**Lecture 3 (Solving problems by searching)**

This section covers several search strategies that come under the heading of **uninformed search** (also called **blind search**). The term means that the strategies have no additionalinformation about states beyond that provided in the problem definition. All they can do isgenerate successors and distinguish a goal state from a non-goal state. All search strategiesare distinguished by the *order* in which nodes are expanded. Strategies that know whetherone non-goal state is “more promising” than another are called **informed search** or **heuristic search** strategies;

**Breadth-first search** is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. Breadth-first search is an instance of the general graph-search algorithm in which the *shallowest* unexpanded node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is *generated* rather than when it is selected for expansion. This decision is explained below, where we discuss time complexity. Note also that the algorithm, following the general template for graph search, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found. Thus, breadth-first search always has the shallowest path to every node on the frontier.

How does breadth-first search rate according to the four criteria from the previous section?

We can easily see that it is *complete*—if the shallowest goal node is at some finite depth **d**, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor **b** is finite). Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test. Now, the *shallowest* goal node is not necessarily the *optimal* one;

Technically, breadth-first search is optimal if the path cost is a nondecreasing function of the

depth of the node. The most common such scenario is that all actions have the same cost. So far, the news about breadth-first search has been good. The news about time and space is not so good. Imagine searching a uniform tree where every state has b successors. The root of the search tree generates **b** nodes at the first level, each of which generates **b** more nodes, for a total of **b2** at the second level. Each of *these* generates b more nodes, yielding **b3** nodes at the third level, and so on. Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is



(If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be **O(bd+1)**.)

As for space complexity: for any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of **b** of the time

complexity. For breadth-first graph search in particular, every node generated remains in

memory. There will be O(bd−1) nodes in the explored set and O(bd) nodes in the frontier, so the space complexity is O(bd), i.e., it is dominated by the size of the frontier. Switching to a tree search would not save much space, and in a state space with many redundant paths, switching could cost a great deal of time.



An exponential complexity bound such as O(bd) is scary. Figure 3.13 shows why. It

lists, for various values of the solution depth d, the time and memory required for a breadthfirst search with branching factor b = 10. The table assumes that 1 million nodes can be

generated per second and that a node requires 1000 bytes of storage. Many search problems

fit roughly within these assumptions (give or take a factor of 100) when run on a modern

personal computer.



Two lessons can be learned from Figure 3.13. First, *the memory requirements are a bigger problem for breadth-first search than is the execution time.* One might wait 13 daysfor the solution to an important problem with search depth 12, but no personal computer hasthe petabyte of memory it would take. Fortunately, other strategies require less memory.

**The uniform cost search**

When all step costs are equal, breadth-first search is optimal because it always expands the

*shallowest* unexpanded node. By a simple extension, we can find an algorithm that is optimal

with any step-cost function. Instead UNIFORM-COST of expanding the shallowest node, **uniform-cost search** expands the node **n** with the *lowest path cost* g(n). This is done by storing the frontier as apriority queue ordered by g. The algorithm is shown in Figure 3.14.

In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search. The first is that the goal test is applied to a node when it is *selected for expansion* (as in the generic graph-search algorithm shown in Figure 3.7) rather than when it is first generated. The reason is that the first goal node that is *generated* may be on a suboptimal path. The second difference is that a test is added in case a better path is found to a node currently on the frontier.



Both of these modifications come into play in the example shown in Figure 3.15, where

the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97=177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99+211=310. Now a goal node has been generated,

but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+97+101= 278. Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned. It is easy to see that uniform-cost search is optimal in general. First, we observe that whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found. (Were this not the case, there would have to be another frontier node n’ on the optimal path from the start node to n, by the graph separation property of Figure 3.9; by definition, n’ would have lower g-cost than n and would have been selected first.) Then, because step costs are nonnegative, paths never get shorter as nodes are added. These two facts together imply that *uniform-cost search expands nodes in order of their optimal path* *cost.* Hence, the first goal node selected for expansion must be the optimal solution.

Uniform-cost search does not care about the *number* of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions—for example, a sequence of NoOp actions.

**Depth-first search**

**Depth-first search** always expands the *deepest* node in the current frontier of the search tree. The progress of the search is illustrated in Figure 3.16. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.

The depth-first search algorithm is an instance of the graph-search algorithm in Figure 3.7; whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn,

was the deepest unexpanded node when it was selected. As an alternative to the GRAPH-SEARCH-style implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn. (A recursive depth-first algorithm incorporating a depth limit is shown in Figure 3.17.)

The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used. The graph-search version, which avoids repeated states and redundant

paths, is complete in finite state spaces because it will eventually expand every node. The tree-search version, on the other hand, is *not* complete—for example, in Figure 3.6 the algorithm will follow the Arad–Sibiu–Arad–Sibiu loop forever. Depth-first tree search can be modified at no extra memory cost so that it checks new states against those on the path from

the root to the current node; this avoids infinite loops in finite state spaces but does not avoid

the proliferation of redundant paths. In infinite state spaces, both versions fail if an infinite

non-goal path is encountered. For example, in Knuth’s 4 problem, depth-first search would

keep applying the factorial operator forever. For similar reasons, both versions are nonoptimal. For example, in Figure 3.16, depth first search will explore the entire left subtree even if node C is a goal node. If node J were also a goal node, then depth-first search would return it as a solution instead of C, which would be a better solution; hence, depth-first search is not optimal.

The time complexity of depth-first graph search is bounded by the size of the state space (which may be infinite, of course). A depth-first tree search, on the other hand, may generate

all of the O(bm) nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space. Note that m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.

So far, depth-first search seems to have no clear advantage over breadth-first search, so why do we include it? The reason is the space complexity. For a graph search, there is no advantage, but a depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. For a state space with branching factor **b** and maximum depth **m**, depth-first search requires storage of only O(b\*m) nodes.

Using the same assumptions as for Figure 3.13 and assuming that nodes at the same depth as

the goal node have no successors, we find that depth-first search would require 156 kilobytes

instead of 10 exabytes at depth d = 16, a factor of 7 trillion times less space. This has led to the adoption of depth-first tree search as the basic workhorse of many areas of AI.



**Depth-limited search**

The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit **l**. That is, nodes at depth **l** are

treated as if they have no successors. This approach is called **depth-limited search**. The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional

source of incompleteness if we choose **l < d**, that is, the shallowest goal is beyond the depth

limit. (This is likely when d is unknown.) Depth-limited search will also be nonoptimal if

we choose **l > d**. Its time complexity is **O(bl)** and its space complexity is **O(bl)**. Depth-first

search can be viewed as a special case of depth-limited search with l=∞.

Sometimes, depth limits can be based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so l = 19 is a possible choice. But in fact if we studied the map carefully, we would discover that any city can be reached from any other city in at most

9 steps. This number, known as the **diameter** of the state space, gives us a better depth limit,

which leads to a more efficient depth-limited search. For most problems, however, we will

not know a good depth limit until we have solved the problem. Depth-limited search can be implemented as a simple modification to the general tree or graph-search algorithm. Alternatively, it can be implemented as a simple recursive algorithm as shown in Figure 3.17. Notice that depth-limited search can terminate with two kinds of failure: the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit.

**Iterative deepening search** (or iterative deepening depth-first search) is a general strategy,

often used in combination with depth-first tree search, that finds the best depth limit. It does

this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.

This will occur when the depth limit reaches d, the depth of the shallowest goal node. The

algorithm is shown in Figure 3.18. Iterative deepening combines the benefits of depth-first

and breadth-first search. Like depth-first search, its memory requirements are modest: O(bd)

to be precise. Like breadth-first search, it is complete when the branching factor is finite and

optimal when the path cost is a nondecreasing function of the depth of the node. Figure 3.19

**ITERATIVE-DEEPENING-SEARCH** shows four iterations of on a binary search tree, where the solution is found on the fourth iteration. Iterative deepening search may seem wasteful because states are generated multiple times. It turns out this is not too costly. The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times. In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times. So the total number of nodes generated in the worst case is



which gives a time complexity of **O(bd)**—asymptotically the same as breadth-first search. There is some extra cost for generating the upper levels multiple times, but it is not large. For

example, if b = 10 and d = 5, the numbers are



If you are really concerned about repeating the repetition, you can use a hybrid approach that runs breadth-first search until almost all the available memory is consumed, and then runs iterative deepening from all the nodes in the frontier. *In general, iterative deepening is* *the preferred uninformed search method when the search space is large and the depth of the*

*solution is not known.*

Iterative deepening search is analogous to breadth-first search in that it explores a complete

layer of new nodes at each iteration before going on to the next layer. It would seem worthwhile to develop an iterative analog to uniform-cost search, inheriting the latter algorithm’s

optimality guarantees while avoiding its memory requirements. The idea is to use increasing path-cost limits instead of increasing depth limits.

The idea behind bidirectional search is to run two simultaneous searches—one forward from

the initial state and the other backward from the goal—hoping that the two searches meet in

the middle (Figure 3.20). The motivation is that bd/2 + bd/2 is much less than bd, or in the figure, the area of the two small circles is less than the area of one big circle centered on the

start and reaching to the goal. Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found. (It is important to realize that the first such solution found may not be optimal, even if the two searches are both breadth-first; some additional search is required to make sure there isn’t another short-cut across the gap.) The check can be done when each node is generated or selected for expansion and, with a hash table, will take constant time. For example, if a problem has solution depth d=6, and each direction runs breadth-first search one node at a time, then in the worst case the two searches meet when they have generated all of the nodes at depth 3. For b=10, this means a total of 2,220 node generations, compared with 1,111,110 for a standard breadth-first search. Thus, the time complexity of bidirectional search using breadth-first searches in both directions is O(bd/2). The space complexity is also O(bd/2). We can reduce this by roughly half if one of the two searches is done by iterative deepening, but at least one of the frontiers must be kept in memory so that the intersection check can be done. This space requirement is the most significant weakness of bidirectional search.



The reduction in time complexity makes bidirectional search attractive, but how do we search backward? This is not as easy as it sounds. Let the **predecessors** of a state x be all those states that have x as a successor. Bidirectional search requires a method for computing

predecessors. When all the actions in the state space are reversible, the predecessors of x are

just its successors. Other cases may require substantial ingenuity.