SEARCH METHODS USING HEURISTIC STRATEGIES

Michael Georgeff

Department of Computer Science,
Monash University,
Clayton, Vic, Australia.

Abstract

Real-valued heuristic functions have been extensively used as a means for constraining search In large problem spaces. In this paper we look at an alternative approach, called strategic search, In which heuristic information is expressed as strategies. Strategic search generates a search graph by following some strategy or set of strategies, backtracking to previous choice points when the current strategy falls. We first examine algorithms for performing strategic search using both determlnisltic and nondeterministic strategies. Some examples are given which indicate that strategic search can out-perform standard heuristic search methods. The construction of strategies is also considered, and reans for acquiring strategic information both from analagous problems and from example execution traces are described. Finally, we indicate how meta-level strategies can be used to guide the application of object level strategies, thus providing a hierarchy of strategic information.

1. Introduction

Search for solutions in combinatorially large problem spaces still remains one of the major problems in artificial intelligence. For such problems exhaustive search methods are not feasible, and it is necessary to guide the search process in some way. Most of the early techniques were based on heuristic functions whose value for a given state provided a numeric measure of the promise or otherwise of pursuing the search from this state [10].

However, it soon became apparent that some form of planning had to be Incorporated into such problem solvers if substantial reductions in search effort were to be achieved. This led to the development of a number of schemes for constructing and then executing plans [4, 12, 13]. The basis of these schemes is to construct a global plan to constrain the set of possible paths In the search space, and then to search for a solution in this constrained solution space. The entire plan is constructed from the initial state to the goal state, making use of information associated with the operators to devise the plan.

However, there are many problems where this approach either does not work or is inappropriate. For example, in chess it is not clear how to define the abstract spaces in which the planning can take place, and the variation in environmental influences (i.e. the responses of the other player) makes global plan formation and modification very difficult. In such situations we may wish to construct localized plans that involve conditional operations [17]. In other problem domains, plan knowledge may simply be part of the domain specific knowledge of the problem solver (e.g. tesuji in the game of GO [1], strategies for Rublk's Cube [16] and program transformations [2], and methods of experimental design [5]), These plans are often heuristic in nature——that is, they are intended to advance us towards a goal, but they may take us down a dead-end, or fall by suggesting an illegal or impossible transition.

In this paper we examine some basic search algorithms which use such heuristic plans or strategies rather than numeric valued heuristic functions. That Is, for any given state, the heuristics provide a set of promising strategies or plans for advancing towards the goal. The basis of these algorithms Is quite straightforward -— we pursue promising strategies until either they are no longer applicable or they generate an illegal transition. Then, exactly as in the standard heuristic search methods, we back-up and try some other promising strategy. In the more general case, we may alternatively apply corrective strategies, allow for the dynamic creation of strategies or use meta-level strategies to guide the use of object-level strategies. Such search algorithms will be called strategic search algorithms.

2. Definitions

We define a problem P to be a quadruple

P - <D, Q, ss, Dg>

where D is a setof states called the problem space, Q is a set of partial functions D -->* D called operators, ss in D is the start state, and Dg is a subset of D called the set of goal states.

We say that a state s in D directly generates a state s* in D, denoted s»> s' (assumed to be Indexed by P), if for some operator o in Q we have o(s) « s^{v}. We will also say that <s,s*> is a legal transition in P. We call 8* an (immediate) successor of s, and we call s an (immediate) predecessor of s'. A state s in D generates a state s* in D if s •"> s', where -* is the transitive closure of -*. We call s' a descendant of s, and s an ancestor of s*.

2.1 Strategies

A strategy is simply a program or machine that generates sequences of state transitions. We will first consider deterministic strategies, which for a given state in the domain of the problem generate at most one such sequence.

Consider a problem $P = \langle D, Q, ss, Dg \rangle$. Let S_p be a strategy for P, and assume that, for some state s_0 in D, S_p generates the sequence $s_1, s_2 \ldots s_n$ of states in D. State s_n is called a terminal state and the states s_i, $i = 1..n-1$ are called intermediate states. Then we say s_0 in D generates the states s_i in D, $i = 1..k$, $k <= n$, under the strategy S_p, if $\langle s_{i-1}, s_i \rangle$ are legal transitions in P. Where no ambiguity can arise, we will simply say that s_0 strategically generates the states s_i, $i = 1..k$. If $k < n$, then we say s_k is a failure state, or that S_p fails at s_k. The sequence of states s_i, $i = 0..k$, or some interval thereof, is called a strategic path of S_p.

Note that S is allowed to be partial——that is, it may be that a strategy is not defined for some states in D. Furthermore, no restrictions are placed on the state transitions specified by the strategy. In particular, they need not be legal transitions of P. Neither is there any requirement that the final state of the sequence be a goal state of P, although it would be expected that the strategy at least moved us closer to a goal. Finally, note that it is possible for states in D to have more than one strategic successor ——that is, the strategic successor of an intermediate state may depend on the strategic path to that state.

For example, consider the eight puzzle [10]. A very simple strategy for this problem might be given by

```
procedure strat1 (s)
        while s is not the goal state do
            if the blank is not in the centre then
                swap the blank with the tile
                whose position the blank occupies
            else swap the blank with the firt
                misplaced tile
            end-if
        end while
end strat1.
```

There are three points to note about this strategy. First, the strategy contains conditional and iterative constructs, and in general could have an arbitrarily complex control structure. Second, it is clear that in many cases the strategy will invoke an illegal transition for P. Finally, the strategy is problem specific, as it depends on the goal state of P. In general, the strategy could also depend on other properties of the problem, such as the start state, the operators or some cost function.

2.2 Strategy-first Search

Consider a problem $P = \langle D, Q, ss, Dg \rangle$, and the graph G defined by $G = \langle D, P^* \rangle$. The search problem Is to find a path (possibly of minimum cost) in this graph from the start state ss to some goal state. Generation of all successors of a state is called expanding that state [10]. If, during a search, all successors of a state have been generated, we will say that the state is closed. Otherwise the state is said to be open. If a state s generates more than one successor in the search graph then we call state s (or the node corresponding to s) a choice point.

Given a strategy S_P for P, we will attempt to find a path to a goal state by applying the strategy S_P to the start state ss. However, in most cases we will reach a point in the search where the strategy cannot be followed (either because the strategy terminates without finding a goal or because some Illegal transition is invoked). We might then want to apply the strategy afresh to some of the strategically generated states or to some of their non-strategic successors. We therefore need a selection scheme for determining which choice point to expand next.

Given that strategic moves can be expected to advance the search towards a goal, one possible scheme is to pursue paths containing the least number of non-strategic moves in preference to other possible paths. The search resulting from the use of such selection schemes will be called strategy-first search.

3. The Basic Algorithm

The procedure for performing strategy first search is essentially the same as standard search algorithms [10], except that the selection scheme is based on choosing strategic transitions in preference to non-strategic transitions. The simplicity of this selection scheme, however, has important consequences for the representation of open choice points. In particular, unlike best-first search, selection of the next choice point need not involve a search of the list of open choice points. For example, all strategically generated states can be placed directly on the list representing the open states, where as all those generated by non-strategic transitions can be held temporarily on some other list. Only when all strategic transitions have been exhausted need the held states be opened for selection. Indeed, there is no need to expand held states until they are opened, resulting in further gains in efficiency.

In the following algorithm, the operator . denotes the standard list constructor (LISP cons), hd denotes the head of the list and tl denotes the tail of the list. The expression (x - y) denotes set (more accurately, list) difference and (x + y) denotes set union.

```
function basic (ss)
    open := (list ss);
    held := emptylist;
    closed := emptylist;
    while (not (null open)) do
        st := (select open);
        if (succ st) then return (path st);
        closed := (st . closed);
        stratseq := (strat st);
        open := (stratseq - closed + open);
        held := (st . held);
        if (null open) then
            forall s in held do
                open := ((expand s) - closed + open);
            end-forall;
            held := emptylist;
        end-if;
    end-while;
    return 'failed;
end basic.
```

The function select pops the top element off open; strat returns the list of states generated at st by the strategy S_p ; expand returns the list of states directly generated from the state s: and path returns the (unique) path in the search tree from the start state to state st (in this case the path may simply be taken to be the first path found).

The ordering placed on open states and held states will determine the order in which strategic moves and non-strategic moves, respectively, are generated. One of the simplest schemes is to treat both the open and held lists as stacks, giving depth first generation of both types of move. This scheme is used in the examples that follow.

3.1 Examples

The example to be considered is the 8 puzzle. We will assume that the square is numbered clockwise starting at the top left hand corner, with square 9 in the centre. We will further assume that the goal state is such that the number on each square corresponds with its position and that the blank "tile" is in the centre (i.e. at position 9).

We will consider two strategies represented by the following rules:

(i) If the blank is not in the centre, then swap the blank with the tile whose position the blank occupies; otherwise swap the blank with the first misplaced tile.

(ii) If the blank is not in the centre, swap the blank with the tile whose position the blank occupies; otherwise rotate the 3 tiles in some quarter of the square so that the number of misplaced tiles is reduced, and the blank is returned to the centre.

In order that strategy (ii) be deterministic, the first of the (possibly many) rotations that satisfy the condition is taken as the strategy. For example, in state (2 8 3 A 5 6 7 1 9) the strategy produces a clockwise rotation of the top left hand quarter of the square, giving state (12 34 56 789).

Note that using strategy-first search with depth-first expansion of open states, strategy (1) generates exactly the same search tree as does the strategy strati given in section 2. Note also that strategy (11) Is partial and does not generate a strategy when the blank is In the centre and the number of misplaced tiles cannot be reduced by a quarter square rotation.

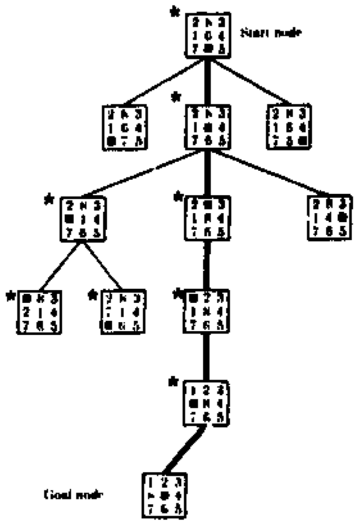

Fig. 1. Tree from strategy (i)

Figure 1 shows the search tree generated by strategy-first search using strategy (i). The states at which the strategy is evaluated are indicated with *. Strategy (ii) generates the solution path directly with two evaluations of the strategy. These search trees may be compared with the search tree generated using the standard heuristic search method with the heuristic function h(n) ■ W(n), where W(n) is the number of misplaced tiles in the state n (see [10]). This search tree finds the same solution path and contains 14 nodes (states). Moreover, the heuristic function is evaluated at every node in the tree.

Thus for both strategies, and particularly strategy (ii), the number of evaluations of the strategy during the search is considerably less than the number of evaluations of the heuristic function. Moreover, evaluation of strategy (1) and evaluation of the above heuristic function are of comparable complexity, and evaluation of strategy (11) can be achieved in about twice the time. Thus, in this case, strategic search is more efficient than heuristic search by a factor of almost two for strategy (i) and over three for strategy (ii). Furthermore, the heuristic search method requires that the entire list of open states be searched either on state insertion or selection — strategy-first search simply pops a stack.

It is important to note that in the above cases it was possible to determine the appropriate strategic transitions without explicitly generating and evaluating different paths in the search space. Thus, for example, under strategy (11) it was not necessary to try all possible (or in fact any) quarter souare rotations in order to determine which to use. If this criterion were not met, then strategic search would be grossly inefficient. Interestingly, this is exactly the kind of criteria imposed on real world strategies——a strategy is not much help if one needs to do a complete search to determine what to do next.

4. Non-deterministic strategies

In many problems the strategy adopted may be non-deterministic in the sense that for a given state in the domain of the strategy a set of strategic paths, rather than a single strategic path, is generated. That is, the strategy is generated by a non-deterministic program or machine. The situation is exactly the same if, instead of a single strategy, we specify s set of strategies that can be applied to each problem state——the set of strategies can be considered to be a single non-deterministic strategy.

The basic strategic search algorithm can be readily generalized to allow of non-deterministic strategies. We simply require that the function strat return the (possibly non-singleton) set of state paths generated by the non-deterministic strategy. Given that we usually process lists rather than sets, it is natural to also allow that the set of state paths generated by strat be ordered with preferred sequences occurring first.

Of course, while this approach will work, considerable effort can be expended in generating states that are not subsequently used. This is not so much of a problem n the standard heuristic search methods as the expense of applying a single operator is usually quite small. However it can be very inefficient in strategic search, where the generation of new states can involve the calculation of long sequences of state transitions. When possible, it is thus preferable to initially generate only the most promising state path while allowing the possibility later in the sesrch of generating more state sequences. In the most general case it may then be best to represent open as s list of co-routines or generators. Each time a non-deterministic branch occurs in the strategy, additional co-routines are sprouted and added to open.

4.1 Example

let us say that a state s_2 is better-ordered than a state s^ If, considering only those tiles around the perimeter of the square, more tiles in s_2 are followed by their proper successor than in s_1. Now consider the following strategy.

If the blank is in the centre, and a rotation of the tiles is some quarter of the square or in some half of the square produces a better-ordered state, then rotate the appropriate tiles. If more than one rotation Is applicable, order the set so that the quarter square rotations occur first. If the tiles are perfectly ordered, but not in place, then rotate the tiles around the entire perimeter so that each moves closer to home.

Figure 2 shows the search tree generated by applying the above strategy to the configuration (2 1 6 8 3 5 7 4 9). Only those states at which the strategy is evaluated are shown. Each arc is labelled with the operator sequence used——the symbol :: represents the four quarters of the square, the arrow specifies the tiles rotated and the direction of rotation.

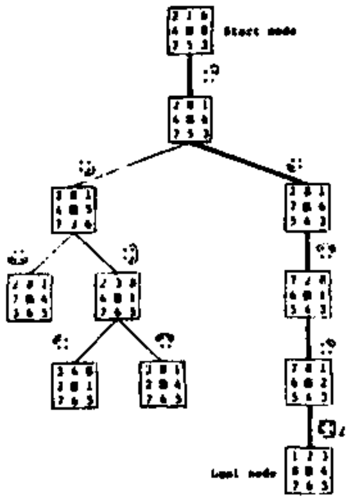

Fig. 2. Tree from strategic search

For comparison, consider the tree produced by the heuristic search algorithm based on the heuristic

h(n) - P(nH3S(n)

where P(n) Is the sum of the distances that each tile Is from home, end S(n) is a sequence score obtained by checking around the perimeter tiles in turn, alloting 2 for every tile not followed by its proper successor and 0 for every other tile, except that a (non-blank) tile in the centre scores 1 (see [10]). This search tree contains 44 nodes and the length of the (optimal) solution path is 18 moves.

Although the path found by the strategic search is considerably longer than that found by the standard heuristic search, it is interesting to note that the strategy is again evaluated for far fewer states than is the heuristic function. Furthermore, the solution found by the strategic search is very similar to the type of solution generated by humans attempting the task——that is, the first priority is to achieve the proper ordering; the perimeter rotation is straightforward and can be done with little processing effort at the last stage.

In fact, the comparison above is a little unfair to the strategic search method as it did not make use of any information on distance from home. If we use the same strategy as above, but order the strategic moves on the basis of the change in distance from home and weighting the half square rotations by +2, then in this case strategic search generates the minimum cost path to the goal state directly——no other path is explored.

Strategies also have other advantages. They tend to be more transparent that parameterized heuristics, local information can be readily incorporated and subsequent modifications can be made easier. In development, it is not necessary to try adjusting the values of a numeric values expression, which, while improving selection on one area may impair it in another. For example, consider that the goal state for the eight puzzle Is changed so that the blank now occurs in the corner. Then we only need modify the above strategy so that once it achieves the "intermediate" goal with the blank tile in the centre, it makes two further moves to get the blank to the appropriate corner. It is far less clear how one would go about modifying the heuristic function.

It is also possible to modify the above algorithms so that they are admissable [10] when the strategy satisfies certain restrictions. The details can be found in [7].

5. Discovering Strategies

Several approaches to constructing heuristic functions have been proposed. Most have attempted to optimise the values of coefficients in an heuristic function so as to maximize overall performance [11, 14]. A potentially more powerful approach has been suggested by Gashnig [6]. Given some problem P, instead of seeking an heuristic directly for P, one seeks instead another problem, P' say, similar to the given problem but easier to solve. In particular, there must exist an algorithm A' for solving P', and the transitions specified by A^1 applied to P* must be capable of being represented by a graph which is an "edge subgraph" or "edge supergraph" of the given problem. Gashnlg then proposes that the value of the heuristic function for some state s in P be simply the number of transitions from s to the goal state under the algorithm A'.

For example, we can use the MAXSORT algorithm for sorting the numbers 1 to 9 to calculate heuristic values for the 8 puzzle. MAXSORT simply swaps the 9 with the element whose proper place in the permutation It occupies, except when the 9 Is in the 9th position, in which case the first misplaced element is swapped. For a given state, the number of swaps to reach the goal is taken as the heuristic value of this state. (Note that except for a small perturbation caused by swapping the 9 into the ninth position, the number of swaps under MAXSORT is the same as the number of misplaced tiles).

The major problem with this approach is that unless an analytic result Is available, the problem P' has to be solved (i.e. the algorithm A* executed) separately for each state In the search graph of P. However, we can use the same approach much more advantageously if we use strategic search rather than heuristic search in solving P. In this case we simply transfer the strategy of P^9 (in essence, the control structure of A') to the problem P. For example, the strategy strati given in section 2 (equivalently, strategy (1) in section 3) is exactly the MAXSORT strategy.

Informally, consider that we are given two problems P - <D, Q, ss, Dg> and P' - <D', Q^1, as', Dg*> and a partial map g from D' to D. Now assume that we have a strategy (program) A' for solving P^f, and that we can construct (appropriate) mappings h- and h from the functions f' and predicates p! occurring in A' to functions f. and predicates p. over the domain D of P. Then we can obtain a strategy A for P by replacing each occurrence of f* in A' by h_f(f!) and each occurrence of p' by h (p!). Of course, there is nothing in this definition that guarantees that A will be a useful strategy for solving P.

For example, consider that we know how to fix leaking taps:

```
procedure leak
  determine whether you need a spanner
  if you do then go to position of spanner
        grasp spanner
        take spanner to tap
        use spanner to fix tap
        release spanner
  else go to tap
       fix tap
  end-1f
end-procedure,
```

This strategy can be transferred to the monkey and bananas problem [10] by associating the functions and predlcatea in the tap world with corresponding one8 in the banana world, giving the following algorithm:

```
procedure bananas
  determine whether you need the box
  if you do then go to poaition of box
        hands on box
        push box to bananas
        climb on box; grasp bananas
        get off box
  else go to bananas
        grasp bananas
  end-if
end-procedure.
```

With this strategy the monkey can solve his problem determlnistlcally. In fact, it is not too difficult to generalize this strategy to one governing the use of most types of tool, viz.

> if you need to use a tool, go to the tool first, then take it to the job and use it.

As far as automating the problem solving process is concerned, the problem of constraining the expansion of the search graph for a given problem has been reduced to finding an analagous problem for which we already have a solution method. This is a more tractable task than that of generating an heuristic function [8,9].

Another approach is to learn the strategy directly from traces of aample solutions generated during some sort of training session. This is a standard inductive Inference problem: find a program that generates successful execution traces for the problem P. Associating with each operator o. in Q and (pre-specified) predicate p. over D a distinct symbol from some alphabet V, then successful execution traces form a language over V. The problem then reduces to a grammatical inference problem, that is, to the construction of a grammar which generates this language. Some interesting work has been done along these lines by Stolfo [16].

6. Meta-level strategies

Strategy-first search is a simple but effective way for constraining search in many problems. However, in more complex problems we may need more sophisticated and possibly problem specific selection schemes. One way to achieve this is to use other strategies for defining the selection scheme. We will call such strategies meta-level strategies, in contrast to the object level strategies such as those discussed above. Thus strategy-first search corresponds to the very simple problem-independent meta-strategy

> Choose the choice point generated by the fewest number of non-strategic transitions.

The use of numeric valued heuristic functions is also a special case of meta-level strategic information, as is the use of mats -level production rules in TEIRESIAS [3]. However, none of these cases involve; sequences of selections. More general meta-level strategies could take account of information -derived during the aearch, and could allow for dynamically changing lines of reasoning.

More abstractly, given a problem P we construct a meta -level problem M over states that represent the progress of the search. A meta-level strategy Is a strategy for M; that Is, a program (possibly non-deterministic) that specifies how the search space should be expanded.

As the meta-level problem M is no different from any other problem, the basic procedure outlined above, or some variation of it, can be used to generate a solution to M, and hence to P. However, in almost all cases we can expect the meta-level problem to be commutative — that is, if for a given state there exists a number of possible successors, and one of these leads to a solution, then so do all the others. In such a situation there is no need to back-up and consider previous states of the computation. In systems that are required to Interact with experts, such uniformity of knowledge representation is an important consideration. Furthermore, with such a scheme it is possible to provide a hierarchy of meta-level problems and strategies, each determining how to handle the non-determinism of the one below it.

References

1. BROWN, D.J.H. (1979) "Hierarchical Reasoning in the Game of GO", Proc IJCAI 6, 114-116.
2. BURSTALL, R.M. and DARLINGTON, J. (1976) "A Transformation System for Developing Recursive Programs" DAI Research Report 19, Uni of Edinburgh
3. DAVIS, R. (1977) "Generalized Procedure Calling and Content-Directed Invocation", SIGPLAN/SIGART Newsletter, August, 45-54.
4. DAWSON, C. and SIKLOSSY, L. (1977) "The Role of Preprocessing in Problem Solving Systems", Proc IJCAI 6, A65-471.
5. FRIEDLAND, P. (1979) "Knowledge-based Experiment Design in Molecular Genetics", Proc IJCAI 6, 285-287.
6. GASHNIG, J. (1979) "A Problem Similarity approach to Devising Heuristics:First Results", Proc IJCAI 301-307.
7. GEORGEFF, M.P. (1981) To appear.
8. McDERMOTT, J. (1979) "Learning to Use Analogies", Proc. IJCAI 6, 568-582.
9. MOLL, R. and ULRICH, J.W. (1979)"The Synthesis of Programs by Analogy", Proc IJCAI 6, 592-594.
10. NILSSON, N.J. (1971) Problem Solving Methods in Artificial Intelligence, McGraw-Hill, N.Y.
11. RENDELL, L. (1977) "A Locally Optimal Solution of the Fifteen Puzzle Produced by an Automatic Evaluation Function Generator", Report CS-77-36, Dept. Comp. Sci., Uni. Waterloo.
12. SACERDOTI, E.D.(1975) "Planning in a Hierarchy of Abstraction Spaces", Artificial Intelligence, 5, 115-135.
13. SACERDOTI, E.D. (1977) A Structure for Plans and Behaviour, Elsevier, N.Y.
14. SAMUEL, A. (1963) "Some Studies in Machine Learning Using the Game of Checkers", in E. Feigenbaum and J. Feldman (ed) Computers and Thought, McGraw-Hill, 71-105.
15. SINGMASTER, D. (1979) Notes on the 'Magic Cube', Polytechnic of the South Bank, London.
16. STOLFO, S.J. and HARRISON, M.C. "Automatic Discovery of Heuristics for Non-deterministic Programs", Proc IJCAI 6, 853-855.
17. WILKINS, D. (1979) "Using Plans in Chess" Proc IJCAI 6, 960-967.