UNINFORMED SEARCH

Announcements

- Reading
  - Section 3.4, especially 3.4.1, 3.4.2, 3.4.3, 3.4.5

Robbie has no idea where room X is, and may have little choice but to try going down this corridor and that. On the other hand, if she has a map of the building layout, with room numbers indicated, the task may be a bit simpler. Instead of actually moving about, Robbie can first search the map to see where room X is and what paths might lead there from her present location. That will surely save time, since traveling virtually on a map is a lot faster than actually moving about in the physical world. Still, the map won't show certain things such as possible temporary impediments like 50 students streaming out of classrooms, slowing progress when she finally tries out a plan based on the map.

But let's ignore such worries about missing information in the map for now. In that case, searching the map is much like searching in the real world. It may not seem so, since our eyes can pick out map features at a glance, without having to move along from one location to the next. But a computer program is not so lucky; it has no eyes, only a file to navigate in by following pointers or links – unless of course we first solve the computer vision problem! (And even then, such a solution itself will have to do lots of searching behind the scenes.)

So, whether by map or in the real world, Robbie needs to search for room X along step-by-step edges in the search-space. But there are important distinctions to be made in how this can go. If Robbie has some general knowledge about room layouts in buildings, such as the consecutive rooms tend to have consecutive (or at least fairly close) numbers, that can help a lot.

Let's start with the assumption that Robbie has *no* special information about rooms and buildings. Then she has nothing to guide her choices in where to look first, second, etc. This is called **blind** or **uninformed** search, and it has been well-studied, giving rise to a number of powerful algorithms that exploit the structure of the search tree and that all take on a similar form, with one key factor that varies from one to another.

Basic AI Search Paradigm:

- Given
  - Possible States.
  - Start State.
  - Allowed actions per state.
  - Goal State(s).
- Then a search tree is automatically defined.

By *automatically defined*, we mean that this information is enough to uniquely identify a search tree. The actions allowed for each state and the start state are used to define possible states.

Basic Algorithm
Start at the root node and **explore** it: check if it is a *goal node* and if not then **expand** it by methodically **generating** all children of that starting node – new nodes one can get to by applying all available actions to it one time each. These newly *generated nodes thus are not yet explored* and are at a deeper level than their parent. We call nodes that have been generated but not yet explored the **frontier** set. Now we repeat the above over and over:

1. Choose one frontier node and remove it from the frontier (it is now being explored).
2. If its state is a goal state then we are done; else expand it by generating its children and adding them to the frontier.

We repeat the above two steps until the exploration of a node reveals that it's state is a goal state (which we return along with the path that led to it) or until the frontier is empty (in which case we return *failure*); or else the process goes on forever.

And that's it! The only thing we left unspecified is how to choose one node from the frontier node: for instance, it could be the one of the nodes most recently added to the frontier (that would be one of the deepest-level frontier nodes – why?); or instead one of the oldest ones (one of the shallowest – why?); or chosen at random; or perhaps chosen based on some knowledge making it seem a better choice for getting to closer to the goal. This last option – based on special knowledge – is so-called **informed** search, that we will look at next time.

How do we represent the frontier?
It will help to think about this if we make a deliberate choice of data-structure for the frontier. One useful choice is that of a queue, where removal (popping) of an item is always from the front (that is, the queue is an *ordered list*, with a front and a back). There are different kinds of queues, depending on whether *insertion* of a new element is also to the front (a **LIFO** queue, aka a stack); or to the back (**FIFO**); or according to a numerical ranking (**priority**) that is computed for each element, from highest (front) to lowest (back). A priority queue is used in an informed search, as there has to be some knowledge in order to determine priorities.

In the LIFO and FIFO cases, if a parent node has more than one child, there is also an issue as to the order of their generation, i.e., the order in which available actions are considered at a given parent node(since that determines the order of frontier insertion of the children). We assume that whatever implementation is used will have its own way of determining this. Be aware that a tree (or a graph more generally) does not really have left and right aspects; it is just that drawings force us to introduce them. But – as long as

we don't misunderstand it – we can suppose that the left-right order (at a given level) corresponds to the order in which an implementation happens to generate those child nodes.

Breadth-First Search

Let's consider the case FIFO case, in which case the search is called **breadth-first**, or **BFS**, for reasons that will become clear. Suppose below we have a portion of the search tree for Robbie's problem of finding room X, from a start location S. A node's children are the ones at the next level shown close below it; some nodes do not have children, some have three, some two, some just one. Thus for instance S's children are abc; b's are ghi; j's is only o; k and l have none; and o's are s and t; and there are three nodes (indicated by an X at levels 3, 4, and 5) that have the goal X as their world state. (Recall that a node is not just a state but also a record of a traversal to that state from a parent node via an action.)

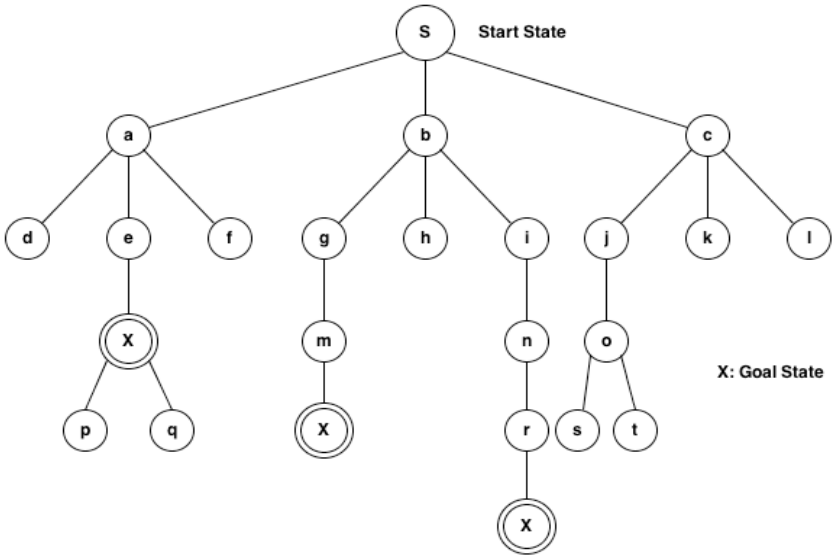

Figure 1

So there are at least three ways to get to an X node from S, at three levels. (While the letters simply indicate nodes, we can pretend they are room numbers if we like.) Initially, the frontier has only the start-node S; and it is immediately removed and explored: it is checked to see if it is X (it is not) and then its children are generated (by applying actions to it – at this point the search tree does not yet look like the above since only nodes a, b, and c – S's children – have been generated). And just to be definite, we'll assume left-right order: a goes on first, then b, then c; so when we have finished expanding the start node, the frontier consists of the list abc, with a at the front since it was generated first and this is a FIFO queue. Then we take a off and explore it, generating its children (since it is not a goal), and putting them at the back of the frontier which now is: bcdef.

Note that the frontier list is stored somewhere in the processing, it is not shown explicitly in the tree; but one can envision it as an evolving fringe of outermost nodes, waiting for exploration. In the FIFO case, the frontier nodes always lie on either one level or two consecutive levels (why?).

We then remove/explore b, then c, and so on, with new nodes continuing to go on the back and older ones coming off the front. Thus what we are doing, in effect, is "walking" through the tree from left to right on each level, checking for the goal and adding children for all nodes at that level before going on to check nodes at the next level. Thus the name "breadth-first".

And in our example, what will happen? Eventually the search starts to explore frontier nodes that are at level three, and the X there will be the first such node explored, so the algorithm stops when it is found to contain a goal state. (Recall that search-tree nodes are more than just states; they contain information about parent and action that they came from as well.) Thus BFS will not even generate as much of the tree as is shown above. (How much will it generate?) Here is the changing frontier as we go through the loop over and over; the newly added frontier nodes are in *blue*. The nodes in *red* are those to be removed. We can see in line 2, that a is *purple*, this is because it was both newly added (blue) and was the node to be removed from the frontier (red).

	Front										**Back**	
1	S											
2	a	b	c									
3	b	c	d	e	f							
4	c	d	e	f	g	h	i					
5	d	e	f	g	h	i	j	k	l			
6	e	f	g	h	i	j	k	l				
7												

	f	g	h	i	j	k	l	X				
8	g	h	i	j	k	l	X					
9	h	i	j	k	l	X	m					
10	i	j	k	l	X	m						
11	j	k	l	X	m	n						
12	k	l	X	m	n	o						
13	l	X	m	n	o							
14	X	m	n	o								

Figure 2

Nothing is added in lines 6, 7, 10, 13, and 14. The goal state X is found in step 14 when it is taken off the front of the queue and explored.

Exercise: what would have happened if an implementation had the opposite left-right order (i.e., actions were considered in the opposite order), so that child nodes were generated c b a, f e d, etc.?

Depth-First-Search

Now let's consider changing the rule about frontier insertion, treating the frontier as a LIFO queue – aka a stack – with newly-generated nodes going to the front (again generated in left-right order based on the tree drawing).

What happens in this case? Initially it's much the same, but with minor-seeming changes: on exploring the start-node S, we'll get the frontier cba (a goes on, then b, then c, each added to the front). So now it is node c that is removed and explored, which puts its children on the frontier, giving us l k j b a; and then l is explored, and the frontier becomes k j b a since l has no children; and then j b a; and then (as before underlining the newly inserted frontier nodes)

	Front										**Back**
1	S										

2

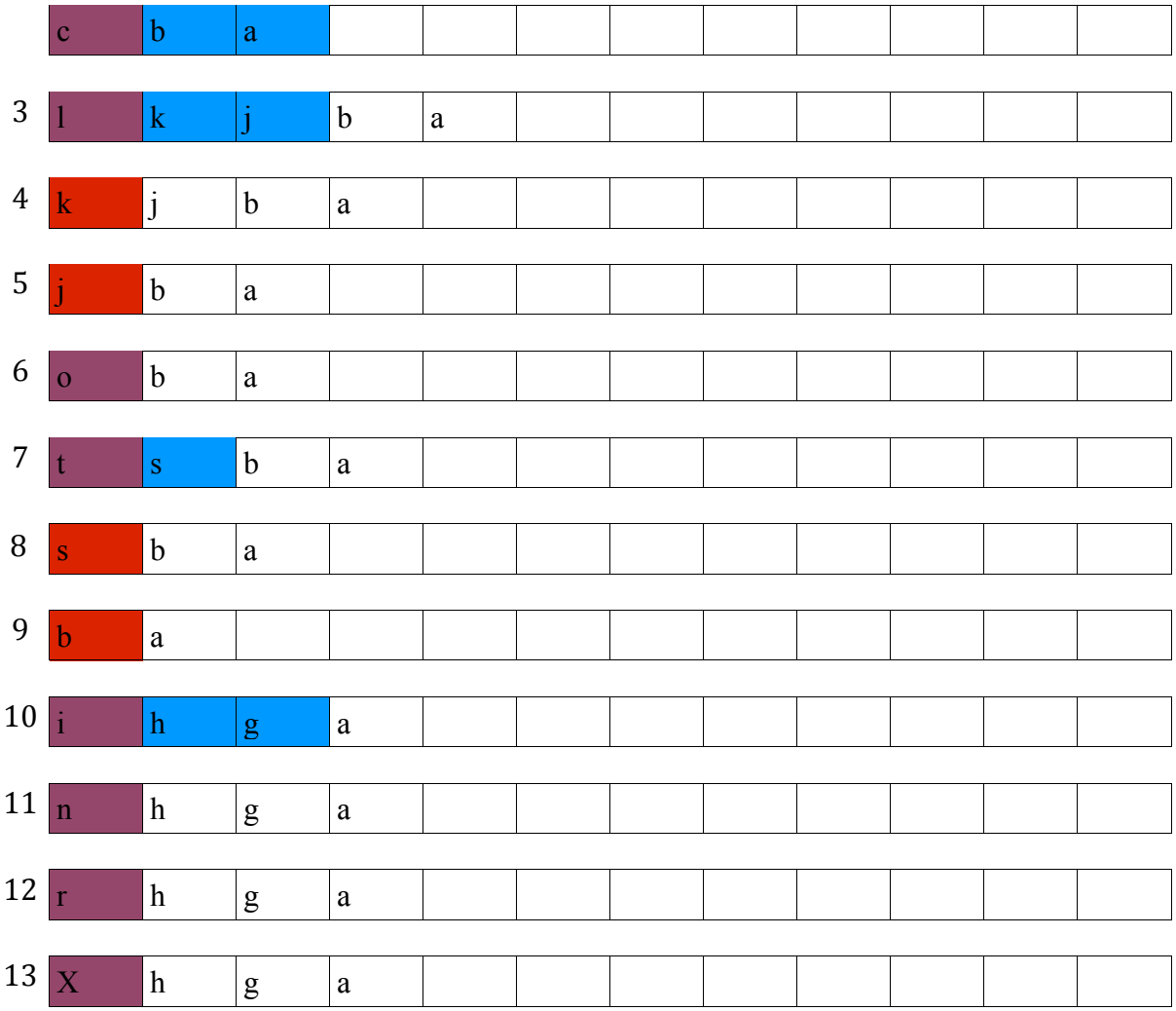

Figure 3

The same color scheme is used here as the Figure 2 for BFS.

This version of uninformed search always explores as deep as possible before looking across at other nodes. It is called *depth-first search* (DFS).

Let's pause to compare BFS and DFS. There are many interesting things going on. First, note that with BFS, the frontier can get pretty long, in this case up to nine nodes at a time. This is because it searches an entire level before going deeper, and the number of nodes at any level n in a tree can be exponential in the depth (2^n if on average each node has two children, and 3^n if three, etc). This poses a problem for BFS, since memory usage can become prohibitive.

On the other hand, precisely because BFS searches an entire level, it will find a goal at that level if there is one, before looking deeper. So: BFS always finds a goal as close to

the start as possible. In particular, it will find a goal if there is one. These two properties are called *optimality* and *completeness*, respectively. So BSF is optimal and complete. Its frontier is like a largely horizontal swath that slowly moves deeper but cuts all the way across from left to right.

BFS is **complete**: if there is a goal, it will be found using BFS.
BFS is **optimal**: it will always find the goal closest to the start.

What about DFS? Well, in our example it did find a goal, but a deep one much farther from the start than we might have hoped. So DFS is not optimal. Worse, it is not even complete: it might not find a goal at all even if there is one. This is because, in general, the search tree can be infinite (typically can always apply actions to get new nodes, deeper and deeper, and there is no guarantee that the search will encounter any of them as it goes deeper while letting any shallower nodes on the frontier simply sit there forever).

DFS is *neither optimal nor complete*.

On the other hand, DFS does not use nearly as much memory; only a relatively small number of higher-up nodes are even generated. Its frontier is like a largely vertical swath that drives deeper and deeper and tends to leave unexplored and even ungenerated lots of the tree close to the start state.

Note that although DFS found the deepest X in our example, this was partly an accident, due to our exploring the deepest frontier node all the way to the left, which was arbitrary (based on the left-right order we happened to use). If instead we had used right-left order, we'd have found the X at level three, and very quickly:

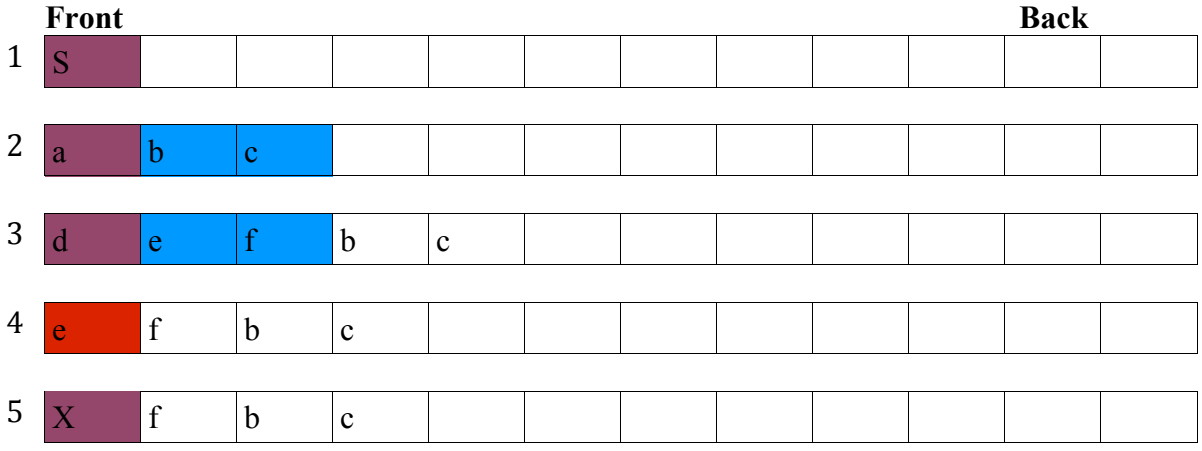

Figure 4

So, unlike BFS, DFS is *very* sensitive to the order in which children are generated (i.e., in which actions are considered).

It would be great to combine the good features of BFS and DFS, if possible. This can be done, in so-called **iterative deepening search (IDS).** Here one essentially uses DFS, but where no nodes below some level n are considered (so the search-tree is finite, given any fixed n, and thus DFS will eventually look at *all* nodes down to that level, if no goal is found first); and this is repeated over and over for n = 1,2,3… until a goal is found during one of the iterations. So while the search goes deep (to level n) like DFS, it also goes broadly across the tree once it bottoms out at level n. (As n increases, it also repeats all the work it did for earlier values of n, starting over from the root each time. This sounds very wasteful, but in fact it is not, since typically most of the nodes in a tree are at deep levels, and repeating work higher up is not very costly.) See Fig 3.18 (p 89) for an example.

A homework problem will give you experience with BFS, DFS, and IDS.

There is a subtlety that may have occurred to you. Why do we bother (in the above algorithms) with nodes that contain world states that have already been looked at in earlier nodes? This seems wasteful, and easy to correct. Indeed it is, as follows (where the change is underlined and in italics):

1. Choose one frontier node and remove it from the frontier (it is now being explored).
2. If its state is a goal state then we are done; else expand it by generating its children and adding *any whose states are unseen* to the frontier *and marking those states as seen*.

The textbook calls this the *graph-version* of the basic search algorithm (as opposed to the tree-search version).

We have not considered the *cost* associated with search, other than time (number of nodes and/or deepest level generated) and space (memory). But actions can differ a great deal in cost, whether computational effort, actual expense, and so on. One way to treat cost is via a cost-function g(n) that represents the cost to get from the start-node S to node n, that is, the cost of the unique path from S to n (why is this unique?). And one way to exploit this is by putting new nodes on the frontier in the order of their cost, with lower costs toward the front (so they are expanded first). One modification is needed, though: when a newly-generated child's state is also the state of a current frontier-node state with a higher cost, then remove the current node and insert the new (cheaper) one.

This is called **uniform-cost search** because it's frontier evolves so that its nodes are the cheapest unexplored ones seen so far. It can be thought of as a penny-pinching algorithm that always proceeds as cheaply as possible. When the cost g(n) is simply the level of n (i.e., how many steps from S to n), then it is the same as BFS.

In the next lecture, we will see that uniform-cost search leads to a natural extension that takes advantage of possible extra information about the states and actions, that allows a search to be a lot “smarter” or “informed” or “heuristic”.