Recognition-based Problem Solving

Andrew Howes
MRC Applied Psychology Unit, 15 Chaucer Road, Cambridge, CB2 2EF, United Kingdom.[1]
Andrew.Howes@mrc-apu.cam.ac.uk

Abstract

This paper describes a space of possible models of knowledge-lean human problem solving characterised by the use of recognition knowledge to control search. Recognition-based Problem Solvers (RPS) are contrasted to Soar and ACT-R which tend to use large goal stacks to control search and to situated theories of cognition that tend not to be able to do search at all (e.g. Pengi). It is shown that with appropriate knowledge increments RPS can apply algorithms such as depth-first search with a bounded demand on Working Memory. The discussion then focuses on how some weak methods, such as depth-first search, are more difficult to encode in RPS than others. It is claimed that the difficulty of encoding depth-first reflects human performance.[1]

Introduction

Architectural theories of cognition such as Soar (Newell, 1990) and ACT-R (Anderson, 1993) are based on a classical decomposition between Working Memory (WM) and Long-Term Memory (LTM). As their name suggests Working Memory is used for storing temporary information about the problem solver's immediate situation and goals, whereas Long-Term Memory is used to code persistent knowledge that is used on many successive occasions. In these architectures WM is organised using a *goal-hierarchy*: a stack of goals, each of which is the subgoal of the immediately preceding goal.

Soar's goal-hierarchy is a particularly rich data-structure. Each level may contain information about a problem space, about a state and about many alternative operators. This structure can support powerful computations. For example, it is a trivial exercise to do exhaustive depth-first search of large problem domains. The goal-hierarchy can expand ad infinitum, whilst storing as many intermediate problem states and operator evaluations as are required. The goal-hierarchy and therefore the WM are essentially unbounded. As a result an emergent theoretical problem has been how to equate the computational power provided by this data-structure with the all too evident inability of people to do complex internal problem solving.

One answer to this problem is to propose limitations on the size of WM that supposedly correspond to limitations on human Short-Term Memory (STM) (Anderson, 1983; pages 161-162). That people have limited STM is the standard explanation for the pervasive human inability to methodically search large problem domains. The idea is that the search mechanisms employed by people would work better if they had larger STM, but that they are arbitrarily constrained by technologically (neurally) imposed limits.

An alternative, or at least complementary answer is that STM is *functionally bounded* (Newell, 1990: page 354). The idea is that short-term phenomena might arise just because of how the system must be designed in order to learn, perform and interact. This hope has served to guide the construction of the Soar architecture which has no technologically imposed limits. Here we work with a variation on the theme in which it is hypothesised that much human problem solving may only *need* limited STM. Under this view increasing STM is not required with increasing problem complexity, rather the size of STM is a constant regardless of the problem.

One of the reasons for believing that a functionally bounded WM might be possible is that the external task environment provides many resources for computation. It has long been realised that human problem-solvers are dependent on the environments in which the problem solving takes place (Newell & Simon, 1972). This has recently resurfaced as an issue in the work of many researchers. Problem solving is not something that people do exclusively in their heads, but instead is an activity that involves the combined use of internal and external resources. Models consistent with this approach include Pengi (Agre & Chapman, 1990). However, whilst not requiring much if any WM and whilst exhibiting many interactive phenomena, it is not clear how these models can, even in principle, be applied to solving complex tasks.

1 This work was carried out whilst the author was an exchange visitor at the Department of Psychology, Carnegie Mellon University, Pittsburgh, PA. 15213.

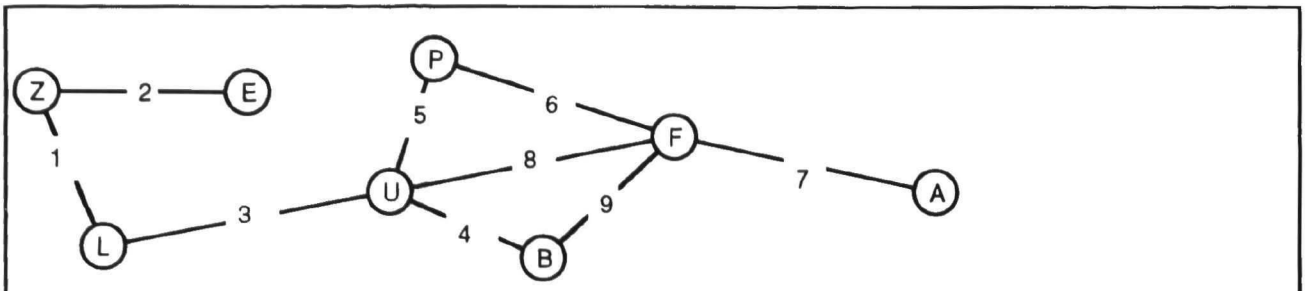

Figure 1: An example graph.

In this paper a framework that defines a class of models called Recognition-based Problem-Solvers (RPS) is proposed. The idea behind an RPS is that the functions that would otherwise be performed by a goal-hierarchy are instead performed by knowledge encoded, during problem solving, in LTM. The basic form of this knowledge is *recognition knowledge*. This is knowledge that adds a 'recognised' marker to an object in WM whenever that object has been in WM on some previous occasion. The use of recognition knowledge is apparent in recent cognitive models (e.g. Aasman & Akyurek, 1992) and it can also be found in the earlier work of Atwood, Masson & Polson (1980).

The description of an RPS is not intended as a specification of a cognitive architecture, but merely as a description of the assumptions of recognition-based problem solving. Its purpose is to facilitate the investigation of encoded algorithms. Which in turn will make it possible to illustrate some of the emergent properties of an RPS. To anticipate the conclusion, the first property is that its use of WM is bounded. This will be demonstrated for a depth-first search algorithm, which in turn will reveal a second property; RPS leads to a more cognitively plausible encoding of depth-first search than does a stack-based problem solver.

The RPS space of models

An RPS can be described in terms of how it uses WM, how it uses LTM and the conditions under which it learns; the *learning conditions*. Learning conditions are a set of rules that determine the acquisition of *chunks*. A chunk is simply a rule that is added to LTM during problem solving. For example the learning condition L_i,

L_i. IF apply(S,O) THEN-LEARN
state(S) & operator(O) → recognised(S,O).

adds a chunk that recognises operators which are actually applied to the state. Once added to LTM the chunk will fire whenever the particular state and operator enter WM.

The rule learnt by L_i is an example of *recognition knowledge*. This can be contrasted to *control knowledge*, which may for example determine whether, in a particular situation, one operator is better than another and to *predictive knowledge* that determines the result of applying a particular operator to the world. The use of either of these types of knowledge is not precluded in an RPS. However, the emphasis here is on the use and role of recognition knowledge.

The initial contents of the LTM is a set of *decision rules* that determine the problem solving method for the task. Each decision rule has a left-hand-side that is a set of conditions in WM and a right-hand-side that proposes some state elaboration. For example, a preference for an operator,

D_j. state(S) & operator(O)
& recognised(S, O) → reject(S,O)

which says, if you recognise that the current operator has been applied to the current state then reject the operator. The problem solving method defined by these rules may be a depth-first search or hill-climbing or some other method.

The RPS's Working Memory includes a description of the current task, of the current state, and of the operators that are applicable to the current state. There are also predicates about what in the current WM has been recognised, and a predicate describing the immediately previous operator. Other predicates may be added by knowledge in LTM but this does not include arbitrary knowledge about previous problem solving history. There is only one task in WM, and only one state. There is no goal-hierarchy.

As with the Soar problem solving architecture the RPS is intend to work using alternate *elaboration* and *decision* phases. In the elaboration phase, WM is updated with the current description of the external world and with knowledge brought from LTM. The elaboration phase runs to quiescence and is then followed by the decision phase which chooses an operator to apply to the world.

In the elaboration phase, rules in LTM may post any number of operator *preferences*. During the decision phase an operator selection is made on the

RPS LEARNING CONDITIONS

L1. IF apply(S,O) & state(S) THEN LEARN [state(S) → recognised(S)]

L2. IF apply(S,O) & state(S) & operator(O) THEN LEARN [state(S) & operator(O) → recognised(S,O)]

L3. IF task(T) & state(T) THEN LEARN [task(T) → recognised(T)]

LEARNING CONDITIONS (DEPTH-FIRST SEARCH)

L4. IF state(S) & not(recognised(S)) & previous-op(P) & operator(U) & undo(P,U)
THEN LEARN [state(S) & operator(U) → backup(U)]

L5. IF task(T) & not(recognised(T)) & state(S) & operator(O) & recognised(S,O) & previous-op(P)
& undo(P,O) THEN LEARN [task(T) & state(S) & operator(O) → reject(O)]

DECISION RULES (DEPTH-FIRST SEARCH)

D1. [task(T) & not(recognised(T)) & state(S) &
operator(O) & not(recognised(S,O)) & not(backup(O)) → acceptable(O)]

D2. [task(T) & not(recognised(T)) & state(S) & recognised(S)
& previous-op(P) & not(backup(P)) & undo(P, U) & operator(U) → best(U)]

D3. [task(T) & not(recognised(T)) & state(S) & recognised(S) & operator(O) & backup(O) → worst(O)]

D4. [task(T) & state(T) → succeeded(T)]

D5. [task(T) & recognised(T) & state(S) & recognised(S)
& operator(O) & recognised(S,O) & not(reject(O)) → acceptable(S,O)]

Figure 2: RPS and knowledge increments required to do depth-first search.

basis of the following preference ordering; *best, acceptable, worst*. Operators with a *reject* preference will not be selected. As soon as the operator is applied the old state is replaced by the new, and all WM elaborations that were dependent on the old state, e.g. the operators are removed from WM.

For the purposes of this paper these components will be sufficient to define an RPS. The important points are that WM is restricted to a description of the current external situation, knowledge is stored in LTM whenever a learning condition is met, and lastly decision rules for the particular task define a search algorithm.

The idea behind separating out the content of the decision rules from the RPS architecture is that it is now possible to derive how different search algorithms (e.g. depth-first search, means-ends analysis and the other weak methods) can make use of the resources provided by recognition chunking. The next section illustrates this with an RPS description of a depth-first search for an undirected graph structure.

Depth-first graph search

Imagine an undirected graph such as that depicted in Figure 1. The structure is an abstraction of a domain in which only one node of the graph is visible to the problem-solver at a time, and transitions are made between nodes by transitions of the represented arcs. How, given a goal node, will an RPS, (a) explore the graph to find the goal, and, (b) acquire control knowledge that will enable it to improve its performance?

A depth-first RPS solution is illustrated in Figure 2. The figure consists of two sets of definitions. The first is a list of the learning conditions (L1 to L5), and the second is a list of the decision rules (D1 to D5).

Learning conditions L1 to L3 define the acquisition of recognition knowledge. L1 learns chunks to recognise visited states. L2 learns chunks to recognise operators that have been applied to states, and L3 learns chunks to recognise a task that has been achieved before. These chunks provide the recognition resources by which the RPS controls its exploration. They are not specific to depth-first search. The remainder of the learning conditions and decision rules are knowledge increments that define a depth-first search algorithm.

These rules use three special predicates that define aspects of the currently available operators. The first of these is the *previous* operator, which is the most recently applied operator and which moved the RPS

into the current state. The previous operator is dynamic and changes every time that a new state is entered. Second, the problem solver needs to know how to *undo* the effect of the previous operator. The undo operator will return it to the immediately previous state. Third, the *backup* operator is used in the management of depth-first search. The backup operator undoes the effect of the operator by which a state was *first* entered. When all other routes have been searched this operator can be applied to return the problem solver to the immediately higher node in the depth-first search tree. There is only ever one backup operator for a particular state and it never changes.

The following paragraphs trace the behaviour of the RPS when applied to the graph in Figure 1. The action of the learning conditions and the decision rules is described. The trace is split into a first trial and a second. The initial state is U and the goal state is A, giving an initial WM of, task(A), state(U), operator(3), operator(4), operator(8), operator(5).

Trial 1. First the RPS enters the elaboration phase and gathers all relevant knowledge from LTM. The problem solver is in state U and no recognition chunks will fire as neither the state, operators nor the task has been seen before. As a result only decision rule D1 (Figure 2) will fire. It is relevant whenever there is a new task and an operator that has not been tried before. In this situation it will give acceptable preferences to operators 3, 5, 8 and 4.

Next the RPS enters the decision phase and a selection is made randomly between the operators with acceptable preferences. Let's say that 4 is selected. Then the predicate, apply(U, 4) is added to WM and learning conditions L1 and L2 will subsequently fire, adding chunks C1 and C2 to LTM,

C1: state(U) → recognised(U)
C2: state(U) & operator(4) → recognised(U,4).

Both of these chunks will be used later on in this trial. For now the RPS will move into state(B), and its WM is updated. This involves replacing state(A) by state(B), then removing all predicates that were derived from state(A) (in this case the operator proposals), and then adding the predicates previous-op(4) and undo(4,4). (In this simple graph an operator is its own inverse).

Once in state(B), WM will be elaborated with a description of the available operators (which are 4 and 9) and then learning condition L4 will add the following chunk to LTM,

C3: state(B) & operator(4) → backup(4).

This chunk will fire immediately and add predicate backup(4) to WM. Again decision rule D1 fires and gives an acceptable preference to operator(9). This time operator(4) does not get an acceptable preference because it is the backup operator. As a result operator(9) is selected and applied moving the problem solver to state(F). In the process the following chunks are formed,

C4: state(B) → recognised(B)
C5: state(B) & operator(9) → recognised(B,9)
C6: state(F) & operator(9) → backup(9)

and the predicates dependent on state(B) are removed from WM (including operator(4), operator(8), backup(4), previous-op(4), and undo(4,4)). Let's imagine that the RPS now selects operator(8) and traverses back to state U, forming the chunks,

C7: state(F) → recognised(F)
C8: state(F) & operator(8) → recognised(F,8)

Now the recognition knowledge learnt earlier comes into play. State U was of course the initial state, and has therefore been visited before. The RPS will know this because chunk C1 will fire and post recognised(U). The problem solver has wandered in a loop, and the appropriate thing to do is return to the previous state. This decision is captured by rule D2, which proposes a best preference for operator(8). Operator(8) will be selected and the RPS will return to state(F).

It is in the return to state(F) that the point of the RPS mechanism can be seen. The detection of the loop and the return to a previous state has been achieved using recognition knowledge in LTM rather than by storing intermediate states in WM.

On returning to state(F), learning condition L5 (see Figure 2) fires. This condition is relevant when the task has not yet been achieved and the RPS has just backed up from a tried operator. Its effect is to create a chunk that rejects the tried operator. In this case the chunk would be,

C9: task(A) & state(F) & operator(8) → reject(8)

The logic of this rule is that immediately following the deliberate return to a visited state, that branch of the search space must have been fully explored (remember that depth-first is an exhaustive algorithm). Now this new chunk C9 fires immediately and puts a reject preference for operator 8 into WM. Decision rule D1 fires for operators 6, 8 and 7 and gives them acceptable preferences. Chunk C6 marks operator 9 as the backup operator, which in turn will be given a worst preference by decision rule D3.

The result of these preferences is to leave a random choice between operators 6 and 7. If the RPS chooses operator 6 then the problem solving and chunking will continue in the same manner as has been described. If on the other hand it chooses operator 7 then it will lead to the goal state. Once there, learning

condition L3 will create a chunk that recognises the task as something that has been achieved,

C10: task(A) → recognised(A)

and decision rule D4 will fire and post success.

Trial 2. On trial 2 a different strategy is possible. In contrast to the first trial the RPS now has some knowledge about how to get to its goal state. There are two parts to the trial 2 strategy. First, rather than D1 giving acceptable preferences to operators that haven't been tried before, decision rule D5 gives acceptable preferences to operators that *have* been tried before. The rational behind this is that if the task has been done before then there is no point in doing anything new. (Assuming that the initial state is the same.)

Second, the chunks formed by learning condition L5, in this case C9, ensure that branches that lead to loops or deadends get rejected.

The combination of the action of decision rule D5 and the chunks learnt by learning condition L5 is such that on trial 2 the RPS will get to its goal without branching or looping. The loop path back to U from F would be eliminated by the chunk C9, and the untried options, operator(U,3), operator(U,5), and operator(F,6) would be eliminated because D5 would not make them acceptable. The same behaviour would be repeated on subsequent trials.

In this example, decision rule D5 continues using recognition knowledge to guide search, whereas chunks formed by L5 add control knowledge to WM. There are other plausible combinations of the use of recognition and control knowledge but unfortunately there is not space to explore them here.

For now we have demonstrated one weak-method by which an RPS can be incremented so as to search graph structures in the external world. The important point is that although the search is exhaustive, and as efficient as can be expected without additional domain specific knowledge, it imposes a bounded demand on WM. This is achieved through the effective acquisition and use of recognition knowledge.

It is worth briefly contrasting this solution to how a goal-hierarchy based problem solver would have achieved this task. If we assume that the task knowledge can be coded internally then in Soar the classical solution would involve creating a subgoal for every decision point. These subgoals would form a hierarchy in WM with one level for each state that had so far been visited and not backed out of. The number of states stored in WM would be equal to the maximum depth of exploration done so far. In our example trial the maximum number of states stored in WM at any one time would have been four (U, B, F and U). In general the *stock* of states in WM would grow with the size of the problem domain being explored. WM use is therefore unbounded.

Other than using unbounded WM a goal-hierarchy solution has two other major differences to an RPS solution. First the resources that it provides do not naturally support loop detection. There is no recognition knowledge and so loops can only be detected if a special function is programmed that matches pairs of items held in WM and determines whether they are the same. In contrast loop detection emerges from the RPS encoding of recognition knowledge. Second, a goal-hierarchy supports effortless back up to previous states, or *backtracking*. Backtracking is simply a matter of popping a goal. In contrast, in RPS backup functions must be explicitly coded. As can be seen from the previous analysis, to encode depth-first search in RPS requires considerably more knowledge than it does with a stack-based problem solver. The difficulty of encoding and maintaining the backup operator may capture the reason for why people are not commonly observed doing depth-first search.

A Depth-first search algorithm similar to the RPS one described here has been investigated in Soar by Aasman & Akyurek (1992). For the purposes of this paper a Prolog program was written to test the algorithm.

Other weak methods in RPS

Laird & Newell (1983) introduce the concept of a Universal Weak Method (UWM) and subsequently define a UWM for Soar. The idea is that rather than having a pre-programmed set of weak methods from which an architecture can select, a weak method emerges as a consequence of the task knowledge and environment. In the same spirit the RPS framework does not commit to a particular weak method. It is possible that many weak methods could be defined that make use of recognition knowledge. Two that are of particular interest from the perspective of modelling human cognition are means-ends analysis and progressive deepening.

Means-Ends Analysis. Atwood, Masson & Polson (1980) describe an empirically validated model that uses a combination of means-ends and recognition knowledge to evaluate operators in Missionaries and Cannibals and other transformation problems. A distinctive feature of the model is that it uses a frequency measure of recognition to help control search.

Progressive deepening. The depth-first search algorithm described in the previous section was for searching a graph structure that was *external* to the problem solver. We can also speculate on what implications RPS has for internal problem solving. Internal problem solving can be achieved in RPS if the results of operator applications (i.e. predictive knowledge) are known. Given such knowledge it

would be possible to construct a new state internally, without actually applying operators to the world. As before this new state would replace the old. Now assuming that recognition knowledge could be chunked for internally visited states it would in principle be possible to do depth-first search internally. This would support the acquisition of plans that consist of internally constructed recognition and control knowledge. The construction of the plan would correspond to trial 1 in section 3 and the application of the plan to trial 2. However for many domains (e.g. chess) internal depth-first search may be prohibitively difficult because of the complexity of defining and chunking the backup operator. Recall that the backup operator caused much of the complexity in Figure 2. If backup is not possible, then neither is depth-first search. Instead the problem solver must explore paths by always moving forward. If a failure state is reached then it must return to the base position and start exploring again. This is progressive deepening.[2]

Whatever the weak method used, the factorisation between what comes with the architecture and what counts as method-specific knowledge is different in RPS to what it is in the UWM for Soar. The RPS (with bounded WM) is far less powerful than the UWM for Soar (with its WM goal-hierarchy and backtracking facility) and it requires correspondingly more knowledge (in the form of decision rules) to define a weak method. The UWM and RPS clearly furnish different predictions for the ease with which a particular weak method can be used by people. Which is correct is an empirical question.

Discussion

An interesting aspect of an RPS is that learning is intrinsic to its problem solving capability. In contrast Soar can solve many problems without learning, and even when learning is done it often occurs as a side-effect of problem solving. But in RPS learning *must* occur to make problem solving possible in the first place. Without learning there would be no way of controlling search. RPS thereby takes the tight architectural integration of learning and problem solving a stage further.

A second consequence of the RPS mechanism is that the burden of explanation for the errors and inefficiencies in human performance is shifted from the limited capacity of WM to the ability of the problem solver to encode sufficient recognition knowledge for the task. For example there may not be distinctive states in the world (e.g. a maze with white walls) or the problem solver's attention may not be focused on the right aspects of its environment. Inadequate recognition knowledge would rapidly disable the RPS's ability to solve problems.

The point of this paper has not been to question the role of goals and subgoals in models of human cognition per se. It has been to question the use of large goal-hierarchies in working memory. As an alternative the use of recognition knowledge was proposed. The argument in favour of it is twofold. First, that with a bounded WM RPS is computationally powerful enough to support exhaustive external search, and second, the possibility that for internal look-ahead, cognitively plausible weak methods, such as progressive deepening, will emerge.

2In fact, people are observed to do *modified* progressive deepening. This involves a one step look-ahead and backup, that would require the RPS working memory to expand to two state descriptions rather than just one.

Acknowledgement

Thank you to Richard Young and Jeff Shrager for many detailed comments.

References

Aasman, J. & Akyurek, A. (1992) Flattening goal hierarchies. In J.A.Michon & A.Akyurek (eds.) *Soar: A Cognitive Architecture in Perspective*, 199-217. Kluwer.

Agre, P.E. & Chapman, D. (1990) What are plans for? In P. Maes (Ed.) *New Architectures for Autonomous Agents: Task-level Decomposition and Emergent Functionality*. MIT Press, Cambridge, Massachusetts.

Anderson, J.R. (1983) The Architecture of Cognition. Cambridge MA: MIT Press.

Anderson, J. (1993) *Rules of the Mind*. Erlbaum. Hillsdale, NY.

Atwood, M.E., Masson, M.E.J. & Polson, P.G. (1980) Further explorations with a process model for water jug problems, *Memory and Cognition, 8* (2), 182-92.

Laird, Rosenbloom, Newell (1984) Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies. Kluwer Academic Press, MA.

Newell, A. (1990) *Unified Theories of Cognition*. Harvard University Press.

Newell & Simon (1972) *Human Problem Solving*. Prentice-Hall, London.