Bidirectional Search

Recall that we have talked about two basic ways in which search could proceed.

Reasoning Forward – start with an initial state and apply operators to hopefully reach a goal state.

Reason Backward – start with a goal state and apply another type of operator which will take you closer to the initial state.

What makes one decide whether to reason forward or backward?

1. Whether or not you have clear/complete definitions of the initial and goal state.
2. Branching factor – one direction or the other might have a smaller branching factor.

But, what happens when:

1. both initial and goal states are unique and completely defined
2. the branching factor is exactly the same in both directions

What one could do is a combination of forward and backward reasoning. The key idea in bidirectional search is to replace a single search graph (which is likely to grow exponentially) by two smaller graphs – one starting from the initial state and one starting from the goal state. The search terminates (roughly) when the two graphs intersect.

So, what is required to do bidirectional search?

1. start node S, and goal node G – these are going to be the two start states for the (independent) graph searches that will be done.
2. S-OPEN and S-CLOSED are lists of unexpanded and expanded nodes, respectively, generated from the start state.
3. G-OPEN and G-CLOSED are lists of expanded and unexpanded nodes, respectively, generated from the goal node.
4. The cost associated with the arc from node n to node x – note that to get this cost appropriately one might need to have two different successor functions. This is apparent in the dog-cat problem – the actual states generated by the two successors functions would be the same (the set of words that one can get by changing one letter at a time in the state and still having a legal word) but the cost would be different. For instance, in the forward direction the cost of going from dog $\rightarrow$ jog would be the cost of replacing a d with a j (i.e., 4). In the backward direction the cost of going from dog $\rightarrow$ jog would be the cost of replacing an j with a d (i.e., 2). This is because, in the end, we want the cost of going from the start node to the goal node.

5. Two heuristic functions – hs and hg – where hs estimates the cost of going to the goal node and hg estimates the cost of going to the initial node.

6. Two sets of properties associated with each node – e.g., S-parent, G-parent which stand for the parent in the forward and the backward search respective.y, gs and gg – which are the costs of the shortest path from start to the node and from the goal to the node (respectively).

The basic algorithm:

1. Start by generating the successors of Start and the successors of Goal (in the forward and backward graph) and installing them on their appropriate OPEN lists (and putting START and GOAL on their appropriate CLOSED lists).

2. Decide whether to go forward or backward. This could be done by alternating or by doing something like deciding to go forward whenever the length of S-OPEN was smaller than the length of G-OPEN and backward otherwise.

3. In either direction (for this example take forward) –

   (a) decide which node to expand in the normal fashion. Call that node n

   (b) if n ϵ G-CLOSED (i.e., the closed list in the other direction) then n is an intersecting point and so we can end the search – notice that this test for intersecting graphs basically replaces the test via the goal predicate used in our normal search algorithm – but we want to pick the best path so:

      - consider the set of nodes on G-CLOSED and intersect this set with the nodes that are either on S-CLOSED or S-OPEN (notice the resulting set is the set of nodes where the two graphs actually intersect). Select from this set the node n with gs(n) + gg(n) at a minimum (where gs is the g value of that node in the forward direction and gg is the g value of that node in the backward direction). Exit with the solution path tracing the path from n back to s and forward to g.

Problems

- Ending conditions quite tricky if we need to guarantee cheapest path cost.

- works very well for a breadth-first type of search. In practice, heuristic searches seem to "miss" each other and search ends up being longer.