The Best-First Search Algorithm

Implementing Best-First Search

In spite of their limitations, algorithms such as backtrack, hill climbing, and dynamic programming can be used effectively if their evaluation functions are sufficiently informative to avoid local maxima, dead ends, and related anomalies in a search space. In general, however, use or heuristic search requires a more flexible algorithm: this is provided by *best-first search*, where, with a priority queue, recovery from these situations is possible. Like the depth-first and breadth-first search algorithms or Chapter 3, best-first search uses lists to maintain states: open to keep track or the current fringe of the search and closed to record states already visited. An added step in the algorithm orders the states on open according to some heuristic estimate or their "closeness" to a goal. Thus, each iteration or the loop considers the most "promising" state on the open list. The pseudo code for the function best-first search appears below.

```
begin
   open := [Start];                          % initialize
   closed :=[];
   while open ≠ [] do                        % states remain
      begin
         remove the leftmost state from open, call it X;
         if X = goal then return the path from Start to X
         else begin
            generate children of X;
            for each child of X do
            case
               the child is not on open or closed:
            begin
               assign the child a heuristic value;
               add the child to open
            end;
               the child is already on open:
            if the child was reached by a shorter path
            then give the state on open the shorter path
               the child is already on closed:
            if the child was reached by a shorter path then
               begin
                  remove the state from closed;
                  add the child to open
               end;
             end;                            % case
            put X on closed;
            re-order states on open by heuristic merit (best leftmost)
      end;
return FAIL                                  % open is empty
end.
```

At each iteration, best_first_search removes the first element from the open list. If it meets the goal conditions, the algorithm returns the solution path that led to the goal. Note that each state retains ancestor information to determine if it had previously been reached by a shorter path and to allow the algorithm to return the final solution path. (See Section 3.2.3.)

If the first element on open is not a goal, the algorithm applies all matching production rules or operators to generate its descendants. If a child state is already on open or closed, the algorithm checks to make sure that the state records the shorter of the two partial solution paths. Duplicate states are not retained. By updating the ancestor history of nodes on open and closed when they are rediscovered, the algorithm is more likely to find a shorter path to a goal.

best_first_search then applies a heuristic evaluation to the states on open, and the list is sorted according to the heuristic values of those states. This brings the "best" states to the front of open. Note that because these estimates are heuristic in nature, the next state to be examined may be from any level of the state space. When open is maintained as a sorted list, it is often referred to as a *priority queue*.

Figure 4.10 shows a hypothetical state space with heuristic evaluations attached to some of its states. The states with attached evaluations arc those actually generated in best_first_search. The states expanded by the heuristic search algorithm are indicated in bold; note that it does not search all of the space. The goal of best-first search is to find the goal state by looking at as few states as possible; the more *informed* (Section 4.2.3) the heuristic, the fewer states arc processed in finding the goal.

A trace of the execution of best_first_search on this graph appears below. Suppose P is the goal state in the graph of Figure 4.10. Because P is the goal, states along the path to P tend to have low heuristic values. The heuristic is fallible: the state O has a lower value than the goal itself and is examined first. Unlike hill climbing, which does not maintain a priority queue for the selection of "next" states, the algorithm recovers from this error and finds the correct goal.