Searching State Spaces

1 A General Search Algorithm

How do we search for paths in our implicitly-specified graph (state space)?

We have some notion of the *current state* — the one we're currently looking at. At the start of the search, the current state is the one associated with the start state.

We check whether the current state is a goal state (whether it satisfies the goal condition). If it does then, assuming we're looking for only one solution path, we can stop.

If the current state is not a goal state, we *expand* the current state. What this means is that we apply operators to this state to *generate* its *successor* states.

While there might be no successor states (a dead-end) or just one successor state, in general there will be multiple successors. The essence of search is to choose one state for further exploration (it becomes the new current state) and to put the others somewhere in case we want to come back to them, e.g. if the chosen one does not lead to a solution.

The data structure in which we keep states that have not yet been explored is called an *agenda*.

Given that there are multiple states waiting on the agenda, yet to be explored, the policy which determines which state to explore next is called the *search strategy* (or *control strategy*).

Here's the algorithm in pseudocode:

```
insert start state onto agenda;
while agenda is not empty
{   currentState := remove from front of agenda
    if currentState satisfies goal test
    {   return the path of actions that led to currentState;
    }
    else {  successors := states that result from expanding currenState;
            insert successors onto agenda;
    }
}
return fail;
```

Different search strategies result from different implementations of the line that I have underlined.

2 Search Trees

One way to think about the search algorithm is that it is making explicit parts of the implicitly-specified state space: the nodes it actually visits and the edges it actually traverses. The parts of the state space that the search algorithms visits can be shown in the form of a tree, called the *search tree*.

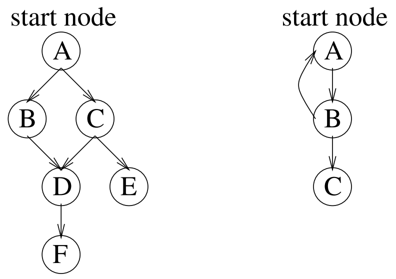
It's important to distinguish the state space from the search tree. The state space is all the states reachable by sequences of actions from the start state. The search tree is different because:

- Some search strategies may leave parts of the state space unexplored. In other words, there may be nodes and edges in the state space that never get visited and so do not appear in the search tree. This, of course, is a 'good thing': it improves efficiency, although, if we skimp too much, we may end up missing the solution paths, which, in general, is a 'bad thing'.
- Some search strategies may re-explore parts of the state space. This can happen when two or more paths in the state space lead to the same node. Unless steps are taken to deal with this, then some nodes in the state space may get visited and expanded more than once, and so they appear in the search tree more than once. This is, in general, a 'bad thing', although the cost of eliminating it can be high. (A common special case of this is when the state space is cyclic. Unless steps are taken to deal with cycles, the search tree may then be infinite.)

Let's explore the second bullet point in more detail.

3 Avoiding Re-exploration

Strictly, we cannot draw search trees unless we know what the search strategy is. But, to illustrate the second bullet point above, without getting too bogged down in wondering what the exact search strategy is (i.e. how to decide what to visit next), in the lecture we will draw possible search trees for the following two state spaces:



To avoid re-exploration of parts of the state space, we must be more selective about which states we add to the agenda. Some of the successors of the current state should be discarded. There are various ways of deciding which to discard, and they vary in how effective they are in avoiding re-exploration and in how much time & space they cost us. (Sometimes it might be better to allow some re-exploration, rather than pay the price of eliminating it).

Three options for avoiding (some or all) re-exploration are common:

- Discard any successor that is the same as the current node's parent.
- Discard any successor that is the same as another node on that path.
- Discard any successor if it is the same as any previously generated node.

Class Exercise. *How effective at avoiding re-exploration are these three options? What do they cost in time & space?*

[Advanced point. In fact, I'm oversimplifying this third option. (*Ignore this if it makes no sense.*) From the above, you might assume that if the newly-generated node is the same as a previously-generated node, we always throw away the new node. *But, this is not correct.* If the cost of the path to the new node is less than the cost of the path to the previously-visited node, then we should not throw away the new node (because the path to it is cheaper.) To handle all of this properly makes the algorithm so complicated and its costs (both space and time) so much higher that this option is hardly ever implemented. If you want to read a proper description of such an algorithm, consult a textbook such as Nils Nilsson: *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann, 1998.]

4 Search Strategies

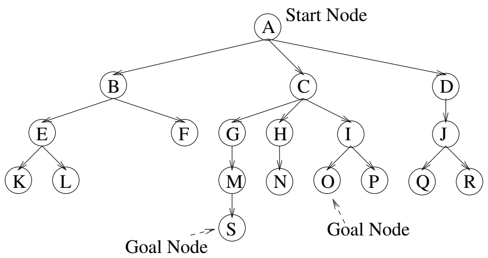
The search strategy is responsible for deciding which node to expand next. Search strategies fall into two classes.

In *uninformed search* (also called *undirected search* and *blind search*), the strategy has no problem-specific knowledge that would allow it to prefer to expand one node over another. It can distinguish goal states from other states, and it will know the length of the path or the cost of the path from the start state to each state on the agenda. But it will know nothing about the probable length or cost of extending a path so that it leads to a goal state.

In *informed search* (also called *directed search* and *heuristic search*), we make available to the strategy some problem-specific knowledge about the likely length or cost of the paths from each state on the agenda to a goal state.

It turns out that we can use our general search algorithm but easily implement different search strategies simply by altering the way the agenda works. Our algorithm always expands the node that is on the front of the agenda. But we have not yet said where, when it is adding nodes to the agenda, it adds them. By changing this, we get our different strategies.

We'll be illustrating the two strategies in the lecture on the following state space:



In some ways, this isn't a very standard state space. First, I've shown it explicitly, rather than specifying a start state and some operators. Second, to keep the lecture simple, I've not allowed more than one path to each node. This means we don't have to think about avoiding re-exploration: the searches will all be finite. But remember, this is the state space; it's not the search tree (which is what we'll show in the lecture).

4.1 Uninformed Search: Breadth-First Search

In *breadth-first search*, we treat the agenda as a *queue*. Nodes come off the front (as always), and new nodes are added to the back. This means that all nodes at depth i in the search tree are expanded before any at $i + 1$. (All paths of length 1 are considered before any of length 2, and all of length 2 are considered before any of length 3, and so on.)

4.2 Uninformed Search: Depth-First Search

In *depth-first search*, we treat the agenda as a *stack*. Nodes are popped from the front of the agenda (as always), and new nodes are pushed onto the front of the agenda. This means that the strategy always chooses to expand one of the nodes that is at the deepest level of the search tree. It only expands nodes on the agenda that are at a shallower level if the search has hit a dead-end at the deepest level. (A path is expanded as much as possible —until it reaches a goal state or can be expanded no more— prior to extending other paths.)

5 Evaluating a Search Strategy

We're going to look at several search strategies. We need some criteria in terms of which we can compare them. The criteria used in AI are these:

Completeness: A search strategy is complete if it guarantees to find a solution when there is one.
Note that is a somewhat one-sided definition. It doesn't impose any requirement on the strategy in the case where there is no solution. In these circumstances, maybe the strategy will say 'There's no solution', or maybe it will run forever. All that matters for completeness is, if there is at least one solution, then it gets found.

Optimality: A search strategy is optimal if it guarantees that it will find the highest-quality solution.
What we mean here is that the first solution path it finds must be the highest-quality one. We're not entertaining the idea that it finds a solution, and then continues to search for other solutions so that it can choose the best of them afterwards.
Note that an algorithm cannot be optimal if it isn't complete. Or, if you prefer, optimality implies completeness.
The notion of 'highest-quality' here concerns the path cost. Recall that we have a function g, the path cost function, which sums the costs of the actions along a path. We'll be writing $g(n)$ to mean the cost of the path from the start state to state n. For optimality, we want the strategy to find the cheapest solution path. (In the case where the actions haven't been assigned any costs, then we want to find the shortest path — this, of course, is equivalent to treating all actions as having uniform cost, and $g(n)$ is then just the length of the path to n.)

Time complexity: How long does it take to find a solution? We'll generally report worst-case results, but best-case and average-case are also of interest.

Space complexity: How much memory is needed to perform the search worst-, best- and average-case)?

Exercise (Part of past exam question)

A farmer, a wolf, a goat and a sack of cabbages are on the left bank of a river. There is a boat on the left side of the river too. It must be crewed by the farmer, and has room for only one of the other three. At no point can the farmer leave the wolf and goat together unattended. Similarly, the goat and the cabbages cannot be left together unattended. The goal is to ferry all four across to the right bank.

One (analogical) problem representation is to represent the farmer, wolf, goat, cabbages and boat as variables, F, W, G, C and B respectively, and then to represent states by two sets: those items on the left bank, and those on the right. Therefore, the start state is $\{FWGCB\} - \{\}$ and the goal state is $\{\} - \{FWGCB\}$. There are 8 operators: one for moving farmer and wolf from left bank to right, another for moving them back, two more for moving farmer and goat, two more for moving farmer and cabbages, and two for moving the farmer unaccompanied.

1. Draw the *state space*.

2. Your state space is to be searched using an agenda-based search algorithm. However, when the current node is expanded, if a successor is the same as any node already visited on that *path*, it is discarded.

   Hence, draw the *search trees* that are built by *depth-first* search and *breadth-first* search. (Multiple answers are possible. You need give only one such answer in each case.)

3. The algorithm is changed so that a successor is discarded if it is the same as *any* previously visited node.

   Would the search tree for *depth-first* search be any different from the one you gave above. If so, draw it.

   Would the search tree for *breadth-first* search be any different from the one you gave above. If so, draw it.