INFORMED SEARCH

Announcements

- Read 3.5 (esp 3.5.2) and 3.6 (intro, before 3.6.1)

– Food For Thought –

A few students after Tuesday's lecture told me that BFS and DFS were things that most of you already have seen before in detail and know very well. That is good to hear – but there are some differences between the AI versions and the general tree-search algorithms of the type that you may have seen in a general algorithms class. The differences have to do with the tree (or graph) being known and stored in advance, or being implicit in a set of actions that can be applied to states. Thus a stored tree comes with its own built-in ordering of nodes on a given level, and the nodes are *visited*, not generated; but a tree that is *being generated by* the search algorithm has an order given by how the actions are ordered. That is, to be sure, a smallish difference, but it says a lot about the kind of problem being solved: is it really just a tree/graph to be searched, or is it a problem arising out of some other context where the aim is to find a plan (sequence of actions) leading to a goal?

In this lecture's search topic, we assume Robbie has some specialized knowledge about the state space, allowing "intelligent" decisions as to which action to take at a given state. This called **informed search**, also known as **heuristic** or **best-first** search; the decisions might not always be truly the best, but at least there is reasonable evidence pointing that way.

For instance, Robbie may know that room number X is an even number and that even-numbered rooms are along the east corridor. But a more customary kind of knowledge is this: Robbie has a way to estimate how "close" he is getting, such as might be supplied by a map of the building. Of course, with a map, the room number itself should be visible, and once Robbie has found this on the map, the problem becomes a lot easier.

So we now take a new viewpoint: Robbie has a map, but it is a very large complicated map, and it is stored in a file, so Robbie can't use vision to look for X (not that this is any easier – in fact it is a lot harder to build a vision system!). So Robbie has to search the map file for X, by "choosing" to examine corridors (on the map) to the left, right, straight-ahead, back, downstairs, etc.

If Robbie knows that room X is at least one floor down and one wing over, he can create a lower bound on how far away it is, and use that to help decide where to look next. This sort of idea leads to a famous algorithm, the so-called **A* search** algorithm.

As indicated in the previous lecture, A* can be seen as arising out of one of the uninformed search algorithms: uniform-cost search, which makes use of the cost-function g(n). The idea behind A* is simple: let the next frontier node to explore be the one with the least total estimated cost, based on g(n) plus an estimate h(n) of how much it will cost to get from n to a goal. So now the frontier nodes will be ordered front to back not by increasing values of g(n) but by those of g(n)+h(n).

Of course, to come up with a useful h(n), one needs to know something special about the domain: special information about the states and the actions. That is why A* is an example of so-called **informed** search. The actual algorithm is then already known to us: it is the same general one we already saw before, but with the rule that newly generated child nodes are put on the frontier so as to preserve the least-to-greatest node values of g(n)+h(n). Here h is called the **heuristic function** – the estimate (which might or might not be a good one) of the cost to get from n to a goal node.

How can one possibly find a good h, or even a plausible one? This is actually often quite easy. Consider the 8-puzzle. If we start at a given state, say

1	2	3
4		6
7	5	8

and wish to get to the goal

1	2	3
4	5	6
7	8	

then there is an obvious two-action sequence that does it: move the 5 up and then the 8 to the left. But the available initial actions also include moving the 2 down, the 4 right, and the 6 left. An **uniformed** search algorithm would have these in some order determined in advance by the algorithm's designer, without taking into account any special information about this particular problem. But now suppose we have an actual cost for each action, that combine (add) to give total costs of taking any sequence of actions. Assume in fact that the cost is just the number of individual tile-moves involved in getting to a goal state.

Moreover, we want h(n) to be something we can calculate only from information already available in the portion of the tree that has been generated so far – i.e., without having to look ahead to future moves and nodes. Of course, looking ahead is not forbidden, but that is more complicated, and not in the spirit of search based on growing the tree as we decide where to look.

What general properties of the 8-puzzle can be exploited to make 5-up the best apparent choice, *without* our considering the following move (8-left)? Why is moving the 5 up a good idea in general, if it is where it is (in the center)? One answer is: because the goal has the 5 lower down than that. That is, we can reasonably suppose that it is in general plausible that *moving a tile closer to where it belongs in the goal* is a good thing to try. So one possible heuristic function – let's call it h1 – for this particular domain (the 8 puzzle) could be

h1(n) = the number of tiles that are out of place in state n (compared to the goal)

Another is

h2(n) = the total "manhattan distance" from n to the goal, where the *manhattan distance is the sum of the vertical and horizontal differences between where each tile is in n and where it is in the goal*. (This then is a lower bound on the amount of moving the tiles must do to get to a goal state. Indeed, h1 is also a lower bound.)

Of course, this can backfire. Sometimes, to get to a goal, one has to move one thing far out of the way for awhile in order to first get something else where it belongs. We can see this easily in a blocks-world setting:

Start **Goal**

where C is already where it belongs (on A) but we have to move it off for a bit in order to get A on B, and then we can put C back on A.

This is why we refer to *heuristic* search: it is not guaranteed to do the right thing at all times, but at least it seems to stand a good chance of being better than blind (uninformed) search.

A homework problem will have you hand-simulate A* for the 8-puzzle. What we want to emphasize here is the following wonderful result:

Theorem *(A* optimality and completeness)*: Suppose for every node n and every child m of n, the a function h is such that h(n) <= cost(n,m) + h(m), where cost(n,m) means the actual cost of the action to go from n to m. Then A* is complete and optimal using that h (with g added in to give g(n)+h(n) for ordering the frontier).

Such an h (satisfying the inequality in the theorem) is called **monotonic**. We will not prove the theorem (the textbook discusses details). But we can easily see that coming up with a monotonic h need not be hard at all. For instance, both h1 and h2 above are so. Why is this? Consider h1, for instance: h1(n) is the number of tiles out of place in the state associated with node n. But clearly the cost(n,m)=1, since m is a child of n (one move away); and clearly the number of tiles out of place in m is either the same as for n (if the moved tile is still out of place in m) or is 1 less; in either case the inequality holds. A similar argument works for h2.

The theorem is really good news! It says not only will A* work (get us to a goal, if there is a path to a goal at all) but it will get us the cheapest path to a goal (if there is more than one).

One can always of course just use h(n)=0 for all nodes n; but then A* is the same as uniform-cost search, and there is no real informed search going on.

– Food For Thought –

Is there a way to automate the choice of h, so an agent can do this on its own (as a human would)? Well, a truly general automation of the selection of a useful h is not currently available; but in specific cases there often are things that can be automated, by the agent just comparing a given node's state to a goal state and calculating some difference between them. What do you think might be a fairly general way to try to do this?

Let's look at the above blocks-world example, in some detail, where we take the cost of each move to be 1. What might be a good h here? One choice is h = *the number of blocks that are on the table but shouldn't be, divided by two, plus the number of blocks that are on a block they shouldn't be on (when compared to the goal)*.

Thought problems: (i) is this h monotonic? (ii) can you think of a reason why favoring blocks misplaced on the table over blocks misplaced on other blocks might be a good idea? (ii) is this something an automated agent is likely to be able to come up with?

Let's proceed to see how this h works out for our blocks-world example. To avoid the nuisance of writing fractions, we'll use twice h instead, and we'll also double the g-values so everything stays in the right proportion. (Does this affect the applicability of the A* theorem?)

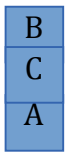
In the start-state, there are three possible actions: C to table, C to B, and B onto C, with h-values of their resulting nodes as 2, 3, and 3, resp. (Why?) And the doubled g-values of these are 2 each (why?). Also the start-node has h-value 1 and g-value 0. (So each possible action applicable at the start actually increases

the sum g+h, again illustrating what we said earlier: sometimes things must temporarily get "worse" so they can then get better.)

Let's start the search: the start-node is taken off the frontier and its children put on. It'll make the simulation a little easier if we don't always bother to list the frontier elements in any particular order now, since it is their g+h values that determine which one really goes in the front. We'll just write the correct g+h under each node.

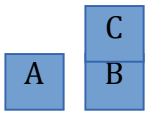
The frontier now looks like this:

1



g=2
h=3

2



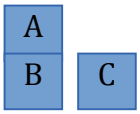
g=2
h=3

3

g=2
h=2

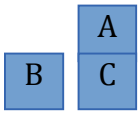
We pop the node off the "front" (which is configuration **3**, by g+h value). The frontier now has nodes **1**, **2**, **3a**, **3b**, **3c**, and **3d** on it. The relevant (not yet seen) children of that node are

3a



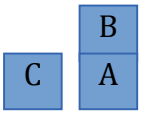
g=4
h=1

3b



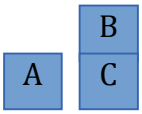
g=4
h=3

3c



g=4
h=4

3d



g=4
h=4

so they go on the frontier along with the two still there. Nodes **1**, **2**, and **3a** have a g+h value of 5, so we have to have some sort of tie-breaker. Let's assume that the tie-break rule is that if the tied nodes have unique h values, we choose the one with the lower h value first. So **3a** is popped; its only not-yet-seen child is:

3aa

g=6
h=0

The frontier now consists of **1**, **2**, **3b**, **3c**, **3d**, and **3aa**. The lowest g+h value is now 5 for **1** and **2**. **1** results in no children that haven't already been seen. **2** results in one child that hasn't already been seen:

2a

g=4
h=4

Now we look back at the frontier which consists of **3b**, **3c**, **3d**, **3aa**, and **2a**. Node **3aa** has the lowest g+h value so it is popped off the frontier and is found to be the goal.

– Food For Thought –

Ok, we have seen lots of ideas already. But for the most part, the role of an active automated *agent* has been rather thin at best. And, for that matter, so has the role of any sort of *reasoning*. Let's pause a bit to think about the blocks-world again, and how a human might approach such problems.

As mentioned earlier, we have a large fund of experience with blocks and other real-world objects, in terms of what we can do with them – or what is sometimes referred to as their *affordances: what opportunities they afford us*. A block can be lifted, dropped, placed, slid. A tower of blocks can topple. Towers are built/supported from the bottom up. When one block moves, blocks under it usually do not, but ones above it usually do. And so on. It is hard to move a block with another block on it. Thus putting all blocks down on the table can be a good thing, as a preamble to then starting to build a tower. That is not to say it is optimal in all cases, but it is not a bad approach.

We also have some ability at visualizing the result of potential moves, even short sequences of moves. This is akin to mentally exploring the search-tree down a few levels, perhaps akin to a shallow version of iterative deepening, but adjusted by our large background knowledge as to what are likely good general sorts of moves. We can also reason back from the goal to potential states that easily lead to the goal, then back from those to yet earlier states, etc. This is variously called means-ends reasoning, or meet-in-the-middle, or bidirectional search.

What all this adds up to is that there is a rich terrain of directions to consider, many with agent-like and/or reasoning-like aspects, including what Hector Levesque – one of the leading lights of current AI research – has called *vivid knowledge*. (In a few more weeks I plan assign a reading by Levesque.)

For the most part we have finished our study of search as a topic in its own right. One more lecture will be devoted to a few specialized search topics (game search – aka adversarial search – and constraint-satisfaction search) and then we will move on to more agent-centric themes (in which search plays an important but often out-of-sight role).