by train’ to ‘me at Waverley’.

SAQ 13

Method 3 is sometimes required to apply operators, Q, whose conditions are already satisfied in the current state A. These are trivial goals and are dealt with by the box labelled ‘produce result A''. Find another place in one of the methods where trivial goals are dealt with.

With luck then, each method breaks a goal into easier subgoals and this process continues until all the goals are trivial. Notice how this structure is analogous to the hierarchical structuring of procedures and sub-procedures within SOLO (see Units 3-4, section 6.6).
GPS attempts to ensure that goals get easier by ordering the differences according to difficulty. A typical order might be: location; communication; dress. There is a constant check to see that there is no attempt to reduce a hard difference as a subgoal of reducing an easier difference.

Each method can activate other methods, including itself. For example, in method 3 the last box (apply Q to A", producing A") specifies a subgoal which is actually a new instance of the original method itself (i.e. apply Q to A, producing A'). Just as when a procedure activates a fellow member of its ‘reserve pool’ in SOLO, this is called recursion (see Units 3–4 pp 78–82). Recursion is not circular provided care is taken that the same method is not used twice to solve the same goal. This happened in our Trafalgar Square protocol where we rejected a second attempt to apply ‘go by train’ to ‘me at home’ as a potential loop. GPS incorporates a loop checking mechanism which is on the look out for such possibilities.

SAQ 14

Method 1 sets up a method 2 subgoal (‘reduce difference’) and a method 1 subgoal (‘transform states’). Thus it activates both method 2 and method 1 (the latter is a recursive activation). Which methods do methods 2 and 3 activate?

5.5 And/Or trees

Look back at the problem reduction tree above. Compare it to the search tree for the missionaries and cannibals problem on p 17. There is a crucial difference. In the missionaries and cannibals problem the links represent alternative solution paths. If one fails we can back up and choose another. In the problem reduction tree all the subgoals of a goal must be solved for the goal to be solved. The links do not represent alternatives. We have indicated this by joining the links together with arcs in the problem reduction tree. To further distinguish the two situations we will call the nodes in the missionaries and cannibals tree Or nodes, because the links are genuine alternatives. We will call the nodes in our problem reduction tree And nodes, because the links must all be dealt with eventually for the problem to be solved.

Actually a problem reduction tree could also contain Or nodes if there happen to be several ways of achieving a goal. The tree below represents the situation in
which goal A is reduced to subgoals B and C (which must both be achieved) but C itself can be achieved in one of two ways: either by solving D or by solving E. Such a tree is called an And/Or tree, because its nodes are a mixture of And nodes and Or nodes. In means–ends analysis, goals of methods 1 and 3 can be represented as And nodes and goals of method 2 can be represented as Or nodes. This is because methods 1 and 3 each generate two subgoals, both of which must be solved, no choice being allowed. Method 2, on the other hand, generates only one subgoal, but if this should prove unsuitable it is prepared to suggest an alternative. This is indicated by the arrow marked 'try for another operator' in the flow chart for method 2.

The protocol in section 5.2 for the Trafalgar Square example can be represented as a search through an And/Or tree. The nodes of the tree will represent not states of the world, but the different goals and subgoals we set up when trying to solve the problem. The links will represent not individual operators, but the three types of method used to solve these goals. I have used a different shaped box for each of the three goal types: rectangular for type 1 (transform A into B), elliptical for type 2 (reduce D) and hexagonal for type 3 (apply Q to A). Look back to the protocol on p 36 and the three methods on pages 37–8 as you work through this tree. Our protocol on p 36 defines a depth first search through the tree, taking the branch furthest to the left at every choice. The nodes are encountered in numerical order. (Nodes 7, 8 and 9 were deliberately omitted – they are to be filled in by you for SAQ 15.)

SAQ 15
The tree on the right accounts for the protocol up to and including the phrase ‘and select “go by taxi”’. Add the extra nodes required to account for the remaining two sentences of the protocol: ‘This cannot be ... “use the telephone”’ i.e. subgoals 7, 8 and 9.

YOUR ANSWER
The above tree is more sophisticated than a heuristic search tree (say the missionaries and cannibals tree in 2.7) in at least two ways. Firstly, it is an And/Or tree as opposed to the plain heuristic search Or tree (i.e. we can represent several steps which are all subgoals of some higher goal). Secondly, the level of representation is higher than we are used to. The nodes represent, not states of the world, but the goals of the problem solver (i.e. mental states!). The links represent not operators or legal moves, but problem-solving methods. On the other hand the tree will be searched in a very simple way, namely depth first.

It is worth comparing this tree with the heuristic search type tree where nodes are world states (e.g. 'me at home') and links are operators (e.g. 'go by train'). Such a tree was shown on p 35, which you should look back to now.

The main difference between these trees becomes apparent if we try to find a correspondence between the trees and the thinking-out-loud protocol presented on p 36. In that protocol my thought (or reasoning) sequence corresponds precisely to the depth first sequence of nodes shown in the And/Or tree for means–ends analysis (p 40). For the heuristic search tree, however, the sequence in which nodes are encountered (even just considering the successful pathway leading towards a solution) is completely different from the sequence in which my mental plan unfolded.

Since heuristic search always grows the search tree downwards from the initial state, reflecting a sequence of physical states rather than plans, it is not at all helpful for understanding our protocol. The situation is a classic one in AI. Faced with an inadequate representation and search technique (say a heuristic search state/operator tree plus evaluation functions) we can try to improve the situation in one of two ways:

- design a more sophisticated search technique;
- design a more sophisticated representation.

The And/Or tree of goals and methods opposite, which embodies means–end analysis, reflects the second alternative.

### 5.6 Task-specific information

We must now return to the question posed at the end of section 5.3. How do we describe a problem to GPS? What information does it need to apply means–ends analysis? This would include telling GPS:

(i) the goal state (e.g. 'me at Trafalgar Square');
(ii) the initial state (e.g. 'me at home');
(iii) the available operators (e.g. 'go by train') and a procedure for applying them to states;
(iv) procedures for finding the most significant differences between states and between preconditions of operators and states (e.g. find a location difference given 'me at home' and 'me at Trafalgar Square');
(v) an account of which operators are relevant to which differences (e.g. 'go by train' is relevant to differences of location);
(vi) an ordering of the difficulties of differences (e.g. location differences are more difficult to deal with than differences of information);
(vii) a procedure for testing the feasibility of an operator (e.g. is 'walk' feasible to travel from 'home' to 'Waverley'?).

(i), (ii) and (iii) are the sort of information any problem solver (e.g. heuristic search) would have to be given. The information provided under (iv), (v), (vi) and (vii), however, is unique to GPS. This information enables GPS to extract
differences and to use the three methods to deal with them. With it GPS can make sensible choices about which operators to use to transform one state into another. Unfortunately (i)–(vii) represents a lot of work each time GPS is set a new problem.

This drawback has been confronted by recent successors to GPS which have tried to reduce the amount of information required to define the problem while retaining the means–ends analysis. These will be described in the next section. Still other programs have concentrated on being more flexible by incorporating the problem-solving tools of GPS within a more general purpose programming language. These will be described in later sections and in Unit 28.

Despite its limitation, GPS remains a valuable source of ideas for later programs. In particular, means–ends analysis and difference descriptions occur in many more recent problem-solving programs. Also the idea of comparing steps in a program with a human protocol (as we did when comparing the And/Or tree with the Trafalgar Square protocol on p 40) has become a valuable tool in the simulation of human behaviour. This particular idea will be taken up again in detail in Unit 28.

<table>
<thead>
<tr>
<th>Progress box four General Problem Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type of problem</strong></td>
</tr>
<tr>
<td><strong>Formalization of problem</strong></td>
</tr>
<tr>
<td><strong>Method of solution</strong></td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
</tr>
<tr>
<td><strong>Relevance to humans</strong></td>
</tr>
</tbody>
</table>
6 STRIPS AND OPERATOR TABLES

6.1 STRIPS and SHAKEY

In this section we discuss a successor to GPS called STRIPS (Stanford Research Institute Problem Solver). STRIPS was designed as the plan formation part of the Stanford Research Institute robot, SHAKEY. SHAKEY is a mobile robot, who lives in a world consisting of seven rooms, connected by doors and containing several large boxes which he can push about.

A typical task for SHAKEY would be to push a box from one room to another. He achieves this task by first making a plan consisting of a sequence of operators, each corresponding to a primitive ability of SHAKEY. These operators would then be executed in turn. Typical operators might be to go from one place to another within a room, or to go through a door. A typical plan composed of these operators might be:

- go from start to location of box;
- push box to door;
- push box through door;
- push box to next door; etc.

For each of these operators, SHAKEY will have an action routine, which when run will move his wheels, etc. in the appropriate way to execute the action.

Decoupled from SHAKEY and the real world, STRIPS can be used as a problem solver on a wide variety of tasks, most of which SHAKEY would not be able to execute. We will explain how STRIPS works using the Trafalgar Square example from the previous section.

6.2 Operator tables

Like GPS, STRIPS solves problems using means–ends analysis. Unlike GPS, however, it is not necessary to provide large amounts of information about differences in order to make the means–ends analysis work. This difficulty is
avoided by making the differences implicit and incorporating the information about them in the definition of the operators. This means that STRIPS can work out for itself the differences and the relevant operator for reducing these differences. The operators are defined by an 'operator table' (described below) which can be used for:

(i) applying operators to states (done by method 3 in GPS);
(ii) finding differences between states and the preconditions of operators (done within methods 1 and 3 in GPS);
(iii) choosing relevant operators (done by special procedures within method 2 in GPS).

This covers all the needs of means–ends analysis, as we will demonstrate in sections 6.3 and 6.4.

These operator tables have four columns: the name of the operator; its preconditions; the delete list; and the add list. The last two columns represent the effect an operator has when it is applied to a state: namely, the facts that are known to be no longer true (the delete list) and the facts that become true (the add list). Below is an example of an operator table for the operator 'go by train'. For the purpose of exposition I have translated it into a notation very similar to that used by SOLO.

<table>
<thead>
<tr>
<th>operator</th>
<th>preconditions</th>
<th>delete list</th>
<th>add list</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO GOBYTRAIN</td>
<td>CHECK [X]-- AT --&gt;[Y] CHECK [Y]-- ISA --&gt;STATION CHECK [Z]-- ISA --&gt;STATION</td>
<td>FORGET [X]-- AT --&gt;[Y]</td>
<td>NOTE [X]-- AT --&gt;[Z]</td>
</tr>
</tbody>
</table>

This table is to be read as follows: 'X can go from Y to Z by train provided he is initially at Y and both Y and Z are stations. The result is that X is no longer at Y, but he is at Z'. That is, provided the preconditions check out, remove everything on the delete list from the data base and add everything on the add list to the data base.

Triples, like [X]-- AT -->[Z], are used instead of the equivalent English phrases (e.g. 'X is at Z') in order to avoid the problem of ambiguity. There are just too many equivalent English phrases, e.g. 'the location of X is Z', 'X is at place Z', 'X is in the station Z' etc. The strokes signify that X, Y, and Z are variables which can stand for any suitable nodes. They play a role very similar to the 'slots' provided by SOLO's parameters, but are not exactly the same, so I have used a slightly different notation, namely vertical rules. The difference is not crucial for the purpose of this discussion, and you can think of each STRIPS variable as an empty slot waiting to be filled with the name of a particular node in the data base. When a given slot has been filled in, we say that that variable has been assigned a specific value. Thus the operator GOBYTRAIN can be used to transport me from Waverley to King's Cross in one plan and a crate of pigeons from York to Newcastle in another plan (although I understand sending pigeons unaccompanied is not now allowed by British Rail).

States of the world are also described using triples. For instance, the initial state of the world with 'me at home' might be represented by a data base consisting of ME-- AT -->HOME, Waverley-- ISA -->STATION, King's Cross-- ISA -->STATION etc. Notice that the absence of strokes implies that ME, HOME, Waverley, etc. are not variables but specific nodes. STRIPS knows that an operator is applicable to a state if it can find specific nodes to assign to the variables (i.e. to fill into the slots), which would make all the preconditions check out (i.e. be present in the data base).
6.3 Using operator tables

How can these operator tables be used for the three tasks of applying operators, finding differences and choosing operators described in the last subsection?

1 Using the table to apply the operator to a state is the easiest to understand. ‘Applying an operator’ is similar to ‘activating a procedure’ in SOLO. Suppose that all the variables are assigned to specific nodes, i.e. STRIPS knows the value of each of the variables. First, each of the preconditions is checked in the current data base. If any check returns ABSENT then the application fails. If all the checks return PRESENT then each of the descriptions in the delete list is forgotten and each in the add list is noted.

Example
Suppose that the current state is represented by a data base consisting of the following three descriptions:

- ME- - - AT - - ->WAVERLEY
- WAVERLEY- -- ISA - - ->STATION
- KINGSCROSS- -- ISA - - ->STATION

Now we want to apply GOBYTRAIN ME WAVERLEY KINGSCROSS. First, check the preconditions:

- CHECK ME- - - AT - - ->WAVERLEY
- CHECK WAVERLEY- -- ISA - - ->STATION
- CHECK KINGSCROSS- -- ISA - - ->STATION

These all return PRESENT.

So forget everything in the delete list, i.e.

- FORGET ME- - - AT - - ->WAVERLEY

Note everything in the add list, i.e.

- NOTE ME- - - AT - - ->KINGSCROSS

The new state is represented by the following three descriptions in the data base:

- ME- - - AT - - ->KINGSCROSS
- WAVERLEY- -- ISA - - ->STATION
- KINGSCROSS- -- ISA - - ->STATION

2 Sometimes an operator cannot be applied to a state. This happens when some of the precondition checks return ABSENT. In this case we say that there is a difference between the state and the preconditions of the operator. In STRIPS the difference is represented precisely by those preconditions which returned ABSENT.

Example
Suppose we try to apply

GOBYTRAIN ME WAVERLEY KINGSCROSS

to the state represented by the following descriptions in the data base:

- ME- - - AT - - ->HOME
- WAVERLEY- -- ISA - - ->STATION
- KINGSCROSS- -- ISA - - ->STATION

The first precondition, CHECK ME- - - AT - - ->WAVERLEY, returns ABSENT (the others all return PRESENT). Thus the difference between the preconditions and the state is represented precisely by the absence from the data base of

ME- - - AT - - ->WAVERLEY
A new subgoal is now set up to change the current state (i.e. the data base) into one in which ME-- AT-> Waverley is present.

3 Suppose the current goal (or subgoal) is to try to change the current state to make some triple PRESENT. How can we decide which operators are relevant to achieving this goal? Clearly we are interested in operators which add such triples to the data base. That is, we look for operators with similar descriptions in their add lists.

Example
Suppose we are trying to achieve the goal

\[ \text{ME-- AT->Kingscross} \]

Consider the operator table for GOBYTRAIN. In the add list for GOBYTRAIN there is the triple:

\[ X-- AT--Z \]

This triple matches our goal, i.e.

\[ X-- AT--Z \text{ (triple in add list)} \]

\[ \text{ME-- AT->Kingscross} \text{ (current goal)} \]

provided that ME is assigned to |X| and KINGSCROSS is assigned to |Z|

Therefore GOBYTRAIN is a relevant operator.

Notice that this matching is very similar in principle to the pattern-matching you used with SOLO (but STRIPS, unlike SOLO, allows several parts of a triple to be matched simultaneously).

SAQ 16
Apply GOBYTRAIN PIGEONS YORK
NEWCASTLE to the state represented by the following descriptions:

PIGEONS-- AT-> YORK
YORK-- ISA-> STATION
NEWCASTLE-- ISA-> STATION

What is the resulting state (i.e. what triples are present in the final data base)?

SAQ 17
Why cannot GOBYTRAIN PIGEONS SHED
NEWCASTLE be applied to the state represented by the following descriptions?

PIGEONS-- AT-> SHED
YORK-- ISA-> STATION
NEWCASTLE-- ISA-> STATION

Describe the difference between this state and the preconditions necessary to apply the GOBYTRAIN operator. (Notice that GOBYTRAIN PIGEONS SHED NEWCASTLE is not the same as GOBYTRAIN PIGEONS YORK NEWCASTLE).
6.4 Means–ends analysis

We are now in a position to see how STRIPS can use means–ends analysis to form plans. It does it by putting together the steps (3),(2) and (1) described above.

A typical STRIPS goal might be to change the initial state, a data base containing ME---AT---->HOME, into a final state, namely a data base containing instead ME---AT---->KINGS CROSS.

As before, we find in the add list for the GOBYTRAIN operator, the triple [X]---AT---->[Z]. This happens to match perfectly against our goal triple ME---AT---->KINGS CROSS with ME getting assigned to [X] and KINGS CROSS getting assigned to [Z]. Since these triples match, the operator GOBYTRAIN is regarded as being relevant. The general format for this operator is: GOBYTRAIN [X][Y][Z] (i.e. [X] goes by train from some starting location [Y] to some destination [Z]). The variables [X] (the traveller) and [Z] (the destination) have just been assigned values when we matched the triple [X]---AT---->[Z] against ME---AT---->KINGS CROSS above. Thus, we try to apply the operator GOBYTRAIN ME KINGSCROSS. [Y] (the starting location) has unfortunately not been assigned a value yet, but we can leave it unspecified for the time being. We try to apply GOBYTRAIN ME KINGSCROSS, as described in 6.3 (1) above. That is, we first try to satisfy the preconditions, which are:

CHECK ME---AT---->[Y]
CHECK [Y]---ISA--->STATION
CHECK KINGSCROSS---ISA--->STATION

Remember, the current data base consists of

ME---AT---->HOME
WAVERLEY---ISA--->STATION
KINGSCROSS---ISA--->STATION

The third precondition is already PRESENT. We can see from looking at the preconditions that [Y] must be some place where I am, and simultaneously something which is a station (i.e. what we really want is for me to be located at a station, so that I can apply the GOBYTRAIN operator!). However, upon inspection of the data base, we see that where I am is HOME, but HOME is not a station (i.e. ME---AT---->HOME is PRESENT, but HOME---ISA--->STATION is ABSENT). On the other hand, WAVERLEY is indeed a station, but I am not located there (i.e. WAVERLEY---ISA--->STATION is PRESENT, but ME---AT---->WAVERLEY is ABSENT)!

So, what can we do? The strategy employed by STRIPS is to choose one of these locations (HOME or WAVERLEY) to be the value of the variable [Y], thus satisfying one of the preconditions, and simply leave the other one unsatisfied, making this a new difference to be achieved as a brand new subgoal. Thus, supposing STRIPS decides to assign WAVERLEY to [Y], the operator to apply is now GOBYTRAIN ME WAVERLEY KINGSCROSS. The preconditions are now:

CHECK ME---AT---->WAVERLEY
CHECK WAVERLEY---ISA--->STATION
CHECK KINGSCROSS---ISA--->STATION
Now, the last two preconditions are satisfied (both of those triples are PRESENT in the data base), and only the first one is unsatisfied (triple is still ABSENT). So, STRIPS sets up ME--- AT --- > WAVERLEY as a new subgoal to achieve. It tries to find an operator which will end up adding the triple ME--- AT --- > WAVERLEY to the data base.

Thus, the process is now repeated except that the attempt to apply GOBYTRAIN will be rejected as a loop and some other operator (GOBYTAXI?) will be used instead. Eventually an operator will be found which is applicable to the initial state. It will be applied and work will continue on the new current state.

**SAQ 19**

**YOUR ANSWER**

Write an operator table for GOBYTAXI.

*Hint* Use the GOBYTRAIN table on p 44 as a model. |Y| and |Z| do not need to be stations – you can assume that the taxi can go anywhere.

Notice that, in the above example, if HOME had been assigned to |Y|, then the precondition

$\text{CHECK ME--- AT --- > HOME}$

would have been satisfied, but the precondition

$\text{CHECK HOME--- ISA --- > STATION}$

would not have been, and this would have been set up as a subgoal for STRIPS to achieve. It would have come to a dead end, since there is no practicable way of converting HOME into a STATION, and would have had to find some other unsatisfied subgoals to work on by trying to assign to |Y| something other than HOME. We can see intuitively that triples involving the relation AT probably make much easier goals to achieve than triples involving the relation ISA. In other words, it is easier to move things around than to change what they are.

It is possible to augment STRIPS by including information about which relations happen to be associated with very difficult goals. In this way STRIPS can avoid some obvious blind alleys, like trying to achieve HOME--- ISA --- > STATION. This augmentation is provided by a more sophisticated version of STRIPS known as ABSTRIPS.

**READ BODEN’S DISCUSSION** of GPS, STRIPS, and ABSTRIPS, pp 354–60.

### 6.5 The complete plan

In this subsection we will develop a complete plan for the Trafalgar Square example. First we need to develop operator tables for each of the operators needed in the plan. It was suggested earlier that we might need a GOBYTAXI operator to get to the station. For convenience we would like to ensure that there is no temptation to use a taxi for a journey of more than ten miles. To
accomplish this we add

CHECK |Y| -- - NEAR -- - > |Z|  

as a new precondition, where |Y| is the place of departure and |Z| the destination. The GOBYTAXI operator looks like this:

<table>
<thead>
<tr>
<th>operator</th>
<th>preconditions</th>
<th>delete list</th>
<th>add list</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO GOBYTAXI</td>
<td>CHECK</td>
<td>X</td>
<td>-- - AT -- - &gt;</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td></td>
<td>Z</td>
</tr>
</tbody>
</table>

We will need to add information about NEAR into the data base, otherwise the preconditions of GOBYTAXI can never be satisfied. Suitable information is expressed by adding two new triples, as follows:

NOTE HOME -- -- NEAR -- - > WAVERLEY
NOTE KINGSCROSS -- -- NEAR -- - > TRAFALGARSQUARE

We saw on p 48 that ME-- - AT -- - > WAVERLEY remained as the one subgoal yet to be achieved in order for all of the preconditions of GOBYTRAIN ME WAVERLEY KINGSCROSS to be fulfilled. Looking in the add list of the GOBYTAXI operator, we see that the triple

|X| -- - AT -- - > |Z|

matches the subgoal

ME-- - AT -- - > WAVERLEY

and thus the operator GOBYTAXI is regarded as being relevant (GOBYTRAIN is also regarded as being relevant, but it gets rejected since it would result in a loop). When we try to apply GOBYTAXI ME |Y| WAVERLEY (|Y|, the starting location for the taxi ride, has not been assigned a value in this case either) we find that the two preconditions

CHECK ME-- - AT -- - > |Y|

and

CHECK |Y| -- - NEAR -- - > WAVERLEY

are both conveniently fulfilled provided that HOME is assigned to |Y| (i.e. the triples ME-- - AT -- - > HOME and HOME-- - NEAR -- - > WAVERLEY are both PRESENT in the data base). Then, when we apply GOBYTAXI ME HOME WAVERLEY, we remove from the data base everything on the delete list of the GOBYTAXI operator (i.e. FORGET ME-- - AT -- - > HOME) and add everything on its add list (i.e. NOTE ME-- - AT -- - > WAVERLEY). Thus, the subgoal is accomplished, satisfying the precondition of GOBYTRAIN, and we can now apply GOBYTRAIN ME WAVERLEY KINGSCROSS. This GOBYTRAIN operator, which we have focused on from the start, is itself actually just a substep within an overall plan for getting from my house to Trafalgar Square. That substep was found to be necessary by a similar process of reasoning backwards from the original goal of ending up with

ME-- - AT -- - > TRAFALGARSQUARE. The complete plan comprises a sequence of three operators:

1 GOBYTAXI ME HOME WAVERLEY
2 GOBYTRAIN ME WAVERLEY KINGSCROSS
3 GOBYTAXI ME KINGSCROSS TRAFALGARSQUARE

(For the sake of clarity I have left out other possible substeps of the plan, such as PHONETAXI, which could be dealt with in an analogous fashion.) Here is how the data base changes as the three operators are applied. If you look back at the
operator tables, you will see that the changes at each step are precisely those specified in the delete lists and add lists of each operator. We begin with the initial state, represented by the following structures in the data base:

After applying the first operator, GOBYTAXI ME HOME WAVERLEY, the data base looks like this:

After applying the second operator, GOBYTRAIN ME WAVERLEY KINGSCROSS, the data base looks like this:

Finally, after applying the third operator, GOBYTAXI ME KINGSCROSS TRAFALGARSQUARE the data base represents the goal state:
6.6 Learning master plans

Having constructed a plan for a particular task it is sometimes useful to store it away for future use. Not only will this save recomputing the plan if the program is asked to perform an identical task in the future, but it can make feasible the construction of 'superplans', whose operators consist of remembered plans. For instance, a superplan to go round the world might consist of a sequence of remembered plans like ours, to go from home to Trafalgar Square. Such superplans might otherwise be too time-consuming to construct from the basic operators alone.

Of course, it is unlikely that the program will be faced with the exact same task in the exact same circumstances again, so just remembering the sequence of operators which formed the plan would clutter up the computer memory to no good effect. Fortunately, the means-ends analysis technique enables us to assign much more structure to the plan, giving reasons for each step and separating the essential from the contingent decisions. This structure was captured by the Stanford Research Institute team in a (deceptively simple) 'triangle table'. Each triple used in making the plan is placed in a position which shows how it was added and what it is used for. Particular nodes are replaced by variables, so that the plan is applicable to new circumstances, e.g. HOME is replaced by the variable [ORIGIN], ME is replaced by [TRAV] (this is just a mnemonic name to remind us that the variable refers to a traveller). Similarly, we'll use the variable [DEST] for the final destination, and the variables [STAT1] and [STAT2] for the first and second railway stations involved in the plan. Other variables, such as [X], [Y], etc., could also have been used, but long variable names are much more informative for people reading through these tables (though for STRIPS, [ORIGIN] is as meaningless as [Y]).

An example of a triangle table is given below. The three operators involved in the complete plan are arranged just to the right of the main diagonal of the triangle. For each operator, the entire row of boxes to the left of it contains its preconditions, while the entire column below it contains the triples it adds to the data base. The boxes are numbered so we can refer to them in the discussion which follows. Box number 8 has been deliberately left blank for you to fill in for SAQ 20 below.

<table>
<thead>
<tr>
<th></th>
<th>Present in initial state</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[TRAV] ... AT -&gt; [ORIG]</td>
<td>GOBY TAXI [TRAV] [ORIG] [STAT1]</td>
</tr>
<tr>
<td></td>
<td>[ORIG] ... NEAR -&gt; [STAT1]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[STAT1] ... ISA -&gt; STATION</td>
<td>[TRAV] ... AT -&gt; [STAT1]</td>
</tr>
<tr>
<td></td>
<td>[STAT2] ... ISA -&gt; STATION</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>[STAT2] ... NEAR -&gt; [DEST]</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>GOBY TAXI [TRAV] [STAT2] [DEST]</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td>(see SAQ 20)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>[TRAV] ... AT -&gt; [DEST]</td>
</tr>
</tbody>
</table>

51
Thus, the operator GOBYTAXI [TRAV] [ORIG] [STAT1] has as its preconditions everything in box (i) (i.e. the two triples [TRAV] --- AT --- >[ORIG] and [ORIG] --- NEAR --- >[STAT1] must both be PRESENT in the data base). When that particular operator is applied, it ends up adding to the data base everything shown in the column below it, namely the contents of boxes (5), (6), and (7). In this particular case boxes 6 and 7 happen to be empty. This is because the triple added to the data base by GOBYTAXI [TRAV] [ORIG] [STAT1] happens to also be one of the preconditions for the GOBYTRAIN operator, and thus must appear in the row alongside that operator (that is, the row consisting of boxes (2) and (3)). Box (2) contains the other preconditions for the GOBYTRAIN operator. These particular triples, like those in boxes (1) and (3), need to be present in the initial data base, since they are not added in specifically by any of the operators, and that is why they all appear in the left-most column (i.e. they don't appear in the column underneath any of the three operators since they are present before any of those operators are applied).

Incidentally, of those triples in the left-most column, the choice of whether to place them in box (1), (2) or (3) depends entirely on which operator they provide the preconditions for. If they provide the preconditions for GOBYTRAIN [TRAV] [STAT1] [STAT2], then they must appear in the same row as that operator, and thus box (2) would be the only appropriate spot, etc.

Notice also that box (10) contains the results of applying the final operator, GOBYTAXI [TRAV] [STAT2] [DEST]. The adding of the triple [TRAV] --- AT --- >[DEST] into the data base represents the attainment of the final goal.

Thus, as one works through the complete plan (from upper left to lower right), each substep contributes triples to the data base which help to satisfy preconditions for upcoming substeps (in addition to triples which may have already been present in the data base).

SAQ 20

Box (8) has been deliberately left empty. Fill it in, remembering that it represents both the outcome of the GOBYTRAIN operator above it and also a precondition of the GOBYTAXI operator to the right of it (consult the 'operator tables' for GOBYTRAIN and GOBYTAXI on pp 44 and 49 if necessary).

The complete plan in the triangle table above now becomes a new 'master plan' capability of the computer, so it adds a new operator to its existing stock, called say,

TRAVEL [TRAV] [ORIGIN] [STAT1] [STAT2] [DEST]

This kind of operator is called a MACROP (for MACRo OPerator). It is represented not by an operator table but by the above triangle table, and is known to be potentially relevant whenever an ultimate goal of the form [TRAV] --- AT --- >[DEST] (box 10) is sought.

The procedure which applies MACROPs by interpreting these triangle tables is called PLANEX. The obvious way to apply a MACROP is just to apply each of its constituent operators in turn, but PLANEX is more sophisticated than this. It does not assume that each operator is successful in every situation. Instead it takes a neutral view. It starts at the bottom of the table and works up until it finds the first operator which can be applied. It then applies this operator and starts at the bottom of the table again.
This procedure enables STRIPS to behave correctly in the following situations. Suppose the train is diverted and arrives not at King’s Cross, but at Leeds station. The PLANEX procedure will simply apply GOBYTRAIN again, e.g. GOBYTRAIN ME LEEDS KINGSCROSS. Also, if preconditions for middle steps in the plan are already met, PLANEX will start in the middle of the plan rather than blindly starting right at the beginning.

**SAQ 21**

Suppose PLANEX is asked to apply

```
TRAVEL ME HOME WAVERLEY
KINGSCROSS TRAFALGARSQUARE
```

but `ME--- AT---->WAVERLEY` is already present.

Which operators will it apply?

Thus, the triangle table enables us to store plans in a structured way and hence to apply them flexibly, taking account of the current situation. But the structure of the triangle table is based on the means–ends analysis used to construct the plan. In particular, a triple is placed in the table according to what operator was relevant to achieving it and what operator it makes applicable. Thus, if we had used some other method to form the plan (say heuristic search) then we would not have been able to store it as a triangle table and we would not be able to re-use it in this flexible way. This is a good example of successful and intelligent learning being inextricably bound up with successful and intelligent problem solving.

**SAQ 22**

Read about hill climbing again in section 4.4. In what way does the hill climbing learning technique rely on the problem-solving search technique of Samuel’s program?

6.7 **Advantages and disadvantages of STRIPS**

The STRIPS operator tables overcame one of the major disadvantages of the GPS program, namely that a large amount of information had to be supplied with each new problem (e.g. what the differences were; how to extract them; what operators were relevant to achieve them). This information is all implicit in the definition of a STRIPS operator and is extracted by the STRIPS means–ends analysis process.

In STRIPS, differences are always represented by triples (e.g. `ME--- AT----> TRAFALGARSQUARE`) which are not present in the current data base. Reducing a difference always consists of setting up this difference triple as a new subgoal. Finding the relevant operators consists of finding those operators which have a matching triple in their add list. Applying an operator consists of activating the columns of the operator table in sequence. This may involve a recursive attempt to reduce a new difference, namely a precondition of the operator which is not present in the current state.

However, STRIPS has limitations, in particular a lack of flexibility arising from the fact that only simple triples can be included in operator tables, rather than complex interactions between them. For instance, it might be the case that a given
operator has preconditions which change, depending on the surrounding circumstances (e.g. the precondition \texttt{CHECK [Y] \texttt{NEAR} [Z]} for the \texttt{GOBYTAXI} operator might itself change, depending upon how much money the traveller has!). Moreover, STRIPS has no way of specifying a choice of which operator to apply first, if several are found to be relevant (i.e. if several operators all have the appropriate triples in their add list). These limitations have pushed researchers to develop new representational and problem solving tools, as described in the next section.

<table>
<thead>
<tr>
<th>Progress box five</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STRIPS</strong></td>
</tr>
<tr>
<td>Type of problem</td>
</tr>
<tr>
<td>Formalization of problem</td>
</tr>
<tr>
<td>Method of solution</td>
</tr>
<tr>
<td>Advantages</td>
</tr>
<tr>
<td>Disadvantages</td>
</tr>
<tr>
<td>Relevance to humans</td>
</tr>
</tbody>
</table>

7 **THE PROCEDURAL REPRESENTATION OF KNOWLEDGE**

The response to some of the problems mentioned above has been to develop a new range of programming languages, providing tool kits of problem solving procedures. The kinds of tools offered are:

1. One or more data bases, which can contain not only triples (of the kind we have been using in our discussion of STRIPS), but also more complex symbolic descriptions (e.g. one involving a relation among several nodes, rather than just a single relation between two nodes).

2. The ability to \texttt{CHECK}, \texttt{NOTE}, and \texttt{FORGET} in these data bases, and to have the occurrence of such an event automatically 'trigger' the activation of several other procedures (thus allowing these abilities to interact in complex ways).

3. Search control facilities, i.e. mechanisms for making or influencing choices.

4. The ability to have several procedures operate simultaneously, and thus solve
problems where there are mutually interdependent goals which cannot be tackled by the normal hierarchical structuring of procedures and subprocedures.

The tools mentioned in (4) above have been referred to by the people developing them as 'communities of consulting experts'. In this approach, several specialist or 'expert' procedures communicate directly with one another in order to work out a solution to a complicated problem, much as a group of doctors might cooperate to solve a difficult medical diagnosis problem.

To see what use such techniques can be put to read about Fahlman's BUILD program in Boden, pp 363-70.

SAQ 23

Name some of the 'experts' which 'consult' in BUILD to build structures.

The paradigm of 'communities of consulting experts' is useful whenever the task to be done appears to divide into subtasks, but these subtasks cannot be done in sequence, i.e. one after the other. A classic example is in natural language processing. Both syntactic and semantic analyses have to take place, but experience has shown that an unambiguous syntactic representation cannot be obtained without using semantic information. So the analysis cannot be a two-stage one: syntactic analysis followed by semantic analysis. The use of mutually interdependent procedures ('co-routines'), allows us to design a program in which unambiguous parts of a sentence are both syntactically analysed and translated into a semantic representation. The parsing of other parts of the sentence can then use this semantic representation to resolve syntactic ambiguities.

SAQ 24

Think of examples of other problem-solving processes (either conscious or unconscious) which might benefit from the 'communities of consulting experts' paradigm.

Since knowledge (about language, in the above example) is embodied in the way the procedures for dealing with language are designed to interact with one another (as opposed to the knowledge simply being stored passively as triples in a data base), this approach has come to be known as 'the procedural representation of knowledge'. You have already encountered aspects of this approach in your work with SOLO (Units 3-4), and in your reading of the Minsky article in Johnson-Laird and Wason (you were asked to read this for Units 18-19).

The 'procedural representation of knowledge' and 'communities of consulting experts' are not paradigms in the strong sense in which 'heuristic search' and 'means–end analysis' are paradigms. They do not provide a uniform method of guiding search through a tree of possibilities. Rather they offer a kit of programming techniques which can be put together in a variety of ways to make
problem-solving programs. Many workers in AI no longer believe that it is possible to provide a uniform method of guiding search which will be capable of accurate simulation of human problem solving. They argue that humans use specific kinds of knowledge to guide their problem solving and that this cannot be incorporated into a uniform search method, except in a clumsy and artificial manner. Therefore current problem solving programs are custom built for a specific task, e.g. natural language processing, visual perception, or playing with toy bricks. BUILD is an example of such a custom-made problem solver.

In section 5.6 it was stated that there are two ways of trying to solve the limitations of the GPS. One approach (followed by STRIPS) is to try to minimize the amount of information used to describe a particular problem. The second approach, adopted by those favouring the procedural representation of knowledge, accepts that a lengthy problem description is necessary, and indeed makes a virtue of this. The problem setter is to be encouraged to provide as much information as possible, especially about how choices are to be made and in what order different procedures are to be activated. He is to describe this information in the form of actual procedures in the programming language he is using. Rather than describing his problem in a simple standard format (e.g. operator tables) as with STRIPS, he has to understand the particular procedures incorporated in each particular programming language.

To deal with the limitations of their GPS, Newell and Simon have designed a new programming language, called production systems, incorporating symbolic descriptions, pattern matching, search, and a short-term memory model. This language will be described in Unit 28.

8 SUMMARY AND CONCLUSIONS

In this unit we have considered a number of AI problem-solving paradigms and the programs which implement them: exhaustive search, heuristic search, minimaxing, means–ends analysis and STRIPS. We have seen that the paradigm or way of looking at a problem can help to suggest how we search for a solution, and how learning can take place in that situation. We have also seen how to compare a computer program’s performance with that of a human, discussing how he might set about the same task (i.e. the correspondence between the And/Or tree and my Trafalgar Square protocol). Choosing the best paradigm among several which are theoretically adequate is of crucial importance. As discussed in Unit 1, section 4, there are two criteria by which problem-solving programs can be judged: whether they work at all and whether they simulate human behaviour. So a new paradigm has to be judged, not by its intuitive attractiveness, but by its ability to clarify the search control and learning issues necessary to operate when solving a problem. If a program written under the paradigm was intended as a simulation of human problem solving then its psychological validity should be tested in a principled way.

One should, however, be on guard against the temptation to throw over a paradigm on the first sign of trouble – it may be the descriptive terms which are at fault. For instance, in a planning program, some difficulty may be due to a poor choice of particular triples or operators, rather than to the idea of using operator tables or means–ends analysis. We have seen the importance of developing high level descriptions which enable the program to reason at a strategic level. So the choice of descriptive terms is vitally important to successful problem solving.

The importance of AI problem-solving programs for psychology is that they are exploring adequate theories of problem solving and discovering what problems arise that a psychologist might not otherwise be aware of. In the process AI is
extending the notion of mechanism to cover intelligent activities and giving
the psychologist a range of computational processes with which to build
psychological theories. Progress in either field can only be made from a process of
interaction between them.

References


Acknowledgement

Grateful acknowledgement is made to the following for permission to use material in this unit:

The Stanford Research Institute for the photograph on page 43.