State Spaces and Search Part 2

CS 4710 Course Notes

January 30, 2019

Search Algorithms

Definition. A **search algorithm** is some ordered investigation of a search tree for a goal state. If **correct**, it gives the expected output for any supplied input. If **complete**, it addresses each possible input (though potentially not correctly).

We say a search is **uninformed** if it doesn't use any information about states (or their relations) to produce a shortcut, **informed** otherwise.

Depth-First Search (DFS)

All tree searches are described recursively, starting at some starting state S_1. We then initiate a series of tests and recursive calls on the structure of the tree.

For DFS, consider an arbitrary node n under consideration. Let $child(n)$ refer to the set of nodes m such that $n \to m$ is in our search tree.

(1) First, check $f_t(n)$ is true. If it is, halt and return n.

(2) Otherwise, for each node $c \in child(n)$, return $DFS(c)$. In other words, attempt a DFS search on each child in order.

This procedure produces a "depth-first" traversal of the graph. Since we are calling DFS on each child of n before proceeding to the next in the sequence, we recurse all the way to the maximum depth d of the search tree for each branch under consideration, unless we find a goal state before.

As an illustrative example, consider how to solve a maze with DFS if each branch in the maze (position where one might go in different directions) is viewed as a node in a search tree, with the allowed directions representing transitions between such nodes.

Note: DFS is **not** guaranteed to halt (consider why). When it **does** halt, its worst case performance is linear in the size of the graph. Its worst case storage is the size of the state digraph itself. Since DFS is not guaranteed to halt, it is **not complete**.

Keep in mind you can add elements to a DFS in order to give it better properties. For instance, you can assure that it halts by providing it with a list to record visited states in. (Consider why, and how one might prove this.)

Breadth-First Search (BFS)

In order to implement BFS, we'll need a Queue data structure (call it Q). BFS searches through each set of siblings before moving down the tree's depth. This is accomplished by using Q to order the states we investigate.

Start with $Q = \langle S_0 \rangle$ (containing only our starting state).
(1) Pop Q, call this n.
(2) Test $f_t(n)$. If true, halt and return n.
(3) Otherwise, for each $c \in \mathit{child}(n)$, add c to Q.
(4) Repeat.

Claim: BFS is complete.

Claim: BFS produces the shortest solution (in terms of states/transitions).

Consider how to prove these claims.

Also, note that BFS requires an explicit data structure (Queue) whereas DFS does not require an explicit data structure. The reason why is because of the nature of recursive calls – the order of recursive calls in DFS implicitly forms a Stack data structure. If we were to implement DFS iteratively (such as BFS above), we would have to make explicit use of a Stack.

Heuristic Search

In heuristic search we use an evaluation function (a *heuristic*) to estimate how good a state is (how likely it is to produce a solution). Note we have no guarantee in terms of accuracy.

Definition. Function h is a **heuristic** just in case:
(1) $h(n) \geq 0$ for all nodes n
(2) $h(n) = 0 \Leftrightarrow n \in G$ where $f_t(g)$ is true for all $g \in G \subseteq \Omega$
(3) $h(n) = \infty \Leftrightarrow n$ is a **dead-end** (it cannot lead to a solution)

Definition. We call a heuristic that doesn't overestimate an **admissible** heuristic.
The following are examples of heuristic searches (searches that use Ω and h to short cut):
(1) Best-first search – operates over weighted search graphs (where transitions have costs). The lowest-cost path is expanded and checked for goal states in each iteration.
(2) Hill-climbing – similar to "best-first", but doesn't consider entire paths. Hill-climbing only considers the next step in front of the agent, and selects the best of these single steps as its next move. (Note: hill-climbing is not correct in general. Consider why.)
(3) A^* search (a subtype of best-first search).

A^* Search

Consider the representation of a maze as a set of decision nodes (the branches/crossroads in the maze). Now suppose we can measure the location of each of these crossroads with an x and a y coordinate. In such a case, we can select $h(x)$ as the straight-line distance from x to the goal state (the end of the maze).

Claim: h is admissible.

We'll now introduce how A^* goes. We use function $r(x) = g(x) + h(x)$ where $g(x)$ is the observed cost from S_1 to x, and $h(x)$ is an estimate of the distance from x to the end. In A^* we expand (follow) the path p such that $r(n)$ (for $n \in p$) is minimized.

The algorithm halts when we find a node n such that $f_t(n)$ is true, and for all m, $r(n) \leq g(m)+h(m)$. Note this only works because h is monotone! (Consider why.)

Definition. If h is such that $h(x) \leq c(x, y) + h(y)$ for any nodes x, y and $c(x, y)$ denoting the cost of the transition between x and y, we say h is **monotonic** or, equivalently, **consistent**.

Claim: If h is monotone, A^* search never has to process the same node more than once.