Chapter 4. Beyond Classical Search

function ONLINE-DFS-AGENT(s') returns an action
inputs: s', a percept that identifies the current state
persistent: result, a table indexed by state and action, initially empty
untired, a table that lists, for each state, the actions not yet tried
unbacktracked, a table that lists, for each state, the backtracks not yet tried
s, a, the previous state and action, initially null

if GOAL-TEST(s') then return stop
if a' is a new state (not in untried) then
result[s,a] <- s'
add s to the front of unbacktracked[s']

if untried[s'] is empty then
if unbacktracked[s'] is empty then return stop
else a <- POP(untried[s'])
else a <- POP(unbacktracked[s'])
s <- a
return a

Figure 4.21 An online search agent that uses depth-first exploration. The agent is applicable only in stale spaces in which every action can be "undone" by some other action.

lists, for each state, the predecessor states to which the agent has not yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

We recommend that the reader trace through the progress of ONLINE-DFS-AGENT when applied to the maze given in Figure 4.19. It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice. For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state. An online variant of iterative deepening solves this problem; for an environment that is a uniform tree, the competitive ratio of such an agent is a small constant.

Because of its method of backtracking, ONLINE-DFS-AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

4.5.3 Online local search

Like depth-first search, hill climbing search has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is already an online search algorithm! Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

Instead of random restarts, one might consider using a random walk to explore the environment. A random walk simply selects at random one of the available actions from the
current state; preference cars be given 11 actions that have not yet been tried. It is easy to prove that a random walk will eventually find a goal or complete its exploration, provided that the space is finite.” On the other hand, the process can be very slow. Figure 4.22 shows an environment in which a random walk will take exponentially many steps to find the goal because, at each step, backward progress is twice as likely as forward progress. The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.

Augmenting hill climbing with memory rather than randomness turns out to be a more effective approach. The basic idea is to store a "current best estimate" $H(s)$ of the cost to reach the goal from each state that has been visited. $H(s)$ starts out being just the heuristic estimate $h(s)$ and is updated as the agent gains experience in the state space. Figure 4.23 shows a simple example in a one-dimensional state space. In (a), the agent seems to be stuck in a flat local minimum at the shaded state. Rather than staying where it is, the agent should follow what seems to be the best path to the goal given the current cost estimates for its neighbors. The estimated cost to reach the goal through a neighbor $a'$ is the cost to get to $s'$ plus the estimated cost to get to a goal from there—that is, $c(s, a, s') + H(s')$. In the example, there are two actions, with estimated costs $1 + 9$ and $1 + 2$, so it seems best to move right. Now, it is clear that the cost estimate of 2 for the shaded state was overly optimistic. Since the best move cost 1 and led to a state that is at least 2 steps from a goal, the shaded state must be at least 3 steps from a goal, so its $H$ should be updated accordingly, as shown in Figure 4.23(b). Continuing this process, the agent will move back and forth twice more, updating $H$ each time and "flattening out" the local minimum until it escapes to the right.

An agent implementing this scheme, which is called learning real-time A (LRTA*), is shown in Figure 4.24. Like ONLINE-DFS-AGENT, it builds a map of the environment in the result table. It updates the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state $a$ are always assumed to lead immediately to the goal with the least possible cost, namely $h(s)$. This optimism under uncertainty encourages the agent to explore new, possibly promising paths.

An LRTA* agent is guaranteed to find a goal in any finite, safely explorable environment. Unlike $A*$, however, it is not complete for infinite state spaces—there are cases where it can be led infinitely astray. It can explore an environment of $n$ states in $O(n)$ steps in the worst case, however, it is not complete for infinite state spaces—there are cases where it can be led infinitely astray. It can explore an environment of $n$ states in $O(n)$ steps in the worst case.